Proceedings of ELS 2011

4th European Lisp Symposium

Special Focus on Parallelism and Efficiency

March 31 – April 1 2011
TUHH, Hamburg, Germany

Preface

Message from the Programme Chair

Welcome to ELS 2011, the 4th European Lisp Symposium.

In the recent years, all major academic events have suffered from a decreasing level of attendance and contribution, Lisp being no exception to the rule. Organizing ELS 2011 in this context was hence a challenge of its own, and I'm particularly happy that we have succeeded again.

For the first time this year, we had a "special focus" on parallelism and efficiency, all very important matters for our community with the advent of multi-core programming. It appears that this calling has been largely heard, as half of the submissions were along these lines. Another notable aspect of this year's occurrence is the fact that four dialects of Lisp are represented: Common Lisp, Scheme, Racket and Clojure. This indicates that ELS is successful in attempting to gather all crowds around Lisp "the idea" rather than around Lisp "one particular language". The European Lisp Symposium is also more European than ever, and in fact, more international than ever, with people coming not only from western Europe and the U.S.A., but also from such countries as Croatia and Bulgaria.

While attending the symposium is just seeing the tip of the iceberg, a lot have happened underwater. First of all, ELS 2011 would not have been possible without the submissions we got from the authors and the careful reviews provided by the programme committee members; I wish to thank them for that. I am also indebted to the keynote speakers who have agreed to come and spread the good word. I wish to express my utmost gratitude to our sponsors who contributed to making the event quite affordable this year again. Ralf Möller was our local chair, the "Grey Eminence" of the symposium, and we owe him a lot. Finally, my thanks go to Edgar Gonçalves for taking care of the website with such reactivity and attentiveness.

I wish you all a great symposium!

Message from the Local Chair

Welcome to Hamburg University of Technology (TUHH). We hope you will enjoy your stay at our university for the 2011 European Lisp Symposium. Not only interesting presentations will be part of the programme, but also social events such as the the social dinner at the Feuerschiff (`http://www.das-feuerschiff.de`) and the Welcome Reception at Freiheit (`http://www.freiheit.com`). We would like to thank all sponsors for making the event possible. For those of you staying over the weekend, a tour to Miniatur-Wunderland (`http://www.miniatur-wunderland.de`) will be offered.

Yours sincerely,

Ralf Möller

TUHH (`http://www.tuhh.de`) is a competitive entrepreneurial university focussing on high-level performance and high quality standards. TUHH is dedicated to the principles of Humboldt (unity of research and education). TUHH has a strong international orientation and also focusses on its local environment. It contributes to the development of the technological and scientific competence of society, aiming at excellency at the national and international level in its strategic research fields, and educating young scientists and engineers within demanding programmes using advanced teaching methods.

Let's not forget Hamburg, for why are we all here? People say Hamburg is Germany's most attractive city combining urbane sophistication, maritime flair, the open-mindedness of a metropolis, and high recreational value. The second-largest city in Germany, it has traditionally been seen as a gateway to the world. Its port is not only the largest seaport in Germany and the second-largest in Europe, but a residential neighborhood with leisure, recreational and educational facilities. Hamburg is a very special place.

Organization

Programme Chair

- Didier Verna, EPITA Research and Development Laboratory, France

Local Chair

- Ralf Möller - Hamburg University of Technology, Germany

Programme Committee

- António Leitão, Instituto Superior Técnico/INESC-ID, Portugal
- Christophe Rhodes, Goldsmiths College, University of London, UK
- David Edgar Liebke, Relevance Inc., USA
- Didier Verna, EPITA Research and Development Laboratory, France
- Henry Lieberman, MIT Media Laboratory, USA
- Jay McCarthy, Brigham Young University, USA
- José Luis Ruiz Reina, Universidad de Sevilla, Spain
- Marco Antoniotti, Universita Milano Bicocca, Italy
- Michael Sperber, DeinProgramm, Germany
- Pascal Costanza, Vrije Universiteit of Brussel, Belgium
- Scott McKay, ITA Software, USA

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

NovaSparks
86 Sherman Street
Cambridge, MA 02140
USA
www.hpcplatform.com

Freiheit Technologies gmbh
Straßenbahnring 22
20251 Hamburg
Germany
www.freiheit.com

TUHH
Schwarzenbergstraße 95
D-21073 Hamburg
Germany
http://www.tu-harburg.de

Contents

Invited Talks

Compiling for the Common Case

Craig Zilles, University of Illinois, USA

Microprocessor vendors are actively exploring mechanisms that offer the potential to reduce the effort to produce parallel code. One such mechanism, is the ability to atomically execute code which is useful for accelerating critical sections, lock-free data structures, and for implementing transactional memory. With 3 prior implementations (Transmeta's Crusoe, Azul's Vega, and Sun's Rock) this mechanism has a lot of potential to be ubiquitous in the next decade. In this talk, I'll discuss how this mechanism can be re-purposed to provide very efficient user-mode checkpoint/rollback, allowing a compiler to generate "speculative" versions of code that support only the expected case. I'll detail our experiences exploring compiling in such an environment in the context of an x86 binary translator, a Java virtual machine, and the Python dynamic language.

Reconfigurable Computing on Steroids

Marc Battyani, NovaSparks

General purpose CPUs have been hitting the frequency wall but as the number of transistors in electronic chips continues to steadily increase, there is a tremendous need for other computing paradigms. One of them is the use of reconfigurable hardware (FPGA) to accelerate specific kinds of computations. Even though the performance gain can be huge, FPGAs are notoriously very difficult to program, which has been one of the major drawbacks in their adoption. There have been several attempts to solve this problem using C to VHDL/Verilog compilers. Though this can be useful at times, our opinion is that it is not a good approach. In this talk, we will explain how and why we use domain specific languages that enable us to generate high performance Domain Specific Hardware optimized for the final tasks being implemented. We will also present our experience at NovaSparks where we have been using Common Lisp to successfully define and implement those DSL->DSH compilers in financial applications since 2007.

Scala: an Object-Oriented Surprise

Apostolos Syropoulos

Scala is an Object Oriented language that was released in 2003. The distinguished features of Scala include a seamless integration of functional programming features into an otherwise OO language. Scala owes its name to its ability to scale, that is, it is a language that can grow by providing an infrastructure that allows the introduction of new constructs and data types. Scala is a compiled language. Its compiler produces bytecode for the Java Virtual Machine, thus, allowing the (almost) seamless use of Java tools and constructs from within Scala. Most importantly, Scala is a concurrent programming language, thus, it is a tool for today as well as tomorrow.

Session I: Parallelism

Supercomputing in Lisp

Porting SBCL to IBM Blue Gene/P

Valentin Pavlov
Rila Solutions EAD
vpavlov@rila.bg

Abstract

This paper describes the technical details of the process of porting SBCL (Steel Bank Common Lisp) to IBM Blue Gene/P, making it the first Lisp implementation to run on a modern (circa 2010) peta-scale supercomputing platform. We also discuss current limitations of the port and outline some of the future work needed in order to make Lisp an attractive choice for peta-scale application development.

Categories and Subject Descriptors C.5.1 [*Computer Systems Implementation*]: Large and Medium ("Mainframe") Computers—Super (very large) computers; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Languages

Keywords Blue Gene/P, supercomputing, Lisp, SBCL

1. Introduction

For several decades now the research community has been relying on scientific packages written over the years in FORTRAN, C and C++. Most of these are easily ported to modern supercomputing systems, which provide cross-compilers and standard, well-established mechanisms for IPC (usually MPI and OpenMP). However, high-level abstraction languages – as Lisp – are virtually non-existent in the supercomputing domain. The reasons for this are no different than the reasons why we don't see Lisp in mainstream software development – mostly historical and most of them wrong. In our opinion, if we want to move forward and start creating better parallel applications, we have to look into high-level abstractions languages. Solid parallel Lisp development and runtime environments would help future researchers to focus at the problem at hand and step away from tedious low-level infrastructural details. The dynamic nature of Lisp seems very relevant for creating self-adapting, smart and scalable applications that are needed at the peta-scale performance level. At the same time, binding to native libraries will guarantee efficient execution of critical calculations for which robust and proven mathematical packages exist.

The project described in this paper is an attempt to bring Lisp to the modern supercomputing stage. A supercomputing Lisp runtime environment is absolutely necessary in order to make people start considering Lisp as a feasible language for parallel application development. The lack of such runtime environment would mean that their code will only run on small-to-medium cluster installations, which would make it inferior to an application written in C/C++ or FORTRAN. To the best of our knowledge, such environment did not exist, at least not one suitable for supercomputers that are in circulation these years.

With this in mind, the main goal of the project (and the paper) is **to act as enabler** and to show that such environment can be build and in fact exists. Even though the majority of the developers might not have access to supercomputers, just the knowledge that their program can be made to run on one can provide them with enough confidence in order to start writing their next parallel program in Lisp. This put Lisp on equal footing with C/C++ and FORTRAN, at least in regards with the availability of the language for peta-scale development.

One of the planned uses for this environment is to run the Quantum Computing simulator application described in [14], but this is by far a lesser goal. Since Lisp is a general programming language, there is no limit for the practical usefulness of such environment. The domain of possible applications is not restricted in any way.

The rest of the paper is organized as follows: Section 2 introduces some aspects of the the target system organization that are relevant to the discussions that follow. Section 3 describes the process of porting SBCL to IBM BlueGene/P. Section 4 describes the current status of the port and provides a list of known limitations. Section 5 outlines directions for future work in regards with the port and also in regards to general parallel Lisp programming. Section 6 gives a brief overview of related work and Section 7 concludes the paper.

2. Target system organization

The IBM Blue Gene/P supercomputer is a massively parallel distributed memory system that supercedes the company's previous line of supercomputers, BlueGene/L [12]. It is designed to reach a maximum theoretical performance of 1 PFLOPS, utilizing 294,912 PowerPC cores @ 850 MHz with 144 TB of RAM. We will now briefly describe some system details that are relevant to our discussion.

2.1 Front-end Node

The Front-end Node (FEN) is the system's main entry point. This is where its users login, prepare their work, submit their jobs and receive their results. This is a 4-CPU 2GHz POWER5+ machine running a ppc64 SuSE Linux Enterprise Server 10 operating system. Apart from the software packages found in the regular SLES 10 distribution, the machine also includes:

ELS'2011 TUHH, Hamburg, Germany.
[European Lisp Symposium].

- a cross-compiling GCC 4.1.2 toolchain that targets the computing nodes. C, C++ and FORTRAN compilers are available;
- a cross-compiling GCC 4.3.2 toolchain that supports OpenMP (experimental);
- the IBM XL suite of compilers for C, C++ and FORTRAN. Two sets of compilers are provided – one that produces code to be run on the FEN, and another that is cross-compiling for the computing nodes. The XL compilers also support OpenMP;
- Argonne National Laboratory's MPICH2 [4] implementation of the MPI standard, cross-compiled and ready to be used by the applications running on the computing nodes;
- TWS LoadLeveler – the resource scheduler that organizes job submission, queuing and execution;

2.2 Compute Nodes

The Compute Nodes (CN), as their name suggest, provide the main computing power of the system. Each CN has 4 PowerPC 450 cores @ 850 MHz and 2 GB RAM[1]. Each core has 2 FPUs (double hummer), and each FPU is capable of processing 2 instructions per cycle. This amounts to a theoretical performance of 13.6 GFLOPS per CN. It should be noted however, that only the IBM XL compilers know about the double hummer – the GCC toolchain cannot produce code that utilizes the two FPUs.

Each CN features several networks through which it is directly connected to its immediate neighbors, forming a large 3D cube of interconnected CNs. 1024 CNs are bundled in a rack, and a particular installation may be as small as 1 rack and as big as 72 racks. The total combined power of all CNs in the latter case is 979.2 TFLOPS, or almost 1 PFLOPS.

The CNs are booted on demand. During job startup, a partition with the required size is allocated and all CNs in it start booting a custom, IBM-proprietary kernel, called CNK (Compute Node Kernel). CNK provides a single-process environment – only a single application process runs on top of it; luckily, dynamic loading of shared libraries is supported. When all CNs in the partition boot, they start to execute the program in parallel, all using the same executable image.

From the application's point of view, CNK behaves more or less like a regular Linux kernel. That is, CNK exports a large subset of the syscalls found in Linux. However, some of the syscalls and libc functions are not supported. One such example is `fork`, which contradicts to the single-process nature of the CNK.

A job can be run in 3 different modes:

- VN mode – each core runs its own process and has direct access to 512 MB RAM;
- DUAL mode – each couple of cores on a CN share an application image and have access to 1 GB RAM. The application process may have 2 parallel threads of execution in a shared-memory environment.
- SMP mode – all 4 cores on a CN share a single application image and have to the whole 2GB RAM on the node. The application process may have 4 threads.

Single-threaded applications should be run in VN mode, otherwise half or even three-quarters of the processing power will be wasted. This reduces the amount of memory directly available to single-threaded applications to 512 MB per process.

2.3 I/O Nodes

The I/O nodes (ION) are the interfaces through which the compute nodes interact with the rest of the system and the world. They are connected to a 10 Gbit/s switch that also connects the Front-end Node, File Servers and all the rest of the system.

The IONs boot a Linux-based OS. It uses network file system drivers (NFS, GPFS) to connect to any file servers. An important piece of software – the Console I/O Daemon (CIOD) – acts as a proxy between the compute nodes and rest of the world. All input and output operations issued on a CN eventually end up carried out by one of the IONs. This includes File I/O, stdin, stdout, stderr, networking, etc.

The proxying is transparent for the application, but ultimately it involves communication between the kernel of the computing node and the corresponding CIOD, using an in-house protocol. As discussed later, this has some implications, one of them being the fact that stdin does not support non-blocking reads.

2.4 Other components and features

Other components The system also includes other components like: the Service Node, through which the actual monitoring and control of the bare metal is done; a storage sub-system that comprises several file servers plus a centralized storage device; and some networking infrastructure that connects all this to the Internet. These however don't play any important role in the porting process and will not be discussed in detail. Interested readers are advised to review [11] and [17].

HPC vs. HTC A computing node block can be booted in one of two possible modes: HPC (High-Performance Computing), in which all the CNs in the block start the same process and work cooperatively using MPI; and HTC (High-Throughput Computing), in which the CNs in the block may run different processes and MPI is not available.

CNK Source Code Argonne National Laboratory and IBM sponsored the BG/P open source project [1], which provides access to the source code of all necessary components. This turned out to be very helpful during the port process.

3. Lisp for the CNK

In order to make use of the huge processing power of the supercomputing system, we would need a Lisp implementation that executes on the compute nodes under the control of the CNK. The following sub-sections describe the steps we followed, the difficulties we have experienced and the workarounds that we came up with, in order to make this possible.

3.1 Implementation of choice

The most important criteria for choosing the implementation of choice were: 1) the availability of a Linux/PPC port which to use as a stepping stone; 2) native code compilation and 3) personal familiarity with the implementation. The native code compilation was required in order to silence the opponents of the project, whose main concern is that running an interpreted environment on a supercomputer will be an awful waste of resources. Obviously the other two criteria were required in order to reuse as much knowledge as possible and to complete the project in feasible time.

SBCL [5] fits the above requirements and thus it was chosen to be the implementation of choice. Undoubtfully, there are many other implementations for which a porting attempt could be made in the future.

An excellent reading material regarding the SBCL build system intricates is to be found in [15]. Readers who seek to obtain in-depth understanding of what follows might want to read this article, the SBCL user manual [7] and the SBCL Internals wiki [8]

Having in mind the system's organization, the first thing that becomes obvious is that we will be working in a multi-homed

[1] There are also configurations with 4GB RAM per computing node

environment: first, we will need a Lisp that runs on the FEN, which we will use as a cross-compiler in order to produce a Lisp that runs on the CNs. Although both the front-end and the computing nodes are PowerPC-based, these actually have different architectures: the FEN is 64-bit platform and the CNs are 32-bit. There is also a difference in the instruction sets between the POWER5+ CPU of the FEN and the PowerPC 450 on the computing node. In general, a program compiled for the FEN will not run on the CN and vice versa.

3.2 Compiling for the FEN

As expected, compiling for the FEN turned out to be absolutely straightforward – we just got the latest Linux/PPC binary from SBCL's web site, installed it on the FEN, then grabbed the latest SBCL source code and built it. The new installation would become the `SBCL_XC_HOST` for building the CNK target.

3.3 Tweaking the build system

It is desirable to be able to build both targets from a single source code tree, so we needed a way to tell the build system which target we want. We decided to use an environment variable, `BGPCNK`, for that:

```
~/sbcl-1.0.46> sh make.sh             # FEN build
~/sbcl-1.0.46> BGPCNK=1 sh make.sh    # CN build
```

Inside the shell scripts that make up the build system, we use this variable to control certain aspects of the build, so as to produce one target or the other. Here is an example from `make-config.sh`[2]:

```
if [ $BGPCNK = 1 ]; then
    printf ' :bgpcnk' >> $ltf
fi
```

This particular code leads to the insertion of the keyword `:bgpcnk` into `local-target-features.lisp-expr`, which in turn will allow the cross-compiler to understand that it is compiling for the CNK. This will later get automatically included in `*features*`, so we will know we are running on the CNK. The presence or absence of other local target features is controlled in a similar fashion. For example, due to reasons explained later, the generational garbage collector cannot be used for the CNK build, so in that case it is excluded from the features.

Another thing that the build system does is to select which configuration file to use for building the C runtime and create a soft link to it. Again, this is controlled by the `BGPCNK` variable:

```
if [ $BGPCNK = 1 ]; then
    link_or_copy Config.ppc-linux-bgpcnk Config
else
    link_or_copy Config.$sbcl_arch-linux Config
fi
```

The `Config.ppc-linux-bgpcnk` file is based on the standard Linux/PPC config, but declares that the cross-compiler shall be used when compiling the C runtime.

Setting `:bgpcnk` in the local target features also leads to the definition of `LISP_FEATURE_BGPCNK` during the genesis phase, so when the time comes for compiling the C runtime, we will have this symbol defined and conditional compilation can be based on it.

With these changes the stage is set and with issuing `BGPCNK=1 sh make.sh` we started the CNK build process. It progressed fine through the `make-host-1`, `make-target-1` and `make-host-2` stages, all of which are done on the FEN. The next stage, `make-target-2`, starts the newly build C runtime on the CNK, using the automatic HTC submission mode. This is where the real problems start.

[2] Source code exempts are provided for illustration purposes only; they are modified to make them less verbose and fit the paper margins.

3.4 `mmap` problems

The first problem was related with some peculiarities of the `mmap` syscall implementation in the CNK. SBCL uses a hard-coded configuration of the places where its memory segments should be (`src/compiler/ppc/params.lisp`). When the runtime allocates the segments, it passes their locations as address hints to the corresponding `mmap` calls. Here's the relevant source code (`src/runtime/linux-os.c`):

```
os_vm_address_t
os_validate(os_vm_address_t addr,
            os_vm_size_t len)
{
    int flags =  MAP_PRIVATE | MAP_ANONYMOUS |
                 MAP_NORESERVE;
    os_vm_address_t actual;
    ...
    actual = mmap(addr, len, OS_VM_PROT_ALL,
                  flags, -1, 0);
    ...
    if (addr && (addr!=actual)) {
        fprintf(stderr,
                "mmap: wanted %lu bytes at %p,
                 actually mapped at %p\n",
                (unsigned long) len, addr,
                actual);
        return 0;
    }
    ...
    return actual;
}
```

It turned out that CNK also requires `MAP_FIXED`, but only in case `addr` is not `NULL`. This was easily fixed by adding this before the `mmap` call:

```
#ifdef LISP_FEATURE_BGPCNK
    if(addr) {
        flags |= MAP_FIXED;
    }
#endif
```

However, even with `MAP_FIXED`, CNK treats `addr` only as a hint and it will always use the lowest free address. Thus, it took quite some time to find the right values to put in `params.lisp`, having in mind that any experiment required compilation for about 20-40 minutes. Eventually we figured out the right values and that took care of the `mmap` problems. The runtime started loading the core and then it hanged indefinitely at some unknown place.

3.5 `context` hunting

After an unrecorded number of remote debugging sessions, it turned out that the hanging occurs during an allocation trap. When SBCL needs to allocate memory for a Lisp object, it goes like this: if there is enough free space in the dynamic memory segment, it chops a piece from there; otherwise, it uses a trap instruction to issue a SIGTRAP, which is caught by a signal handler in the C runtime. The signal handler enlarges the dynamic memory segment and jumps 4 instructions after the original trap instruction. The Lisp portion of this mechanism is in the `allocation` macro in `src/compiler/ppc/macros.lisp`:

```
(inst add ,result-tn ,result-tn ,temp-tn)

;; result-tn points to the new end of the region.
;; Did we go past the actual end of the region?
;; If so, we need a full alloc.

(inst tw :lge ,result-tn ,flag-tn)  ; ----> to C

;; These 3 instructions (lr is 2) execute
;; when the trap is NOT fired
(inst lr ,flag-tn (make-fixup
                   "boxed_region" :foreign))
(inst stw ,result-tn ,flag-tn 0)

;; Should the allocation trap above have fired,
;; the runtime arranges for execution to resume
;; here

;; from C <----

(inst sub ,result-tn ,result-tn ,temp-tn)
```

We managed to isolate the place where the hanging occurs, and that is the `tw :lge` instruction that fires the SIGTRAP. Moreover, we found out that the signal handler does get invoked, but something goes wrong in the return sequence.

The signal handler relies on the signal context in order to find the address of the trapping instruction. This is the value of the Program Counter (PC) register from the context. It then uses this address to figure out the cause of the trap and to get the size of the required allocation from the parameters of the previous instruction. When everything is done, it increments the PC in the signal context with 4 instructions, setting up the proper return address. Relevant code in `src/runtime/ppc-arch.c` is:

```
static void
sigtrap_handler(int signal, siginfo_t *siginfo,
                os_context_t *context)
{
    unsigned int code;

    code=*((u32 *)(*os_context_pc_addr(context)));
    ...
    if (allocation_trap_p(context)) {
        handle_allocation_trap(context);
        return;
    }
    ...
}

void
handle_allocation_trap(os_context_t * context)
{
  ...
  (*os_context_pc_addr(context)) = pc + 4; // !!!
}
```

Our experiments showed that the reason for hanging was that upon return from the signal handler, the PC was **not** updated. This led to the execution of the same trap instruction, the same signal handler and so on *ad infinitum*. How was that possible? By single stepping we clearly saw that the context was updated right at the end of the signal handler. So something wrong was happening in the gray area between the end of the signal handler and the signal comeback. Unfortunately this gray area is in CNK, which is in privileged memory and thus cannot be single stepped.

This was very frustrating and almost became a show stopper. We had to reverse engineer the CNK, which was not possible not because of technical, but due to legal reasons. Fortunately, then we came upon the BG/P Open Source project [1], and suddenly everything became clear.

In the primordial signal handler, the CNK creates a copy of the original signal context and it is this copy that gets passed to the user-space signal handler. Then, in `sigreturn`, the CNK does not even look at the context that was passed to the user-space handler; it just restores the state using the original one. We were not working with the real thing, just with a mere shadow of it. We immediately dubbed the One True context *Amber*, after Zelazny's marvelous novel [20].

All this happens on the stack, and Amber was located further down relative to the address of our context. To make matters worse, due to reasons related with cache lineup this relative offset was not constant.

The first thing we now needed to do is to find Amber. By looking at the CNK source code we figured out where to find it – 2K after the end of our context – and we know that the cache lineup may introduce additional 0 to 7 words. We know what it should contain, and that is an exact copy of the 32 general purpose registers, whose values we have from our context. Thus, the function looks like this:

```
#ifdef LISP_FEATURE_BGPCNK
void *find_amber(os_context_t *context)
{
    // Amber is @ 2K after the end of
    // our copy (sizeof(mcontext_t) + 2K)
    int offset = 0x4a0 + 0x800;

    void *ptr1 = context->uc_mcontext.uc_regs->
                      gregs;
    void *ptr2 = ((void *)context) + offset;
    int sz = 128; // NUM_GPRS * sizeof(uint32_t)
    int i;

    for(i=0;i<8;i++) {
        if(!memcmp(ptr1, ptr2 + i*4, sz)) {
            return ptr2 + i*4;
        }
    }
    lose("find_amber: something's wrong");
}
#endif
```

We can use this function at the beginning of all signal handlers in order to find the address of the real context. However, we cannot work with this address directly, since the real context does not adhere to the `mcontext_t` structure. Thus, we decided to proceed as follows: at the beginning of the signal handler, we find the address of Amber and store it; then after the end of the signal handler, we copy the contents of the false context into Amber:

```
#ifdef LISP_FEATURE_BGPCNK
void moveto_amber(void *amber,
                  os_context_t *context)
{
    // 32 GPRs
    void *where = amber;
    memcpy(where,
           os_context_register_addr(context, 0),
           128);

    // FPSCR
    where += 640;
    memcpy(where,
```

```
            &(context->uc_mcontext.uc_regs->
                  fpregs.fpscr),
            sizeof(double));

    // PC
    where += sizeof(double);
    memcpy(where, os_context_pc_addr(context), 4);

    // LR
    where += 12;
    memcpy(where, os_context_lr_addr(context), 4);
}
#endif
```

Now the only thing that remains is to wrap all signal handlers in a kind of `:around` functions, like this:

```
#ifdef LISP_FEATURE_BGPCNK
static void
sigtrap_handler_bgpcnk(int signal,
                       siginfo_t *siginfo,
                       os_context_t *context)
{
    void *amber = find_amber(context);
    sigtrap_handler(signal, siginfo, context);
    moveto_amber(amber, context);
}
#endif
```

This solved the problem. The runtime loaded `cold-sbcl.core` and started to execute it, and all was OK until the first garbage collection occurred.

3.6 `mprotect` and `munmap` misbehavior

SBCL's garbage collection (GC) comes in two flavors: the original Cheney GC and a newer, Generational Conservative GC (gencgc). The default mechanism for the Linux/PPC architecture is gencgc, which is superior in performance. Since we were basing the CNK build on the Linux/PPC architecture, this was our default choice, too.

It is beyond the scope of this article to explain how garbage collection works. A proper treatment of the subject appears in [19]. Here we will only give a brief description of some of the ideas that will support the understanding of the problems encountered.

gencgc is based on the empirical observation that the most recently created objects are the ones that are most likely to become unreferenced quickly. The longer an object survived, the larger the probability for it to survive a little bit more. This hypothesis reflects the largely stack-based organization of modern computing. In order to benefit from this observation, gencgc divides objects into *generations*. Newly created objects are placed in the youngest generation. On each GC cycle, surviving objects from generation N are moved into generation $N + 1$. There are two types of cycles: minor cycle, in which only the youngest generation is collected[3]; and major cycle, in which all generations are collected and surviving objects are compacted in the second generation. After each cycle the youngest generation is empty.

Note that objects from a younger generation may be referenced by objects from older generations, and thus it seems that the GC must sweep through all generations anyway. It is important to understand how these cross-generation references appear, in order to come up with the optimization that makes generational garbage collection much faster during minor cycles. After a GC cycle the youngest generation is empty, so there are no objects in the older generations that may reference an object to be collected in the next minor cycle. The only way that such reference might occur is by creating a new object (which ends up in the youngest generation) and then creating a reference to it in some older generation object. This reference creation involves *writing* to some memory location belonging to an older generation.

The optimization then becomes apparent. At the end of a cycle, the GC sets up a *write barrier* – it write-protects all memory belonging to older generations. Thus, if a memory write occurs in a protected page, a signal is raised. The signal handler un-protects the corresponding memory page and proceeds with the writing. At the next minor cycle, all pages that are still write-protected are *guaranteed* to not have been written to, so then don't have any references to the youngest generation. Only unprotected memory pages are swept. This mechanism is crucial for the performance of the generational garbage collection. If it were not in place, it would in fact take more time to perform a cycle that a regular GC, due to the increased complexity and virtually no other benefit.

The write barrier is set using the OS `mprotect` syscall. It took us considerable amount of time to get to know the generational garbage collection and hack through it, until we finally realized that CNK's `mprotect` syscall does nothing – it's just an empty stub that returns 'no error'. This means that memory writes to older generation objects proceed normally and the signal handler that un-protects the memory pages is never called. This then leads to the fact that, to the best of gencgc's knowledge, all pages in the older generations remain 'write-protected' and thus are never swept. And that means that objects from the youngest generation that are only referenced by older generations are collected, which leaves these references dangling.

The first thing we did after finding out about this `mprotect` misbehavior, was to turn off the optimization and pretend that all pages need sweeping. As expected, gencgc started working correctly, but unacceptably slower. So, we decided to switch to Cheney GC.

In the Cheney GC scheme [9], the available memory is divided into two equal semi-spaces, only one of which is used at any one time. Garbage is collected by copying referenced objects from the live semi-space to the spare one. The semi-spaces then switch roles and the entire spare half is cleared *en bloc*.

Important decisions in this scheme are when and how to trigger GC. In the SBCL's implementation the 'when' part is decided based on some heuristic that relates to how much memory was cleared during the last GC run and some other parameters. The exact details are not related to the scope of this article. It is the 'how' part that turned up very relevant. At the end of each GC cycle, the heuristic sets a high water mark, the `current_auto_gc_trigger`, and the GC *write-protects* all the memory above the mark. Object allocation above this mark leads to a memory write, which raises a signal that triggers the garbage collection.

So it turned out that GC triggering relies on the same broken `mprotect` and as a consequence never happens. A brief glint of hope proved to be this piece of code from `src/runtime/cheneygc.c`:

```
#if defined(SUNOS) || defined(SOLARIS)
    os_invalidate(addr, length);
#else
    os_protect(addr, length, 0);
#endif
```

which shows that other operating systems have the same problem and it is solved by just invalidating the pages above the high water mark. Unfortunately, this option was quickly dismissed after observing that `munmap` also does nothing useful and user-space

[3] Technically, more than one generation may be collected on a minor cycle, but SBCL's gencgc collects only the youngest one.

programs can read and write unmapped memory in any way they want, without raising a signal.

We decided to tackle the problem by hacking the `allocation` macro. When an object is being allocation, the code would first check whether it would end up above the high water mark, and if so it would issue a trap that will trigger the GC (`src/compiler/ppc/macros.lisp`):

```
#!+bgpcnk
(let ((,label1 (gen-label)))
  (inst lr ,temp-tn (make-fixup
                     "current_auto_gc_trigger"
                     :foreign))
  (inst lwz ,temp-tn ,temp-tn 0)
  (inst cmpwi ,temp-tn 0)
  (inst beq ,label1)
  (inst cmpw alloc-tn ,temp-tn)
  (inst bng ,label1)
  (inst unimp force-gc-trap)  ;; ---> GC
  (emit-label ,label1))
```

The `force-gc-trap` is a new trap identifier we defined in `src/compiler/ppc/params.lisp` whose presence is checked in the SIGTRAP allocation and if found, leads to triggering the GC. This approach worked and the garbage collection began proceeding as expected.

3.7 `select` problems

The next problem we experienced was related to the special treatment of `stdin` in the CNK. As mentioned earlier, all I/O, including stdio, proceeds through a proxying mechanism in which the compute node kernel communicates with the CIOD on the I/O node using an internal protocol. Reading from `stdin` and writing to `stdout` and `stderr` is treated differently from reading and writing to/from ordinary files. A crucial difference is that the `select` syscall is not working for `stdin` and `stdout` – it just returns an `EINVAL` error. This leads to the inability to perform non-blocking reads, which messes up `sysread-may-block-p` for `stdin`, which is called early during the SBCL warm up procedure.

We tried without success different approaches to overcome this issue and finally we decided to introduce an additional command line argument, `--stdin <fname>`, to the SBCL runtime. The runtime would then open the file and make the `*stdin*` fd-stream use its descriptor instead of the standard 0. Thus, `*stdin*` would ultimately point to a file, and `select` worked fine for files. Here are the relevant portions of the code (`src/runtime/runtime.c`):

```
#if defined LISP_FEATURE_BGPCNK
int stdin_fd = 0;
#endif
...
#if defined (LISP_FEATURE_BGPCNK)
  }else if (0 == strcmp(arg, "--stdin")) {
    ++argi;
    if (argi >= argc) {
      lose("missing argument for --stdin");
    }
    stdin_fd = open(argv[argi++], O_RDONLY);
    if(stdin_fd==-1) {
      lose("cannot open stdin replacement");
    }
#endif
```

And in `src/code/fd-stream.lisp`:

```
#!+bgpcnk
(sb!alien:define-alien-variable
  ("stdin_fd" *stdin-fd*) int)
...
(setf *stdin*
      (make-fd-stream
        #!-bgpcnk 0 #!+bgpcnk *stdin-fd*
        :name "standard input" :input t
        :buffering :line
        :element-type :default
        :serve-events t
        :external-format
          (stdstream-external-format nil)))
```

This solved the problem and the runtime started processing `make-target-2.lisp` and `make-target-2-load.lisp`, and finally `make-target-2.sh` was all done. This completed the main build process.

3.8 Enter the REPL

Eager to try the new build, we run `./run-sbcl.sh` and found out that `getpwd` was not working. We quickly located the root cause of the problem – for some unknown reasons `calloc` (used in `wrap.c`) ended up allocating 4096 bytes, but zeroing out 4108 (?!?). We decided not to research any further and instead replace the `calloc` call with `malloc`, followed by `memset`. After invoking `./run-sbcl.sh` again, we saw the REPL and declared major success.

3.9 Switching to dynamic linking

The default linking mode of the cross-compilers on the BG/P is static linking. This makes sense, since the CNK only executes one process and having shared libraries is not much of a benefit, at least for RAM savings. Dynamic linking is supported however, and we had to use it in order to support the foreign function interface.

Switching to dynamic linking proved to be a major problem, because it turned out that the CNK places dynamically linked executables in a virtual address space that is above 2GB. However, the Linux/PPC SBCL implementation is 32-bit and the most-significant bit in its pointers is used for tagging purposes. As with the signal context related problems, this was almost a show-stopper.

The solution presented itself in the form of a mechanism called *persistent memory*. In some circumstances one can make the CN boot its operating system, execute a job, and then without shutting down, execute another job. The persistent memory mechanism allows one to designate some memory that does not get wiped out between the jobs. The exact semantics of this did not become apparent, because we actually didn't use persistent memory. We used a side-effect: if any memory is labeled persistent, the CNK will place the executable image lower, around 1GB. The mechanism is controlled via environment variables, so an easy workaround was to just require a single block of persistent memory during startup:

```
export BG_PERSISTMEMRESET=1
export BG_PERSISTMEMSIZE=1
```

The `BG_PERSISTMEMRESET` environment variable reinitializes the block every time, so in fact it is not persistent at all. Still, the workaround worked for us: `load-shared-object` and the whole foreign function interface was now operational.

3.10 Building the contribs

The one remaining challenge was to build the contrib packages. No major problems occurred here, but there were a few minor glitches:

- `sb-grovel` built fine, but was not usable, since it works by forking a `gcc` process, compiling a `foo.c` and then running it; its output is `constants.lisp`, which is included in the target build. All this is not possible on the CNK: there is

no `fork` syscall, not to mention the lack of `gcc`. So, at all places where `sb-grovel` was used (which are `sb-posix` and `sb-bsd-streams`), we had to manually perform the procedure on the FEN and create a pre-defined `constants.lisp` file, just like in the win32 case;

- Some of the contribs (`sb-posix` mainly) use `dlopen(NULL)` and then `dlsym` to find some of the functions from the `libc` library. This however turned out to be non-operational on CNK. The easy solution was to add these functions to a list of similar entities in `tools-for-build/ldso-stubs.lisp`. This makes their addresses available beforehand.

By fixing these, we managed to perform a full build of SBCL for the CNK. We proceeded with loading SLIME's `swank` and then connected to it with an emacs/SLIME environment on our desktop, via ssh tunneling to the FEN.

4. Current status of the port

Given the marginal user base of the port and the fact it is still very young, it is hard to assess whether it has production quality. We can report that the following do work:

- The tests included in SBCL's distribution PASS, except those related to the limitations listed below;
- We managed to run SLIME's `swank` on the CNK and to connect to it from an emacs/SLIME environment on a desktop machine;
- Foreign-function bindings to MPI works, which allows the parallel runtimes to communicate with each other in HPC mode;

While these are sufficient to move forward and start thinking about the items listed in the Future Work section, there are known limitations which have to be considered:

- No threads support. Threads only work in gencgc environment and the port uses Cheney GC.
- Only works in SMP mode. The persistent memory hack used to workaround the 2GB virtual address does not work in VN and DUAL mode. And because there is no thread support, this means a lot of wasted resources.
- `sb-grovel` does not work on the CN. It relies on the `fork` syscalls, which is not implemented;
- `asdf-install` does not work on the CN. It relies on the `gethostbyname` syscall, which only honors the hosts file since there is no resolver on the computing node;
- More tests are required in order to come up with a complete list of syscalls that are not implemented or not working properly;
- The SBCL compiler does not know about the second FPU present in each processor core. This leads to performance waste.

5. Future Work

While the port described above enables SBCL to run in a parallel environment with thousands of processors, that is really all it does. In order to make Lisp a great platform for developing scalable, high-quality parallel software, there is still a lot of work that needs to be done in several different areas.

Overcoming the limitations of the port One of the directions for future work is to try and overcome the limitations of the port, most notably make gencgc work and introduce thread support. It is hard to see how we can proceed in this direction without changes in the operating system, so further research is needed in order to come up with a solution.

Speaking of changes, the port also needs mechanism for dealing with them in the future versions of the OS and a test suite to help check whether the assumptions for the syscalls still hold. This requires constant monitoring of the source code of the CNK and making appropriate decisions. Right now it is hard to forsee if, how and when some of the more difficult syscalls problems will be resolved.

One of the limitations – the lack of support for the second FPU – does not depend on operating system changes. This means that work on this item can start immediately. It would require in-depth knowledge of the compiler inner workings and skills in the practice of code optimizations.

Scientific libraries bindings Another obvious direction for future work is the creation of foreign-function bindings to the most popular scientific libraries and wrapping them in constructs that are closer to the Lisp philosophy. Bindings for these libraries will enable the reuse of knowledge and help jump-start Lisp development by giving optimized tools in the hands of the daring. There is no particular wish list or order of preference for this task and its solution will come naturally and gradually over time. It is important to note that such activities are not directly related to the SBCL port for BlueGene/P, but to the Lisp platform in general.

Native libraries An alternative to the above is the creation of counterpart native libraries. A lot of effort must be put forward in order to make the performance of such code optimal. One idea is to implement critical procedures as VOPs entirely in Assembly language and make them part of the compiler. For example, the fastest BLAS library implementation – GotoBLAS [3] – is hand-coded in Assembler and is open sourced. We might reuse this knowledge and make it part of the Lisp environment. Again, this is not related with the port described above and is of a more general nature. One thing that does relate to the port though, is that such native assembly code can make use of the second FPU of the computing node processors, which will additionally boost the performance.

Shared memory One of the problems with the BlueGene/P architecture is that each computing node has access to 2GB of memory at most. In Virtual Node mode this figure is even smaller – 512 MB. This will definitely be a problem for large calculations and one of the possible solutions is to create a virtual shared-memory block that is distributed over the computing nodes. This will allow large data collections to be used transparently without worrying about memory constraints and details regarding where they are stored and how to access them. Such solution already exists for C/C++/FORTRAN – the Global Arrays Toolkit [2]. Of course we can bind to this library and use it, but it would be even better if similar mechanism could be implemented for native Lisp collection data types (lists, hash tables, arrays). Even though the BlueGene/P port will certainly benefit from such solution, it would also apply to other parallel environments as well.

Parallel interactive programming Interactivity is one of the distinguished features of Lisp and it adds a lot to its dynamic nature. Unfortunately, it is not quite clear how it fits in a parallel environment. When we have thousands of virtual machines running, how are we supposed to interact with any and all of them? Or for that matter, with different subsets of them? Where do we put the REPL, the debugger console, and if we decide where to put it, how are the other machines going to interact with it? How are we to keep the dynamic nature of the language in such a world? Because, if we are to give up the interactivity and the dynamic nature, we might as well be better off just using C. These are some rather difficult questions and they call for new ingenious approaches to interactivity in parallel environments. Can we achieve it without changing the fundamentals of the language? Future research will show.

One simple idea that comes to mind is to place the REPL on a certain master node, say MPI rank 0. Then, by using a reader macro

we can instruct the reader to forward the next form to a certain slave node (or set of nodes). Such forwarding can be made on top of MPI. The following crude example illustrates this idea:

```
(defvar *mpi-rank* nil)
(defvar *mpi-size* nil)
(defvar *mpi-streams* nil)

(defun swamp ()
  (mpi-init)
  (setf *mpi-rank* (mpi-comm-rank))
  (setf *mpi-size* (mpi-comm-size))
  (cond
    ((/= 0 *mpi-rank*)
     ;; slaves
     (setf *standard-input*
       (make-instance 'mpi-input-stream
                      :rank 0 :tag 99))
     (swank::simple-repl))

    (t
     ;; master
     (setf *mpi-streams* (make-array *mpi-size*))
     (dotimes (i *mpi-size*)
       (setf (aref *mpi-streams* i)
             (make-instance 'mpi-output-stream
                            :rank i :tag 99)))
     (set-dispatch-macro-character
      #\# #\>
      #'(lambda (stream sub-char rank)
          (declare (ignore sub-char))
          (let ((out (aref *mpi-streams* rank)))
            (print (read stream) out)
            (finish-output out))))
     (swank:create-server)))
  (mpi-finalize))
```

This example is based on `swank` and uses a couple of classes, MPI character input/output streams, that can be easily built on top of Gray streams (`sb-gray`) by subclassing relevant base classes. The actual I/O can be performed by using `MPI_Recv` and `MPI_Send`.

The slave nodes redirect their standard input to an MPI input stream and run `swank`'s REPL. Each of the slaves' input streams are connected with corresponding output streams, created by the master and stored in an array indexed by slave node rank. The master also installs a reader macro which redirects the next form to a certain node, and creates a `swank` server to which an external SLIME can connect.

The macro interprets its last argument as the target slave rank, finds its stream, reads the following form and puts it on the stream. On the other end of the stream the target slave's REPL reads and evaluates the form. Thus, the following input:

```
CL-USER> #123> (defun fact (i)
                 (if (= i 1)
                     1
                     (* i (fact (1- i)))))
```

will lead to sending the `defun` form for evaluation to node 123.

This example is rudimentary and a real solution would also need to take care of multiple nodes specification, debugger redirection, and many more details. One of the problems with this approach is that the form gets parsed twice – by the REPL at the master, and then by the REPL at the slave. Another approach would be to introduce a new form, `in-node`, which somehow switches the node that will perform the REPL.

All in all, the area of parallel interactive programming seems like a major topic for future research. Again, this is not directly related to the SBCL port for IBM BlueGene/P, but is of a more fundamental nature.

Revisiting *Lisp, CM-Lisp and others Efficient high-level parallel programming languages based on Lisp were conceived in the past. Some of the notable examples are CM-Lisp, *LISP and Paralation Lisp (see the Related Work section). A lot of effort was put in those works and it would be wise to reuse the knowledge. It is worth exploring the idea to revisit these languages and see if they can be made fit to work in distributed-memory MPI-based environment, including clusters and modern supercomputers.

6. Related Work

The IBM BlueGene/P is mainly a distributed-memory parallel machine. A processor does not have direct access to the memory of other processors and interprocess communication is mainly achieved via MPI. A quite different approach to parallelism is the shared-memory model, in which all parallel instances have access to the same memory. There are a lot of Lisp implementations which support the shared-memory model by utilizing multi-thread support, including SBCL itself. Scieneer Common Lisp [6] is another example of an excellent implementation for high-performance computing that supports multi-threading on a variety of different platforms.

The SBCL port for IBM BlueGene/P is definitely not the first supercomputing Lisp environment ever. Such environments existed in the past, the defining example being the Connection Machine [10], invented in the 1980s. This is a massively parallel supercomputer whose computing nodes are organized in a cubic structure not very different from the way the BlueGene is organized. It has thousands of simple processors, each having a small amount of local memory and operating in SIMD fashion. The BlueGene/P however has a more general organization that also allows MIMD mode of operation to be implemented.

A SIMD machine calls for data-parallel languages. Several such languages were designed, the most notable being CM-Lisp, *LISP and Paralation Lisp.

CM-Lisp (Connection Machine Lisp) [18] introduced mechanisms for describing distributed data structures (for example `xapping`, `xector`, etc.) and instructions that operate in parallel over such structures (e.g. `apply-to-all`).

Similar concepts are introduced in *LISP [13]. It is realized as a Common Lisp extension and its fundamental data structure is the `Pvar`, a parallel variable, represented as a vector. Operations on `Pvar`s are executed in parallel over a set of nodes on the Connection Machine. The language introduces primitives for performing basic operations on `Pvars`, as well as means for communication between them. A parallel pretty-printer is also present (`ppp`), which can be useful in deriving the concept for parallel interactive programming described above.

Paralation Lisp is crafted after the model described in [16] and is embedded in Common Lisp. Again, it introduces a parallel data structure (`field`) and means to specify locality, communication and parallel operations on fields.

7. Conclusion

Lisp is known to possess many features beyond the static C/C++ and FORTRAN family of languages, such as:

- interactive work during design-time, debug-time and run-time that greatly reduces application creation efforts;
- built-in exact and large-number arithmetic;

- automatic garbage collection;
- ability to construct high-level syntactic and control abstractions through unparalleled macro facilities;
- condition handling system that provides much wider options for error handling than exception handling in static languages;
- fully dynamic environment in which code, types and data can be replaced during execution, and in which different HPC nodes can work on quite different tasks;
- ability to control the language environment itself on all levels, including parsing, evaluating, printing, effectively allowing the user to create problem-specific languages;

All these (and many more) are quite applicable for the field of parallel computing and will enable new programming techniques, paradigms and algorithms to be created, based on bottom-up, interactive and high-level abstractions programming model. Having a Lisp environment enables authors to come up with entirely new ways of organizing the computing process. We think that implementing highly sophisticated systems that smartly use the computing resources would be easier to implement in Lisp than in C, C++ or FORTRAN.

The SBCL build for IBM Blue Gene/P described in this paper brings all these Lisp features into the present-day peta-scale supercomputing domain. Hopefully, this will help make Lisp the language of choice for creating highly-scalable, self-adapting parallel applications for the science and industry.

The author maintains an open source project at SourceForge (`http://sourceforge.net/projects/bgp-sbcl/`), which includes the patches that need to be applied to the SBCL source tree in order to make it compile for the Blue Gene/P CNK.

Acknowledgments

The author would like to thank the Bulgarian National Centre for Supercomputing Applications (NCSA), for providing access to the IBM Blue Gene/P system operated in Sofia, Bulgaria.

References

[1] Bg/p open source project wiki. URL `http://wiki.bg.anl-external.org/index.php/Main_Page`.

[2] Global arrays toolkit home page. URL `http://www.emsl.pnl.gov/docs/global/`.

[3] Gotoblas home page. URL `http://www.tacc.utexas.edu/tacc-projects/gotoblas2/`.

[4] Mpich2 home page. URL `http://www.mcs.anl.gov/research/projects/mpich2/`.

[5] Sbcl home page. URL `http://www.sbcl.org/`.

[6] Scieneer common lisp home page. URL `http://www.scieneer.com/scl/`.

[7] Sbcl user manual. URL `http://www.sbcl.org/manual/`.

[8] Sbcl internals wiki. URL `http://sbcl-internals.cliki.net/index`.

[9] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13:677–678, November 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362790.362798. URL `http://doi.acm.org/10.1145/362790.362798`.

[10] W. D. Hillis. *The Connection Machine*. PhD thesis, MIT Dept. of Electrical Engineering, 1988.

[11] G. Lakner. *IBM System Blue Gene Solution: Blue Gene/P System Administration*. An IBM Redbooks publication, 2009. URL `http://www.redbooks.ibm.com/abstracts/sg247417.html`.

[12] G. Lakner and C. P. Sosa. *Evolution of the IBM System Blue Gene Solution*. An IBM Redpapers publication, 2008. URL `http://www.redbooks.ibm.com/abstracts/redp4247.html`.

[13] C. Lasser and S. Omohundro. The essential *lisp manual. Technical report, Thinking Machines Corporation, 1986.

[14] V. Pavlov. Efficient quantum computing simulation. In *JVA'06: IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing*, pages 235—239, 2006.

[15] C. Rhodes. Sbcl: A sanely-bootstrappable common lisp. In R. Hirschfeld and K. Rose, editors, *Self-Sustaining Systems*, pages 74–86. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-89274-8. doi: http://dx.doi.org/10.1007/978-3-540-89275-5_5. URL `http://dx.doi.org/10.1007/978-3-540-89275-5_5`.

[16] G. W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, 1989.

[17] C. Sosa and B. Knudson. *IBM System Blue Gene Solution: Blue Gene/P Application Development*. An IBM Redbooks publication, 2009. URL `http://www.redbooks.ibm.com/abstracts/sg247287.html`.

[18] G. L. Steele, Jr. and W. D. Hillis. Connection machine lisp: fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 279–297, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: http://doi.acm.org/10.1145/319838.319870. URL `http://doi.acm.org/10.1145/319838.319870`.

[19] P. R. Wilson. Uniprocessor garbage collection techniques. In *IWMM '92 Proceedings of the International Workshop on Memory Management*, pages 1—42. Springer-Verlag, London, 1992.

[20] R. Zelazny. *Nine Princes in Amber*. Doubleday, 1970.

A Futures Library and Parallelism Abstractions for a Functional Subset of Lisp

David L. Rager
The University of Texas at Austin
1616 Guadalupe Street
Suite 2.408
Austin, TX 78701
ragerdl@cs.utexas.edu

Warren A. Hunt, Jr.
The University of Texas at Austin
1616 Guadalupe Street
Suite 2.408
Austin, TX 78701
hunt@cs.utexas.edu

Matt Kaufmann
The University of Texas at Austin
1616 Guadalupe Street
Suite 2.408
Austin, TX 78701
kaufmann@cs.utexas.edu

ABSTRACT

This paper discusses Lisp primitives and abstractions developed to support the parallel execution of a functional subset of Lisp, specifically ACL2.

We (1) introduce our Lisp primitives (futures) (2) present our abstractions built on top of these primitives (`spec-mv-let` and `plet+`), and (3) provide performance results.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*

General Terms

Performance, Verification

Keywords

functional language, parallel, plet+, spec-mv-let, granularity, Lisp, ACL2

1. INTRODUCTION

Our project is about supporting parallel evaluation for applicative Common Lisp, specifically the purely functional programming language provided by the ACL2 system [11, 9]. We provide language primitives, three at a low level and two at a more abstract level, for convenient annotation of source code to enable parallel execution.

Our intended application for this parallel evaluation capability is the ACL2 theorem prover, which has been used in some of the largest industrial formal verification efforts [2, 14]. As multi-core CPUs become commonplace, ACL2 users would like to take advantage of the underlying available hardware resources [10]. Since the ACL2 theorem prover is primarily written in its own functional language, it is reasonable to introduce parallelism into ACL2's proof process in a way that takes advantage of the functional programming paradigm.

After discussing some related work, we introduce three Lisp primitives that enable and control parallel evaluation, based on a notion of *futures*. We build on these three primitives to introduce two primitives at a higher level of abstraction.[1] We then demonstrate these primitives' utility by presenting some performance results. We conclude with remarks that include challenges for the Lisp community.

2. RELATED WORK

There is a large body of research in parallelizing functional languages and their applications, including work in automated reasoning. Here, we simply mention some of the pioneers and describe some recent developments in the area.

An early parallel implementation of Lisp was Multilisp, created in the early 1980s as an extended version of Scheme [6]. It implemented the *future* operator, which is often defined as a promise for a form's evaluation result [7, 3]. Other parallel implementations of Lisp include variants such as Parallel Lisp [7], a Queue-based Multi-processing Lisp [4], and projects described in Yuen's book "Parallel Lisp Systems" [16]. Our approach builds upon these approaches by implementing parallelism primitives and abstractions for systems that are compliant with ANSI Common Lisp, and thus, available for use in applications like ACL2. Furthermore, our abstractions have clear logical definitions inside a theorem proving system, making it straightforward to reason about their use.

More recent developments include the Bordeaux Threads project [1], which seeks to unify the multi-threading interfaces of different Lisps. We approach the same problem by providing a multi-threading interface [12, 13] to Clozure Common Lisp (CCL) and Steel Bank Common Lisp (SBCL). We have our own multi-threading interface because we need some different features. For example, our interface exposes

[1]The higher-level primitives are defined within the ACL2 logic, and hence have clear functional semantics that are amenable to formal verification. We avoid further discussion of the ACL2 logic in this paper.

ELS'2011 TUHH, Hamburg, Germany.
[European Lisp Symposium].

a CCL feature for semaphores, *notification objects*, that we use to determine whether a semaphore was actually signaled (as opposed to returning from a wait due to a timeout or interrupt).

Another recent development is Haverbeke's PCall library [8]. This library is similar to our futures library, in that it provides a way to spawn a thread to evaluate an expression and then use the returned value in the original spawning thread. As a branch of this PCall library, Sedach has initiated the Eager Future2 project, whose web site [15] reports some additional features like error handling and the ability to force the abortion of a future's evaluation. Our latest extension of ACL2, which is described in this paper, has some of these features.

3. FUTURES LIBRARY

There are three Lisp primitives that enable and control parallel evaluation: `future`, `future-read`, and `future-abort`. The `future` macro surrounds a form and returns a data structure with fields including the following: a closure representing the necessary computation, and a slot (initially empty) for the value returned by that computation. This structure can then be passed to `future-read` to access the value field after the closure has been evaluated. The final primitive, `future-abort`, terminates futures whose values are no longer needed.

The following naïve version of the Fibonacci function illustrates the use of `future` and `future-read`.

```
(defun pfib (x)
  (if (< x 33)
      (fib x)
    (let ((a (future (pfib (- x 1))))
          (b (future (pfib (- x 2)))))
      (+ (future-read a)
         (future-read b)))))
```

Even if only a single core is available, we still want to support the delaying of a computation until its result is needed, as illustrated by the `spec-mv-let` primitive discussed in the next section. Of course, in the single-threaded case we need not provide infrastructure for distributing computation to more than one thread. The following discussion of our implementation of futures primitives thus includes optimizations for the single-threaded case.

The multi-threaded implementation of `future` provides the behavior summarized above as follows: when a thread evaluates a call of `future`, it returns a future, F. F contains a closure that is placed on the *work-queue* for evaluation by a *worker* thread. The value returned by that computation may only be obtained by calling the `future-read` macro on F. If a thread tries to read F before the worker thread finishes evaluating the closure, the reading thread will block until the worker thread finishes. However, when the single-threaded implementation is given a future, F, to read, if the future has not previously been read, the closure is evaluated by the reading thread, and the resulting value is saved inside F. The final primitive, `future-abort`, removes a given future, F, from the work-queue; sets an *abort flag* in F; and aborts evaluation (if in progress) of F's closure.

4. ABSTRACTIONS

We build two abstractions on top of the futures primitives. These abstractions avoid the difficult task of introducing futures into the ACL2 programming language and logic. One primitive, `spec-mv-let`, is similar to `mv-let` (ACL2's notion of `multiple-value-bind`). Our design of `spec-mv-let` is guided by the shape of the code where we want to parallelize ACL2's proof process. `Spec-mv-let` calls have the following form.

```
(spec-mv-let
 (v1 ... vn)  ; bind distinct variables
 <spec>       ; evaluate speculatively; return n values
 (mv-let
  (w1 ... wk) ; bind distinct variables
  <eager>     ; evaluate eagerly
  (if <test>  ; ignore <spec> if true
              ;   (does not mention v1 ... vn)
      <abort-form>     ; does not mention v1 ... vn
      <normal-form>))) ; may mention v1 ... vn
```

Evaluation of the above form proceeds as suggested by the comments. First, `<spec>` is executed speculatively (as our implementation of `spec-mv-let` wraps `<spec>` inside a call of `future`). Control then passes immediately to the `mv-let` call, without waiting for the result of evaluating `<spec>`. The variables `(w1 ... wk)` are bound to the result of evaluating `<eager>`, and then `<test>` is evaluated. If the value of `<test>` is true, then the values of `(v1 ... vn)` are not needed, and the evaluation of `<spec>` may be aborted. If the value of `<test>` is false, then the values of `(v1 ... vn)` are needed, and `<normal-form>` blocks until they are available.

The other abstract primitive, intended to be of more general use to the ACL2 programmer, is `plet+`. `Plet+` is similar to `let`, but it has three additional features: (1) it can evaluate its bindings in parallel, (2) it allows the programmer to bind not just single values but also multiple values, and (3) it supports speculative evaluation, only blocking when the bindings' values are actually needed in the body of the form. `Plet+` is an enhanced version of our previous primitive, `plet` [13], and it supports the Lisp declarations for `let` that are allowed by ACL2: `type`, `ignore`, and `ignorable`. An optional *granularity* form (as for `plet`) provides a test for whether the computation is estimated to be of large enough granularity. To date we have restricted our use of `plet+` to small examples, preferring to use `spec-mv-let` in our ACL2 builds and testing. That may change as we further develop `plet+`, for example by reducing the garbage generated when binding multiple values.

5. PERFORMANCE RESULTS

We first present example uses of each primitive with naïve versions of the Fibonacci function, comparing their times for parallel and serial executions. Figure 1 shows the results for these tests. Then, in Subsection 5.4, we compare the performance of the parallelized ACL2 prover to the unmodified (serial) version.

All testing was performed on an eight-core 64-bit Linux machine running 64-bit CCL with the Ephemeral Garbage Collector (EGC) disabled and a 16 gigabyte Garbage Collection

(GC) threshold. See Subsection 5.5 for the reasons behind this decision. All times are reported in seconds, and each speedup factor reported is a ratio of serial execution time to parallel execution time. In each case we report minimum, maximum, and average times for ten consecutive runs of each test, both parallel and serial, in the same environment. The scripts and output from running these tests are available for download at http://www.cs.utexas.edu/users/ragerdl/-els2011/supporting-evidence.tar.gz.

5.1 Testing Futures

Recall the definition of `pfib` in Section 3. In our experiments, calling `(pfib 45)` yielded a speedup factor of 7.62 on an eight-core machine, which is nearly ideal in spite of asymmetrical computations occurring at the end of parallel evaluation.

Thus futures provide an efficient mechanism for parallel evaluation. But they also provide an efficient mechanism for aborting computation. By running the following test, we can see how long it takes to abort computation that has already been added to the work-queue. The following script takes approximately 6 seconds to finish, so it only takes about 60 microseconds to spawn and abort a future. We call function `count-down`, which is designed to consume CPU time. `(Count-down 1000000000)` typically requires about 5 seconds. Since calling `mistake` 100,000 times only requires 6 seconds, we know that we are actually aborting computation.

```
(defun mistake ()
  (future-abort (future (count-down 1000000000))))

(time
 (dotimes (i 100000)
   (mistake)))
```

5.2 Testing Spec-mv-let

We next define a parallel version of the Fibonacci function using the `spec-mv-let` primitive. The support for speculative execution provided by `spec-mv-let` is unnecessary here, since we always need the result of both recursive calls; but our purpose here is to benchmark `spec-mv-let`. The following definition has provided a speedup factor of 7.75 when evaluating `(pfib 45)`.[2]

```
(defun pfib (x)
  (if (< x 33)
      (fib x)
    (spec-mv-let (a)
      (pfib (- x 1))
      (mv-let (b)
        (pfib (- x 2))
        (if nil
            "speculative result is always needed"
          (+ a b))))))
```

5.3 Testing Plet+

The following version of the Fibonacci function, which uses `plet+`, has provided a speedup factor of 7.82 for the evaluation of `(pfib 45)`.

[2] ACL2 users may be surprised to see `mv-let` bind a single variable. However, this definition is perfectly fine in Lisp, outside the ACL2 read-eval-print loop.

Figure 1: Performance of Parallelism Primitives in the Fibonacci Function

Case	Min	Max	Avg	Speedup
Serial	40.06	40.21	40.08	
Futurized	5.15	5.78	5.26	7.62
Spec-mv-let	5.13	5.22	5.17	7.75
Plet+	5.08	5.18	5.12	7.82

Figure 2: Performance of ACL2 Proofs with the EGC Disabled and a High GC Threshold

Proof	Case	Min	Max	Avg	Speedup
Embarrass	serial	36.49	36.53	36.50	
	par	4.58	4.61	4.60	7.93
JVM-2A	serial	229.79	242.40	231.14	
	par	34.42	39.42	35.51	6.51
Measure-2	serial	175.99	179.93	176.53	
	par	47.07	53.71	50.01	3.53
Measure-3	serial	86.63	86.85	86.73	
	par	24.24	25.36	24.90	3.48

```
(defun pfib (x)
  (if (< x 33)
      (fib x)
    (plet+ ((a (pfib (- x 1)))
            (b (pfib (- x 2))))
           (with-vars (a b)
                      (+ a b)))))
```

5.4 ACL2 Proofs

We currently use `spec-mv-let` to parallelize the main part of the ACL2 proof process. We are not interested in speedup for proof attempts that take a small amount of time. However, we have obtained non-trivial speedup for some substantial proofs.

Figure 2 shows the speedup for four proofs. The first proof is a toy proof that we designed to be embarrassingly parallel and test the ideal speedup of our system. The proof named "JVM-2A" is about a JVM model constructed in ACL2. The third and fourth proofs are related to proving the termination of Takeuchi's Tarai function [5]. These proofs are not intended to be representative of all ACL2 proofs. Parallelism does not improve the performance of many ACL2 proofs, and it might even slow down some proofs. Investigating these issues is part of our future work.

5.5 The Effects of GC

We now consider the performance of parallelized ACL2 with different garbage collector configurations. In Figure 3, we report the performance of proof "JVM-2A" with the EGC either enabled or disabled and the GC configured to use either the default threshold or a threshold of 16 gigabytes.

While it is clear that both serial and parallel executions benefit from having the EGC disabled and a high GC threshold, one may wish to make a comparison not presented in the figures. Specifically, one could compare the optimal serial

Figure 3: Performance of Theorem JVM-2A with Varying GC Configurations

EGC & Threshold	Case	Min	Max	Avg	Speedup
on, default	serial	245.52	246.99	246.79	
	par	372.54	482.62	413.42	0.60
on, high	serial	245.38	247.09	246.90	
	par	377.91	524.78	422.20	0.58
off, default	serial	291.57	292.14	291.97	
	par	110.57	117.17	114.77	2.54
off, high	serial	229.79	242.40	231.14	
	par	34.42	39.42	35.51	6.51

configuration that uses the default GC threshold (where the EGC is enabled) to the optimal parallel configuration that uses the default GC threshold (where the EGC is disabled). In this comparison, the serial execution requires an average of 247 seconds, and the parallel execution takes an average of 115 seconds, yielding a speedup factor of 2.15.

Figure 3 shows that for applications running in parallel, it may be beneficial to disable the EGC and use a high GC threshold. Of course, those steps would likely be unnecessary in the presence of parallelized garbage collection.

6. CONCLUSION

We provide parallelism primitives at two levels of abstraction and demonstrate their successful use in speeding up computation. The higher-level library provides abstractions, `spec-mv-let` and `plet+`, which allow significant speedup with little extra annotation in the code. The lower-level library is based on the concept of futures and provides more explicit control of parallel computations. Note that the higher-level primitives fit nicely into the ACL2 applicative programming environment. Indeed, we parallelized the key ACL2 proof process, which is written in the ACL2 programming language, using `spec-mv-let`. Our results to date are promising, obtaining significant reductions in some proof times using this parallelized version.

It is our hope that by bringing the continued development of this library to the attention of the Lisp community: (1) ideas from our library can be reused in other systems, (2) Lisp implementors will be motivated to continue improving multithreading capabilities, for example by parallelizing garbage collection, (3) the Lisp community will continue to think about parallelism standards, and (4) we will gain feedback on ways to improve our implementation and/or design.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0945316. We thank Jared Davis, J Strother Moore, and Nathan Wetzler for helpful discussions.

8. REFERENCES

[1] Bordeaux Threads. *Bordeaux Threads API Documentation*, March 2010. http://trac.common-lisp.net/bordeaux-threads/wiki/ApiDocumentation.

[2] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam K. Srivas and Albert John Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design (FMCAD '96)*, pages 275–293. Springer-Verlag, 1996.

[3] Marc Feeley. *An efficient and general implementation of futures on large scale shared-memory multiprocessors*. PhD thesis, Waltham, MA, USA, 1993. UMI Order No. GAX93-22348.

[4] Richard P. Gabriel and John McCarthy. Queue-based multi-processing lisp. In *Conference on LISP and Functional Programming*, pages 25–44, 1984.

[5] David Greve. Assuming termination. In *ACL2 '09: Proceedings of the eighth international workshop on the ACL2 theorem prover and its applications*, pages 121–129, New York, New York, USA, 2009. ACM.

[6] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a microprocessor. In *Conference on LISP and Functional Programming*, pages 9–17, 1984.

[7] Robert H. Halstead, Jr. New ideas in parallel lisp: Language design, implementation, and programming tools. In *Parallel Lisp: Languages and Systems*, pages 2–57, 1989.

[8] Marijn Haverbeke. Idle cores to the left of me, race conditions to the right. June 2008. http://marijnhaverbeke.nl/pcall/background.html.

[9] Matt Kaufmann, Pete Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

[10] Matt Kaufmann and J Strother Moore. Some key research problems in automated theorem proving for hardware and software verification. *Spanish Royal Academy of Science (RACSAM)*, 98(1):181–195, 2004.

[11] Matt Kaufmann and J Strother Moore. *ACL2 Documentation*. ACL2, March 2010. http://www.cs.utexas.edu/~moore/acl2/current/acl2-doc-index.html.

[12] David L. Rager. Adding parallelism capabilities in ACL2. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 90–94, New York, New York, USA, 2006. ACM.

[13] David L. Rager and Warren A. Hunt, Jr. Implementing a parallelism library for a functional subset of lisp. In *Proceedings of the 2009 International Lisp Conference*, pages 18–30, Sterling, Virginia, USA, 2009. Association of Lisp Users.

[14] David Russinoff, Matt Kaufmann, Eric Smith, and Robert Sumners. Formal verification of floating-point RTL at AMD using the ACL2 theorem prover. In Simonov Nikolai, editor, *Proceedings of the 17th IMACS World Congrress on Scientific Computation, Applied Mathematics and Simulation*, Paris, France, 2005.

[15] Vladimir Sedach. *Eager Future2*, January 2011. http://common-lisp.net/project/eager-future/.

[16] C. K. Yuen. *Parallel Lisp Systems: A Study of Languages and Architectures*. Chapman & Hall, Ltd., London, UK, 1992.

Session II: Performance & Distribution

Implementing Huge Term Automata

Irène A. Durand
idurand@labri.fr
LaBRI, CNRS, Université de Bordeaux, Talence, France

ABSTRACT
We address the concrete problem of implementing bottom-up term automata and in particular huge ones. For automata which have so many transitions that they cannot be stored in a transition table we have introduced the concept of fly-automata in which the transition function is represented by a (Lisp) function.

We present the implementation of fly-automata inside the Autowrite software. We define some fly-automata in the domain of graph model checking and show some experiments with these automata. We compare fly-automata with table automata.

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
Tree automata, Lisp, graphs

1. INTRODUCTION
The Autowrite[1] software entirely written in Common Lisp was first designed to check call-by-need properties of term rewriting systems [6]. For this purpose, we have implemented 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 [7], 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. The following famous theorem [5] connects the problem of verifying graph properties with term automata.

[1] `http://dept-info.labri.fr/~idurand/autowrite/`

ELS'2011 TUHH, Hamburg, Germany.
[European Lisp Symposium].

THEOREM 1.1. *Monadic second-order model checking is* fixed-parameter tractable *for tree-width [2] and clique-width [5].*

Tree-width and *clique-width* are graph complexity measures based on graph decompositions. A *decomposition* produces a term representation of the graph. For a graph property expressed in monadic second order logic (MSO), the *algorithm* verifying the property takes the form of a term automaton which recognizes the terms denoting graphs satisfying the property.

In [4], we have given two methods for finding such an automaton given a graph property. The first one is totally general; it computes the automaton directly from the MSO formula; but it is not practically usable because the intermediate automata that are computed along the construction can be very big even if the final one is not. The second method is very specific: it is a direct construction of the automaton; one must describe the states and the transitions of the automaton. Although the direct construction avoids the bigger intermediate automata, we are still faced with the hugeness of the automata. For instance, one can show that an automaton recognizing graphs which are acyclic has 3^{3^k} states where k is the clique-width (see Section 3) of the graph. Even for $k = 2$, with which not very many interesting graphs can be expressed, it is unlikely that we could store the transition table of such an automaton.

The solution to this last problem is to use *fly-automata*. In a fly-automaton, the transition function is represented, not by a table (that would use too much space), but by a (Lisp) function. No space is then required to store the transition table. In addition, fly-automata are more general than finite bottom-up term automata; they can be infinite in two ways: they can work on an infinite (countable) signature. they can have an infinite (countable) number of states.

This concept was easily translated into Lisp and integrated to Autowrite.

The purpose of this article is

- to present in detail the concept of fly-automaton,
- to explain how automata and especially fly-automata are implemented in Autowrite,
- to present some experiments done with these automata for the verification of properties of graphs of bounded clique-width.

This automata approach for checking graph properties is an alter-

native to the classical algorithms of graph theory. Many interesting properties on arbitrary graphs are NP-Complete (3-colorability, clique problem, etc). For graphs of bounded clique-width, the automaton corresponding to the property is a linear algorithm. One advantage of the use of automata is that, using inverse-homomorphisms, one can get easily algorithms working on induced subgraphs from the ones working on the whole graph which is not feasible with classical algorithms of graph theory.

2. PRELIMINARIES

We recall some basic definitions concerning terms. The formal definitions can be found in the on-line book [1]. We consider a finite signature $\mathcal{F}$ (set of symbols with fixed arity). We denote by $\mathcal{F}_n$ the subset of symbols of $\mathcal{F}$ with arity n. So $\mathcal{F} = \bigcup_n \mathcal{F}_n$. $\mathcal{T}(\mathcal{F})$ denotes the set of (ground) terms built upon a signature $\mathcal{F}$.

EXAMPLE 2.1. *Let $\mathcal{F}$ be a signature containing the symbols $\{a, b, add_{a\_b}, rel_{a\_b}, rel_{b\_a}, \oplus\}$ with*

$$\mathsf{arity}(a) = \mathsf{arity}(b) = 0 \quad \mathsf{arity}(\oplus) = 2$$
$$\mathsf{arity}(add_{a\_b}) = \mathsf{arity}(rel_{a\_b}) = \mathsf{arity}(rel_{b\_a}) = 1$$

We shall see in Section 3 that this signature is suitable for writing terms representing graphs of clique-width at most 2.

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

$$\begin{array}{lcl} t_1 & = & \oplus(a, b) \\ t_2 & = & add_{a\_b}(\oplus(a, \oplus(a, b))) \\ t_3 & = & add_{a\_b}(\oplus(add_{a\_b}(\oplus(a, b)), add_{a\_b}(\oplus(a, b)))) \\ t_4 & = & add_{a\_b}(\oplus(a, rel_{a\_b}(add_{a\_b}(\oplus(a, b))))) \end{array}$$

In Table 1 we see their associated graphs.

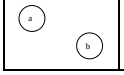
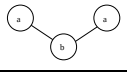
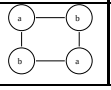
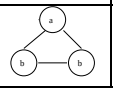
t_1	t_2	t_3	t_4
			

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

3. APPLICATION DOMAIN

All this work will be illustrated through the problem of verifying properties of graphs of bounded clique-width. We present here the connection between graphs and terms and the connection between graph properties and term automata.

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.

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

Properties of a graph G can be expressed by sentences of relevant logical languages. For instance, *G is complete* can be expressed by $\forall x, \forall y, edg_G(x, y)$ or *G is stable* by $\forall x, \forall y, \neg edg_G(x, y)$ Monadic Second order Logic is suitable for expressing many graph properties like k-colorability, acyclicity (no cycle), ...

3.2 Term representation of graphs of bounded clique-width

DEFINITION 1. *Let $\mathcal{L}$ be a finite set of vertex labels 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 $\oplus$[3], the union of disjoint graphs, the unary edge addition add_{a_b} that adds the missing edges between every vertex labeled a to every vertex labeled b, the unary relabeling rel_{a_b} that renames a to b (with $a \neq b$ in both cases). A constant 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 constants.

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.

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

4. TERM AUTOMATA

We recall some basic definitions concerning term automata. Again, much more information can be found in the on-line book [1].

4.1 Finite bottom-up term automata

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[10]. The class of regular term languages is closed under the Boolean operations (union, intersection, complementation) on languages which have their counterpart on automata. For all details on terms, term languages and term automata, the reader should refer to [1]. An example of an automaton is given in Figure 1.

To distinguish these 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 previously defined term automata as *table-automata*.

EXAMPLE 4.1. *Figure 1 shows an example of a table-automaton. It recognizes terms representing graphs of clique-width 2 which are stable (do not contain edges). State* `<a>` *(resp.* `<b>`*) means that we have found at least a vertex labeled* `a` *(resp.* `b`*). State* `<ab>` *means that we have at least a vertex labeled* `a` *and at least a vertex labeled* `b` *but no edge. State* `error` *means that we have found at least an edge so that the graph is not stable. Note that*

[3]`oplus` will be used instead of $\oplus$ inside `Autowrite`.

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

when we are in the state `<ab>`*, an* `add_a_b` *operation creates at least an edge so we reach the* `<error>` *state.*

Run of an automaton on a term

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. If the access time to the transitions is constant, the run takes linear time with regards to the size of the term.

Recognition of a term by an automaton

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

Figure 2 shows in a graphical way the run of the automaton `2-STABLE` on a term representing a graph of clique-width 2. Below we show a successful run of the automaton on a term representing a stable graph.

```
add_a_b(ren_a_b(oplus(a,b))) ->
add_a_b(ren_a_b(oplus(<a>,b))) ->
add_a_b(ren_a_b(oplus(<a>,<b>)) ->
add_a_b(ren_a_b(<ab>)) ->
add_a_b(<b>) -> <b>
```

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 transition function,*

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

where Q *is a countable set of states, disjoint from* $\mathcal{F}$*,*

- fs *is the final state function*

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

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 set of states are finite.

THEOREM 4.1. *Fly-automata are closed under Boolean operations, homomorphisms and inverse-homomorphisms.*

We shall call *basic* 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.

4.3 Relations between fly and table-automata

When a fly-automaton $(\mathcal{F}, \delta, \mathsf{fs})$ is finite, it can be compiled into a table-automaton $(\mathcal{F}, Q, Q_f, \Delta)$. The transition table Δ can be computed from δ starting from the constant transitions and then saturating the table with transitions involving new accessible states until no new state is computed. The set of (accessible) states Q is obtained during the construction of the transitions table. The set of final states Q_f is obtained by removing the non final states (according to the final states function fs) from the set of states.

A table-automaton is a particular case of a fly-automaton. It can be seen as a compiled version of a fly-automaton whose transition function δ is described by the transitions table Δ and whose final state function fs verifies membership to Q_f. It follows that the automata operations defined for fly-automaton will work for table-automata.

Table-automata are faster for recognizing a term but they use space for storing the transitions table. Fly-automata use a much smaller space (the space corresponding to the code of the transition function) but are slower for term recognition because of the calls to the transition function. A table-automaton should be used when the transition table can be computed and a fly-automaton otherwise.

5. IMPLEMENTING TERM AUTOMATA

5.1 Representation of states and sets of states

States for table-automata

For table-automata, the *principle* that each state of an automaton is represented by a single Common Lisp object has been in effect since the beginning of Autowrite. It is then very fast to compare objects: just compare the references. This is achieved using hash-consing techniques.

Often we need to represent *sets* of states of an automaton. For fly-automata, we shall use *lists* of ordered states. Each state has an internal unique number which allows us to order states. Set operations on sorted lists (equality, union, intersection, addition of a state, etc) are faster. For stronger typing, these sorted lists are in fact encapsulated inside *container* objects.

States for fly-automata

For fly-automata however, states are not stored in the representation. For basic fly-automata, they are created on the fly by calls to the transition function. It follows that the previously set out principle is not necessarily applicable.

For composed automata, the states returned by the transition function are constructed from the ones returned from the transition functions of the combined automata.

For operations like determinization and inverse-homomorphisms, sets of states are involved.

If a state is not represented by a unique object, comparisons of states may become very costly when states become more and more complicated. In that case, we shall have no space problem but we may get a time problem.

A solution is, to apply the same principle as for table-automata, that is to say, to represent each state by a unique object. But for this we shall have to maintain a table to store the binding between some description of a state and the unique corresponding state. This table could be reset between runs of the automaton on a term. But it may happen that so many states are created by one single run that we get a space problem. In some cases, a compromise must be found between the two techniques.

```
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: A table-automaton recognizing terms representing stable graphs

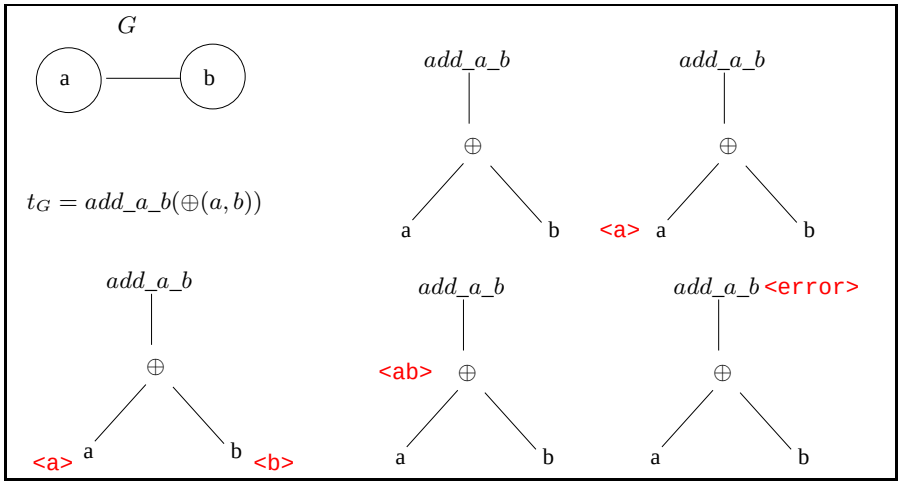


Figure 2: Graphical representation of an (unsuccessful) run of the automaton

5.2 Automata

The implementation of table-automata was partially discussed in [4]. Although the implementation of table-automata benefits from the use of Lisp, it could as well be programmed in any other general purpose programming language. The implementation of fly-automata however is much more interesting because of its use of the functional paradigm to represent and combine transition functions and its use of the object system to deal uniformly with fly or table automata. Other languages like `Dylan`[5] or `Scheme`[6] which provide a similar powerful integration of object-oriented and functional concepts could have been used as well.

The abstract class `abstract-automaton` shown in Figure 3 generalizes the two notions of table-automaton and fly-automaton. An abstract automaton has a signature $\mathcal{F}$ and transitions.

The concrete class `table-transitions` contains the transitions which are repredented by a table.

The concrete class `table-automaton` shown in Figure 3) contains the automata whose transitions are `table-transitions` and whose final states are represented by a set of states.

The concrete class `fly-transitions` contains the transitions which are repredented by a function.

The concrete class `fly-automaton` contains the fly-automata $(\mathcal{F}, \delta, \mathsf{fs})$ whose transitions `fly-transitions` and which have a final state function to decide whether a reached state is final.

```
(defclass abstract-automaton
    (named-object signature-mixin)
  ((transitions :initarg :transitions
                :accessor transitions-of)))

(defclass table-automaton
    (abstract-automaton)
  (finalstates))

(defclass fly-automaton
    (abstract-automaton)
  (finalstates-fun))
```

Figure 3: Automata classes

[5] `http://www.opendylan.org/`
[6] http://www.scheme.com/

5.3 Transitions

The abstract class `abstract-transitions` shown in Figure 4 generalizes the two notions of transitions: table-transitions and fly-transitions.

The transition function applied to a symbol f of arity n and a list of n states $q_1, \ldots, q_n$ returns what we call a `target`. The target can be `NIL` if the transition is undefined, a single state q if the transition is deterministic or a set of states $\{q'_1, \ldots, q'_{n'}\}$ otherwise.

When we have recursively computed the targets $T_1, \ldots, T_p$ for all the arguments $t_1, \ldots, t_n$ of a term $f(t_1, \ldots, t_n)$, we may compute the desired target with

`apply-transition-fun-g` (Figure 4)

which applies the transition function to the elements of the cartesian product of the targets (a target which is a single state q being assimilated with the singleton $\{q\}$).

The operation `compute-target` (Figure 5) implements the run of the transitions on a term $t = f(t_1, \ldots, t_n)$. It computes recursively the targets $T_1, \ldots, T_n$ of the arguments $t_1, \ldots, t_n$ respectively and applies `apply-transition-fun-g` with f and the computed targets $T_1, \ldots, T_n$.

6. IMPLEMENTING AUTOMATA OPERATIONS

The main operations that are implemented on all automata are:

- run of an automaton $\mathcal{A}$ on a term t,
- recognition of a term t by an automaton $\mathcal{A}$,
- decision of emptiness for $\mathcal{A}$ ($\mathcal{L}(\mathcal{A}) = \varnothing$),
- completion, determinization, complementation of an automaton $\mathcal{A}$,
- union, intersection of two (or more) automata,
- homomorphism and inverse homomorphism on an automaton $\mathcal{A}$ induced by a homomorphism (inverse homomorphism) on the constant signature $\mathcal{F}_0$.

For table-automata, we have also implemented

- reduction (removal of inaccessible states),
- minimization.

but this is not discussed in this paper.Some high level operations can be implemented at the level of abstract automata. This is the case for the run of an automaton, the recognition of a term.

For instance, the run of an automaton $\mathcal{A}$ on a term $t = f(t_1, \ldots, t_n)$ is achieved by a call to by the operation `compute-target` on t and $\mathcal{A}$ which returns the target accessible from t using $\mathcal{A}$.

```
(defgeneric compute-target (term automaton)
  (:documentation
   "computes target (NIL, q or {q1,...,qk})
    of TERM with AUTOMATON"))
```

When no state is accessible, the target is `NIL`. Otherwise when the computation is deterministic, the target is a single state otherwise it is a sorted list of states.

A target is *final* if it is not `NIL`, if it is a single final state q or if it contains a final state q.

A term is *recognized* when it reaches a final target.

```
(defgeneric recognized-p (term automaton)
  (:documentation
   "true if TERM recongized by AUTOMATON"))

(defmethod recognized-p
    ((term term) (a abstract-automaton))
  (let ((target (compute-target term a)))
    (values
      (finaltarget-p target a) target)))
```

The decision of emptiness is also done at the level of the class `abstract-automaton` because it involves running an automaton and not creating new ones.

Determinization, complementation, union, intersection, homomorphism and inverse homomorphism can all be implemented for fly-automata. We shall detail some of these constructions further.

Because a table-automaton can always be transformed into a fly-automaton and a finite fly-automaton back to a table automaton we get the corresponding operations for table-automata for free once we have implemented them for fly-automata. It is though possible to deal uniformly with table and fly-automata.

However, for efficiency reasons, it might be interesting to implement some of these operations at the level of `table-automaton`. For instance, the complementation which consists in inverting non final and final states is easily performed directly on a table-automaton.

Implementing operations directly at the level of

`table-automaton`

has the drawback that it depends on the representation chosen for the transitions table. Whenever, we would want to change this representation we would have to re-implement these operations. The only advantage is a gain in efficiency.

Some operations on table-automata may give a blow-up in terms of the size of the transition table (determinization, intersection). In this case, the solution is to avoid compiling the resulting automaton back to a table-automaton.

6.1 Creation of a fly-automaton

To create a basic fly-automaton one should provide a signature, a transition function and a final state function. These three components depend completely on the application domain.

For the automaton `STABLE` of Figure 1, the states are of the class `stable-state`.

The transition function is given by the operation

`stable-transitions-fun`.

The operation

`common-transitions-fun`

triggers the operations

`graph-add-target`,
`graph-oplus-target`,

```
(defclass abstract-transitions () ())

(defgeneric transitions-fun (transitions)
  (:documentation "the transition function to be applied to a symbol of arity
n and a list of n states"))

(defgeneric apply-transition-fun (root states transitions)
  (:documentation "computes the target of ROOT(STATES) with TRANSITIONS"))

(defmethod apply-transition-fun ((root arity-symbol) (states list)
     (transitions abstract-transitions))
  (funcall (transitions-fun transitions) root states))

(defgeneric apply-transition-fun-g (root targets transitions)
  (:documentation "computes the target of ROOT(TARGETS) with TRANSITIONS"))

(defmethod apply-transition-fun-g
    ((root arity-symbol) (targets list) (tr abstract-transitions))
  (do ((newargs (targets-product targets) (cdr newargs))
       (target nil))
      ((null newargs) target)
    (let ((cvalue (apply-transition-fun root (car newargs) tr)))
      (when cvalue
(setf target (target-union cvalue target))))))
```

Figure 4: Transitions

```
(defgeneric compute-target (term transitions)
  (:documentation "computes target of TERM with TRANSITIONS"))

(defmethod compute-target ((term term) (transitions abstract-transitions))
  (let ((targets (mapcar (lambda (arg) (compute-target arg transitions))
                   (arg term))))
    (apply-transition-fun-g (root term) targets transitions)))
```

Figure 5: Run of an automaton

`graph-ren-target`
according to the root symbol of the term
(`add`..., `ren`..., `oplus`).
These operations have a specific implementation when states belong to the class `stable-state`. The code is given in Figure 6.

The function `fly-stable-automaton` shown in Figure 7 returns a fly-automaton which recognizes stable graphs.

```
(defun fly-stable-automaton
    (&optional (cwd 0))
  (make-fly-automaton
   (setup-signature cwd)
   (lambda (root states)
     (let ((*ports* (iota cwd)))
      (stable-transitions-fun root
                              states)))))
```

Figure 7: Fly-automaton for stability

If `cwd>0`, the automaton is finite and works on graphs of clique-width less or equal than `cwd`. If `cwd=0`, the automaton is infinite (by its infinite signature) and works on graphs of arbitrary clique-width.

The call `(fly-stable-automaton 2)` returns a finite fly-automaton whose compiled version is shown in Example 4.1.

The main task for defining a basic fly-automaton is to describe the states and the transition function. Although it is quite simple for the property of stability, it can be quite tricky for some properties like acyclicity for instance. Constructions of automata for many graph properties can be found in [3]. Most of them have been translated into Lisp inside the `Autograph`[7] system which itself uses `Autowrite` for handling table-automata and fly-automata.

6.2 Complementation of a fly-automaton

For a deterministic and complete automaton, the complementation consists just in complementing the final state function. The signature and the transitions remain the same. The corresponding code is shown in Figure 8.

6.3 Determinization of a fly-automaton

If an automaton $\mathcal{A} = (\mathcal{F}, \delta, \mathsf{fs})$ is not deterministic, its transition function returns sorted sets of states $\{q_1, \ldots, q_p\}$.

The determinized version of $\mathcal{A}$ is an automaton $d(\mathcal{A}) = (\mathcal{F}, \delta', \mathsf{fs}')$.

[7] `http://dept-info.labri.fr/~idurand/autograph/`

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

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

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

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

(defmethod stable-transitions-fun ((root constant-symbol) (arg (eql nil)))
  (let ((port (name-to-port (name root))))
    (when (or (not *ports*) (member port *ports*))
      (make-stable-state (make-port-state (list port))))))

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

Figure 6: Fly-transitions for stability

```
(defmethod complement-automaton
    ((f fly-automaton))
  (let ((d (determinize-automaton
            (complete-automaton f))))
    (make-fly-automaton
     (signature f)
     (transitions-fun (transitions-of d))
     (lambda (state)
       (not (finalstate-p state d))))))
```

Figure 8: Complementation

If Q is the domain of δ (the set of states of $\mathcal{A}$), let $d(Q)$ denote the set of states of $d(\mathcal{A})$.

Each subset $\{q_1, \ldots, q_p\}$ of Q yields a state $[q_1, \ldots, q_p]$ in $d(Q)$. δ' is defined by with

$$\begin{array}{rccc} \delta': & \bigcup_n \mathcal{F}_n \times d(Q)^n & \rightarrow & d(Q) \\ & f, S_1, \ldots, S_n & \mapsto & S \end{array}$$

with $q \in S$ if and only if

$\exists q_1, \ldots, q_b \in S_1 \times \ldots S_n$ such that $q \in \delta(f, q_1, \ldots, q_n)$.

And fs' is defined by

$$\begin{array}{rccc} \mathsf{fs}: & d(Q) & \rightarrow & Boolean \\ & S & \mapsto & \exists q \in S \text{ such that } \mathsf{fs}(q) \end{array}$$

The new final state fs' calls the final state function fs of $\mathcal{A}$. It is then obvious to determinize a fly-automaton. This is easily translated into Lisp as shown in Figure 9.

```
(defmethod det-transitions-fun
    ((transitions fly-transitions))
  (lambda (root states)
    (let ((target
   (apply-transition-fun-g
    root
    (mapcar (lambda (state)
                      (states state))
    transitions)))
      (when target (make-gstate target)))))

(defmethod det-finalstates-fun
    ((f fly-automaton))
  (lambda (gstate)
    (some (lambda (state)
            (finalstate-p state f))
  (contents (states gstate)))))

(defmethod determinize-automaton
    ((f fly-automaton))
  (make-fly-automaton
    (signature f)
    (det-transitions-fun (transitions-of f))
    (det-finalstate-fun f)))
```

Figure 9: Determinization

6.4 Other operations

The other operations (completion, union, intersection) are implemented in the same style.

The transition function of union and intersection automata is a function which calls the respective functions of the composed automata.

7. EXPERIMENTS

Most of our experiments have been run in the domain of verifying graph properties as described in Section 3.

7.1 Fly versus table-automata

In order to compare running time of a fly-automaton and of the corresponding table-automaton, we must choose a property and a clique-width for which the automaton is compilable.

This is the case for the *connectedness property*. We have a direct construction of an automaton verifying whether a graph is connected.

The corresponding table automaton has $2^{2^{cwd}-1}+2^{cwd}-2$ states. It is compilable up to $cwd = 3$. For $cwd = 4$, which gives $|Q| = 32782$, we run out of memory.

It is possible to show that the number of states of the minimal automaton is $|Q| > 2^{2^{\lfloor cwd/2 \rfloor}}$. So there is no hope of having a table-automaton for this property and $cwd > 3$.

The P_n[8] graphs have clique-width 3.

We could then compare the computation time with the fly-automaton to the one with the table-automaton, with increasing values of n.

The size of a term representing a graph P_n is $5n + 1$ and its depth is $4n - 3$.

Figure 10 shows that the computation time is roughly linear with respect to n and that the slope of the line is steeper for the fly-automaton.

7.2 Verification of properties

We have built fly-automata (and when possible table-automata) for the following properties. These properties and the corresponding constructions are detailled in [3].

- Edge(X_1, X_2) means that X_1 and X_2 are singleton and that there is an edge between the two vertices.
- Partition$(X_1, \ldots, X_m)$ means that the sets of vertices $X_1, \ldots, X_m$ form a partition of the graph.
- k-Cardinality() means that the graph has excatly k vertices.
- k-Coloring$(C_1, \ldots, C_k)$ means that the partition of vertices $C_1, \ldots, C_k$ forms a coloring of the graph (each C_i is a stable).
- Connectedness() means that the graph is connected (has a single connected component).
- Clique() means that the graph a a clique as an induced subgraph.
- Path(X_1, X_2) means that X_1 contains exactly two vertices and there is a path with vertices in X_2 only connecting these two vertices. This property is useful to express properties on paths.
- Acyclic() means that the graph contains no cycle.
- k-Acyclic-Colorability() means that the graph is k-acyclic-colorable (there exists an acyclic coloring[9] with k colors).
- k-Chord-Free-Cycle() means that the graph does not contain chordless cycle of length k.
- k-Max-Degre() means that the maximum degree of the graph is at most k.

We have direct constructions of the basic fly-automata for the following properties.

1. Polynomial
   - Stable()
   - Edge(X_1, X_2) compilable up to $cwd = 90$
   - Partition$(X_1, \ldots, X_m)$
   - k-Cardinality()
2. Non polynomial
   - k-Coloring$(C_1, \ldots, C_k)$ compilable up to $cwd = 4$ (for $k = 3$)
   - Connectedness() compilable up to $cwd = 3$
   - Clique() compilable up to $cwd = 4$
   - Path(X_1, X_2) compilable up to $cwd = 4$ (for $k = 3$)
   - Acyclic() not compilable

With the previous properties, using homomorphisms and Boolean operations, we obtain automata for

- k-Colorability() compilable up to $k = 3$ ($cwd = 2$), $k = 2$ ($cwd = 3$)
- k-Acyclic-Colorability() not compilable (uses Acyclic)
- k-Chord-Free-Cycle()
- k-Max-Degre()
- Vertex-Cover(X_1) 2^{cwd} states
- k-Vertex-Cover()

The Vertex-Cover[10] property can be expressed by a combination of already defined automata as shown in Figure 11.

Many problems that were unthinkable to solve with table-automata could be solved with fly-automata. For very difficult (NP-complete) problems we still reach time or space limitations.

Figure 12 shows the running time of a fly-automaton verifying 3-colorability on rectangular grids $6 \times N$ (clique-width 8).

8. CONCLUSION AND PERSPECTIVES

We can not think about a better language than Lisp to implement fly-automata whose transition function is represented by a function.

Verifying graph properties on graphs of bounded clique-width is a perfect application field to test our implementation. In the near

[8] A P_n graph is a chain of n vertices

[9] An acyclic coloring is a (proper) vertex coloring in which every 2-chromatic subgraph is acyclic.

[10] A *vertex-cover* of a graph is a set of vertices such taht each edge of the graph is incident to at least one vertex of the set. The problem of finding a minimum vertex cover is a classical optimization problem in computer science and is a typical example of an NP-hard optimization problem that has an approximation algorithm. Its decision version, the vertex cover problem, was one of Karp's 21 NP-complete problems and is therefore a classical NP-complete problem in computational complexity theory.

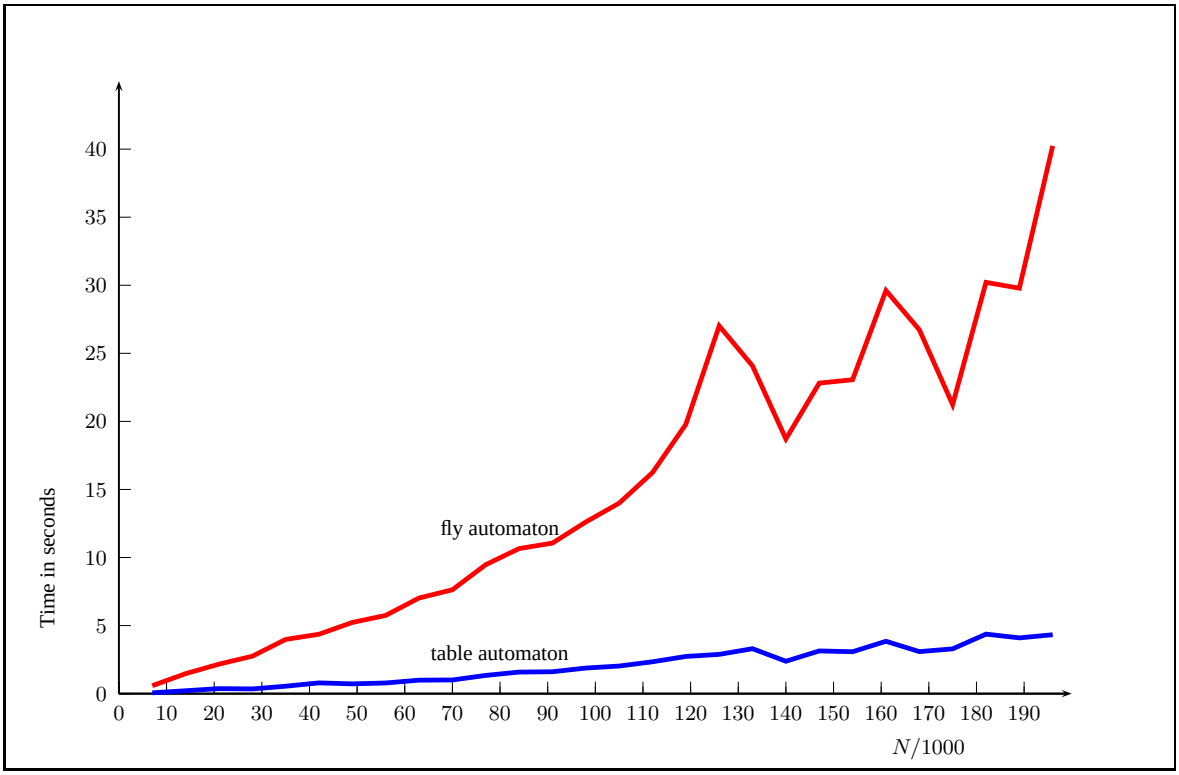


Figure 10: Connectedness on graphs P_N ($cwd = 3$)

```
;; Vertex-Cover(X1) = Stable(V-X1)
(defun fly-vertex-cover (cwd)
  (x1-to-cx1 ; Stable(V-X1)
   (fly-subgraph-stable-automaton
     cwd 1 1))) ; Stable(X1)

;; E. X1 | vertex-cover(X1) & card(X1) = k
(defun fly-k-vertex-cover (k cwd)
  (vprojection
   (intersection-automaton
    ;; Vertex-Cover(X1)
    (fly-vertex-cover cwd)
    ;; Card(X1) = k
    (fly-subgraph-cardinality-automaton
      k cwd 1 1))))
```

Figure 11: Fly-automaton for Vertex-Cover

future, we plan to implement more graph properties and to run tests on real and random graphs.

In this paper, we did not address the problem of finding terms representing a graph, that is, to find a clique-width decomposition of the graph. In some cases, the graph of interest comes with a "natural decomposition" from which the clique decomposition of bounded clique-width is easy to obtain, but for the general case the known algorithms are not practically usable. This problem, known as the *parsing problem*, has been studied so far only from a very theoretical point of view. It was shown to be NP-complete in [8]. [9] gives polynomial approximated solutions to solve this problem. More references can be found in [3].

The concept of fly-automata is very general and could be applied to other domains where big automata are needed. Everywhere where table-automata have already been used, we can hope to solve bigger problems at the condition that basic automata automata involved could be described as basic fly-automata.

Acknowledgements

The author would like to thank the referees for their constructive reports.

9. 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. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85(1):12 – 75, 1990.

[3] B. Courcelle. Graph structure and monadic second-order logic. Available at `http://www.labri.fr/perso/courcell/Book/CourGGBook.pdf` To be published by Cambridge University Press, 2009.

[4] 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.

[5] 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.

[6] I. Durand. Autowrite: A tool for checking properties of term rewriting systems. In *Proceedings of the 13th International*

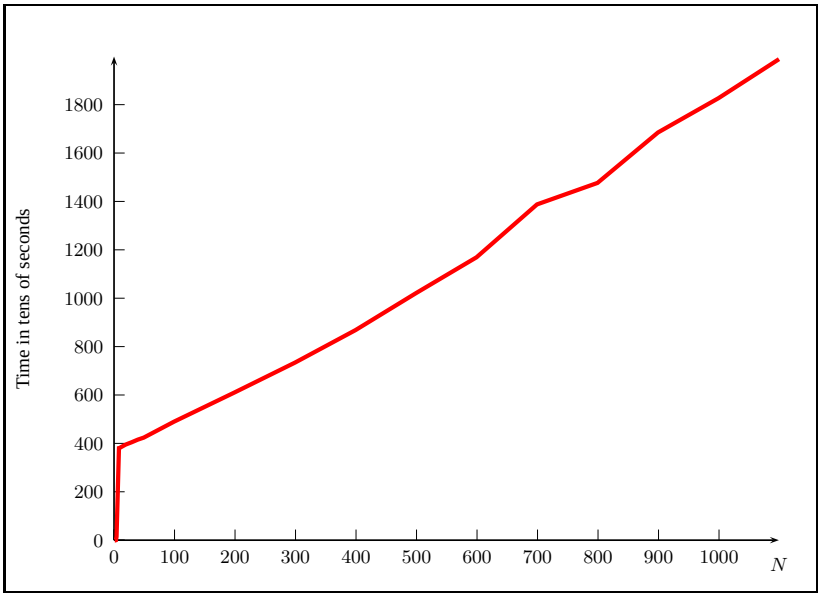


Figure 12: 3-colorability on rectangular grids $6 \times N$

Conference on Rewriting Techniques and Applications, volume 2378 of *Lecture Notes in Computer Science*, pages 371–375, Copenhagen, 2002. Springer-Verlag.

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

[8] 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.

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

[10] 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.

Jobim: an Actors Library for the Clojure Programming Language

Antonio Garrote
Universidad de Salamanca
agarrote@usal.es

María N. Moreno García
Universidad de Salamanca
mmg@usal.es

ABSTRACT

Jobim is a library for building distributed applications in the Clojure Lisp dialect using the actors distributed computing model. To implement this model, Jobim takes advantage of Lisp extensibility features and Clojure coordination primitives. The basic coordination mechanism used by Jobim cluster nodes and built on top of Apache ZooKeeper and supports different communication layers like RabbitMQ AMQP-compliant broker and the ZeroMQ sockets library. Jobim implementation has to deal with limitations imposed by the host Java platform when implementing some of the advanced features available in Erlang's OTP platform, for instance, the creation of a big number of actors being executed in a single Jobim node.The implementation of *evented actors* using the reactor paradigm for non-blocking programming is one of the proposed solutions. The resulting library offers many of Erlang features like process linking, supervisor trees and generic behaviours that can be used to build distributed, fault tolerant applications in plain Clojure.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications

General Terms

Languages, Algorithms

Keywords

Lisp, Clojure, Actors, Erlang, Distributed applications

1. INTRODUCTION

Clojure [1] is a modern dialect of Lisp developed by Rick Hickey with a strong focus on concurrency and including sophisticated programming abstractions like software transactional memory (STM) [8], lock-free asynchronously and synchronously coordinated access to state through Clojure agents and atoms and additional features like futures and promises. As an additional consequence of this emphasis in the support of concurrent programming, Clojure is also a Lisp dialect with a characteristic functional flavour that can be noticed in the use of immutable data types and lazy sequences.

Another major characteristic of Clojure is the hosted nature of the language. Two major implementations of Clojure are currently being developed, one running on top of the Java Virtual Machine (JVM) and the other built on top of Microsoft's Common Language Runtime (CLR).

The presence of a host language offers a mature platform for the deployment of Clojure applications and makes available for the Clojure programmer a big number of development libraries. On the other hand, the host language imposes restrictions and limitations in the current implementation of the language, for instance, the lack of support for tail call optimization in the Java platform makes mandatory the use of the `recur` form to achieve proper tail recursion. It also introduces a backdoor for the use of objects with mutable state in Clojure's purely functional programming model.

One programming area where the hosted nature of Clojure plays an important role is in the application field of distributed computing. Clojure abstractions for concurrency can only be used to coordinate execution threads inside a single node since there is no native support in the language for distributed programming abstractions. The current agreement in the Clojure's developers and users community is that support for distributed computing must be achieved using native features of the hosted language platform like Java RMI and Java Spaces or cross platform solutions like messages queue systems[2].

In this article, Jobim[3], an actors[1] library for Clojure is introduced. Jobim is modeled after Erlang's Open Telecom Platform (OTP)[4] platform trying to offer a set of primitives suitable for writing distributed applications in a functional and reliable way. Jobim has been written as an extension to Clojure that introduces the actor paradigm as a library, in the same vein of other extensions for Lisp dialects like Termite for Scheme [6]. Jobim, as Clojure itself, takes advantage of the hosted platform where the language is executed.

[1] http://clojure.org/

[2] http://groups.google.com/group/clojure/browse_thread/thread/38924bdb1ab63c60/731c5109c59b99af

[3] https://github.com/antoniogarrote/jobim

[4] http://www.erlang.org/

ELS'2011 TUHH, Hamburg, Germany.
[European Lisp Symposium].

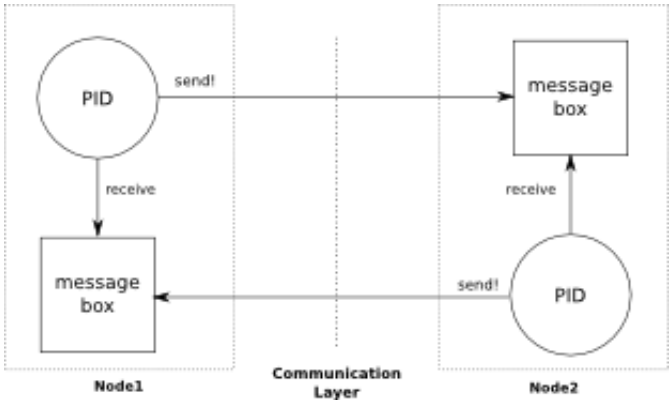


Figure 1: Exchange of messages between actors

It is defined on top of some well known framework and libraries. In particular, Jobim uses Apache's ZooKeeper[5] project to build distributed data structures that can be modified concurrently by actors being executed in different nodes of a Jobim cluster. It also defines an abstraction layer on top of the communication mechanism used by the different nodes in the Jobim application that can be implemented using different communication frameworks. Jobim currently supports two implementations, one defined on top of RabbitMQ[6] AMQP[7] compliant message broker and another one built on top of the ZeroMQ[8] sockets layer.

2. JOBIM ACTORS MODEL

Jobim actors model has been designed after Erlang processes [3]. The core of this model is formed by a single abstraction, the actor, encapsulating a Clojure function, and three forms for creating actors, sending messages to actors and receiving messages.

Each actor being executed in a Jobim cluster has a globally unique PID that can be used by any other actor to send messages to that actor. PIDs are generated in the process of creating a new actor using the `spawn` form. This function returns the PID of the newly created actor. PIDs are just Java `String` objects, and can be sent inside messages exchanged by actors in different cluster nodes, but they identifies unequivocally the node where the agent is being executed. The function `self` can be used by an actor to retrieve the PID associated to itself.

When an actor starts its execution, a message-box for the actor is created and it is associated to the actor PID. An actor can send a message to any other actor providing the destination actor PID as an argument to the function `send!`. When a new message for the actor arrives at the communication layer of a Jobim node, the message is extracted and it is placed in the message-box for that actor, as shown in figure 1. Messages can be retrieved by the actor from the message-box in FIFO order using consequent calls to the `receive` form.

[5]http://hadoop.apache.org/zookeeper/
[6]http://www.rabbitmq.com/
[7]http://www.amqp.org
[8]http://www.zeromq.org/

Sometimes, it is convenient to offer a well known name for an actor in an application. This can be accomplished registering a PID with an associated text handler using the function `register-name`. When an actor registers a PID in a node of the cluster, the name can be immediately resolved to an unique process PID in any other node of the cluster using the form `resolve-name`. Names for PIDs cannot be registered twice. Agents can check the already registered names using the form `registered-names`. When an agent finishes its execution, the name becomes available again for any other agent to register.

Jobim actors are executed inside nodes. Each node is identified by an unique node ID generated when the node bootstraps and a provided node name. Actors can request the execution of a function with certain arguments in a remote node using the `rpc-call` and `rpc-blocking-call` forms. These forms receive the function to be executed, the arguments for the function and the node ID where the function must be executed. A list of available nodes can be obtained with the `nodes` function and node names can be mapped to node IDs using the `resolve-node-name` form. These forms are especially useful when spawning new actors in remote nodes.

Another aspect of Jobim actors taken directly from Erlang processes is the support for the concept of linked actors. Any couple of Jobim actors can be linked using the `link` form. Links are bidirectional, and can be initiated by any party. Once two actors have been linked, any exception interrupting the execution thread of one of the actors will originate a special `signal` message with value `:link-broken` and the failed agent PID to be inserted in the other linked actor message box. This actor can try to handle the failure, restarting the failed process or fail itself with another exception. This new failure will be transmitted to any other linked actor conforming a chain of failed actors. This mechanism for handling distributed failing processes using a hierarchical tree of linked processes is known in Erlang as a supervision tree. Actors in a tree can try to recover from errors in actors placed under that process node. If the agent cannot handle the error, it can fail itself dispatching the error upwards in the supervision tree.

3. ZOOKEEPER AS COORDINATION MECHANISM FOR JOBIM NODES

In order to work properly, certain events in the Jobim cluster need to be notified to all nodes in the cluster, for example a network partition failure, and some nodes must use a coordination mechanism so a distributed application can be executed consistently across the cluster.

Apache's ZooKeeper project, a subproject in Apache's Hadoop MapReduce framework, is used by Jobim to achieve these features. ZooKeeper's basic functionality consists of maintaining a distributed tree of data that clients connected to ZooKeeper can modify atomically. Each of these modifications will be available for other clients in a consistent way. Furthermore, clients can set *watchers* for certain nodes of the distributed tree. Whenever a node in the distributed tree is modified, all the clients that had set up a *watcher* for that node will receive a notification. If a node in the ZooKeeper tree is created as *ephemeral* that datum will only be stored in

ZooKeeper as long as the node is connected and alive. If the node is no longer available because it has been disconnected from the ZooKeeper server or because of a network partition, the *ephemeral* data in the tree will be removed. This basic functionality can be used to build much more generic coordination primitives. Jobim implements two main functionalities on top of ZooKeeper, a distributed register of nodes and named actors, implemented as a distributed *group membership* algorithm [5], and a *2 phase commit protocol* (2PC) [2] for linking processes.

The map of nodes with their identifiers and names is stored in ZooKeeper so each node is notified each time a new node enters or leaves the cluster. This register is implemented as a membership group. New nodes join the group writing their identifiers as ephemeral data in ZooKeeper and the rest of nodes are notified of the event. In the same way, a node can leave the group because of a disconnection or network partition. As a consequence of the removal of the associated ephemeral data in the ZooKeeper tree, the other nodes in the group will be notified of the event. Nodes use these notifications to maintain the current state of their local agents consistent, for example, since the PID of an agent identifies the node where it is being executed, nodes can generate `broken-link` signals for all the local agents linked to some agent being executed in the remote node.

The 2 phase commit protocol implemented on top of Zoo Keeper is another important functionality that is used by Jobim nodes when agreement is required to manipulate the state of the distributed application. One of such examples is the linking of two actors, when one actor starts a link to a remote actor, the linking is only successful if both nodes agree on it. If one node fails in committing in the 2PC protocol or cancels it, a local error for the linking operation is generated.

4. JOBIM PLUGGABLE COMMUNICATION SERVICE

The communication service in a Jobim cluster deals with two main tasks:

- Transforming Jobim node identifiers into network identifiers for the communication mechanism used
- Sending and receiving messages between nodes in the cluster

The communication service in Jobim is defined in a generic fashion using Clojure abstraction features. Two concrete implementations of this abstract service are currently available, one using RabbitMQ AMQP-compliant broker and another one using ZeroMQ sockets layer.

The communication service is defined as a Clojure protocol, a set of abstract operations over a type, named `MessagingService`. This protocol defines two methods, `publish` receiving a message with a destination node to be send, and `set-messages-queue` that receives an object of the `Queue` Java type where the communication layer will store the incoming messages to the node. Finally, a Clojure multimethod named `make-messaging-service` creates a new value of the concrete communication service requested in the arguments of the multimethod.

The RabbitMQ implementation creates a new AMQP exchange in the RabbitMQ message broker for each node, named after the node ID. Additionally it connects a new queue for the node to the AMQP exchange with a well known binding key. When sending a message, the communication layer of a node can generate the name of the exchange for any other node and thus, send the message to the right exchange. RabbitMQ will route the message from the exchange to the only queue connected to that exchange. The communication layer of the target node will receive the messages from the AMQP queue and place them into the Java `Queue` object provided by the Jobim node in the `set-messages-queue` function.

The ZeroMQ implementation retrieves the IP and protocol to establish a connection to the remote ZMQ downstream socket from the node table stored in ZooKeeper. In the receiving node, a Java thread is blocked awaiting messages from the target ZMQ socket and inserting them into the Java `Queue` object passed as an argument in the `set-messages-queue` form.

The communication layer to use can be set up in the configuration of the Jobim node. New types of communication services can be used in Jobim adding a new Clojure implementation for the `MessagingService` protocol and the `make-messaging-service` multimethod.

5. EVENTED ACTORS

Jobim actors created using the `spawn` form create a new Java thread in the Java VM. Each new thread consumes a considerable amount of computational resources in the JVM and can become a problem if a big number of threads are created in a single application. Applications created using the Actors programming model rely on the creation of a big number of actors. As an example, a small Erlang application can create thousand of lightweight Erlang processes and performance testing in Erlang has reported successful execution with as much as 20 million processes [9]. In order to allow the creation of a big number of actors in Jobim, an alternative kind of actors, called *evented actors*, are available. Evented actors in Jobim are based on similar solutions[7] for implementing actor based libraries in JVM based languages like Scala[10].

Multiple evented actors are executed by a single execution thread using the *reactor*[4] design pattern. In the reactor design pattern, different work units share the same execution thread returning the execution control to a *multiplexer* thread, while they await for an interesting event to occur. The multiplexer retrieves the next event happening in the reactor and passes the control to the associated multiplexed thread.

The evented actors API in Jobim offers alternative forms to the `spawn` and `receive` forms named `spawn-evented` and

[9] http://groups.google.com/group/comp.lang.functional/msg/33b7a62afb727a4f?dmode=source

[10] http://www.scala-lang.org/

react. The first one creates a new evented actor while the later interrupts the execution of the actor and returns the control to the multiplexer until an event for this actor arrives. Additional forms like react-loop and react-recur are alternatives to Clojure loop and recur that allows to return the control to the multiplexer thread. The listing 1 shows a very simple actor implemented using a threaded and an equivalent evented actor.

```
(defn ping []
  (loop [[from data] (receive)]
    (do (send! from (str "pong:" data))
        (recur (receive)))))

(defn ping-evented []
  (react-loop []
    (react [[from data]]
      (do (send! from (str "pong:" data))
          (react-recur)))))
```

Listing 1: evented and non evented ping actor

Nevertheless, evented actors are a workaround for a fundamental limitation in the Java platform. Erlang process dispatcher can execute processes in a preemptive manner, assigning a limited number of *reductions* to a lightweight process and then interrupting its execution and starting the execution of a different process. Evented actors cannot be dispatched preemptively by the multiplexer thread and must rely in the cooperation of the evented functions, to allow the execution of multiple evented actors. If a Jobim evented actor includes an infinite loop in its code, it will block the execution of any other evented actor in that multiplexer thread. This is also true for blocking IO actions. To deal with these kind of actions, the form react-future can be used. This form executes the heavy action in a separated Java thread without blocking the reactor thread, and returns the result emitting a special event to the evented actor.

6. GENERIC BEHAVIOURS

The problem of how to implement reusable distributed components on top of the actors programming model has been addressed in Erlang using abstractions known as *behaviours*. Behaviours encapsulate related functionality in a distributed actors system in a similar fashion to classes in an object oriented model.

Jobim implements the idea of behaviours but uses the Clojure concept of *protocol* to define them. A behaviour in Jobim is defined as a Clojure protocol that must be implemented. This mechanism hides all the actor creation, or sending and reception of messages from the implementer that must only write simple functions with well defined semantics.

Currently, behaviors like *supervisor*, *finite state machine*, *event manager* and *generic server* are supported, providing the same interface of their Erlang equivalents.

7. CONCLUSIONS AND FUTURE WORK

Jobim tries to offer an alternative to build distributed applications using pure Clojure code. As Clojure itself, it is built on top of well proved libraries offered by the hosting Java platform like ZooKeeper. It also draws inspiration and the underlying programming model from Erlang's OTP platform.

Nevertheless, Jobim is still at a very early stage of development and many important features offered by Erlang as a platform are missing. One of these features, is the concept of applications as self contained units that can be easily deployed and started in a cluster. Jobim already supports the concept of supervisor tree, but lacks the capacity distributing, starting and stopping this supervisor tree in a cluster-wide fashion. Some promising related work using Java technologies like OSGI has been done, but issues with the current implementation of Clojure's class-loader has prevented any further advances.

Jobim also needs to improve its performance, being one of the current bottlenecks the employ of the Java standard serialization mechanism. Java serialization has important performance issues but makes possible to serialize most of Clojure types. Pluggable support for more efficient serialization mechanisms can boost the performance of Jobim in particular application use-cases.

Finally, Jobim implementation of Erlang's actor model offers a very generic computational model that can be adapted to build other distributed programming models for different application domains. As an example a Petri networks processing library has been built using on top of Jobim[11]. Other libraries for other domains, like Complex Event Processing, could also be easily be implemented.

8. REFERENCES

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] Yousef J. Al-Houmaily and Panos K. Chrysanthis. 1-2pc: the one-two phase atomic commit protocol. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pages 684–691, 2004.

[3] Joe Armstrong. Erlang - a survey of the language and its industrial applications. In *In INAP96*, pages 16–18, 1996.

[4] Edited Jim Coplien, Douglas C. Schmidt, and Douglas C. Schmidt. Reactor - an object behavioral pattern for demultiplexing and dispatching handles for synchronous events, 1995.

[5] Massimo Franceschetti and Jehoshua Bruck. A group membership algorithm with a practical specification. *IEEE Trans. Parallel Distrib. Syst.*, 12:1190–1200, November 2001.

[6] Guillaume Germain. Concurrency oriented programming in termite scheme. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 20–20, New York, NY, USA, 2006. ACM.

[7] Martin Odersky and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 2009.

[8] Ravi Rajwar and James Goodman. Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*, 23:117–125, November 2003.

[11] https://github.com/antoniogarrote/tokengame

Session III: Common Lisp

SICL
Building Blocks for Implementers of Common Lisp

Robert Strandh
Laboratoire Bordelais de Recherche en Informatique
Université de Bordeaux
351, Cours de la Libération
33405 Talence Cedex
France
strandh@labri.fr

ABSTRACT

SICL is a project that aims to supply building blocks for implementers of Common Lisp systems so that they can concentrate on implementation-specific aspects of their systems such as data representation, memory management, and code optimizations, while relying on SICL components for implementation independent "library" code such as functions on sequences, the `format` function, the reader, or standard macros such as `loop`.

While the emphasis of SICL modules is on portability, we are also concerned with other characteristics. In particular, we want SICL modules to be written so as to make it as easy as possible to debug code that uses them. Furthermore we want SICL modules to be competitive with implementation-specific code with respect to performance. Finally, we want the code for SICL modules to be easy to read and understand.

Currently, a single module (most of the *conses* dictionary) has been released, but several other modules are near complete or contain a substantial fraction of the final code, for instance the `format` function, the *sequence* dictionary, iteration macros, and conditional macros.

SICL building blocks are distributed according to a license that is equivalent to "public domain".

1. INTRODUCTION

The main purpose of the SICL project is to provide building blocks (or modules) for implementers of Common Lisp systems. By providing modules that can be implemented portably whenever possible, and by minimizing dependencies on specific implementation details whenever this generality does not harm performance, SICL will make it less daunting to undertake the task of implementing a new Common Lisp system, which we hope will encourage experimentation with new implementation strategies that might improve aspects like performance or debuggability compared to existing systems.

Initially, SICL was meant to be a project with the purpose of realizing a new Common Lisp implementation, mainly in order to improve facilities provided to programmers for debugging their code. However, as mentioned above, implementing a complete Common Lisp system from scratch is indeed a very daunting task.

By changing the focus of the project into providing building blocks for other implementations, we obtain several advantages:

- The code must be written to be independent of a particular implementation. Not only does this make the code useful in other implementations, but in a hypothetical new implementation, it decreases the amount of implementation-specific code, making it easier to modify low-level implementation strategies later.
- It divides the effort into more manageable chunks, with clear intermediate goals.

In the remaining sections of this paper we highlight some of the distinguishing characteristics of SICL. We also give the status of the project and suggest priorities for further work.

2. SPECIAL VERSIONS AND COMPILER MACROS

Many Common Lisp functions take optional and keyword parameters. For such functions, SICL will provide special versions for common combinations of arguments, especially in cases where doing so matters to performance.

For instance, it might be common to supply `eq` as the `:test` for some sequence function. While Lisp implementation might be able to inline such a call and ultimately the call to `eq` as well, we do not assume that the implementation is capable of doing that. An implementation that does not

ELS'2011 TUHH, Hamburg, Germany.
[European Lisp Symposium].

inline such a call might have a performance penalty associated with overhead required by the function-call protocol, especially when the function is as simple as `eq`. Instead of assuming such capabilities of the implementation, we supply special versions of such functions where it is known that `eq` is the test to be used. In the special version, `eq` is called directly (as opposed to via `funcall`) making it more likely that it will be inlined by the compiler.

With special versions for common combinations of arguments, the main function is reduced to doing argument parsing, some error checking and calling one of the specialized versions. By supplying *compiler macros* for calls where the specialized version to be called can be determined at compile time, we can eliminate argument parsing and most error checking as well.

It is interesting to observe that since compiler macros cannot in general inspect the *type* of the arguments (as determined by the declarations and/or type inference) of a call, the specialized version that is ultimately called in place of a sequence function must start by testing the exact type of the sequence. When this can be determined at compile time, this test will likely be elided by the compiler, in later stages. Many Common Lisp implementations provide access to environments that contain type information, that would eliminate this problem, but using such functions would reduce portability, and the potential gain in performance is likely small, at least for the sequence functions.

Providing compiler macros for sequence functions and functions on lists is advantageous only for short sequences. For long sequences, argument parsing represents a vanishingly small part of the total execution time. However, we believe that it is common that sequence functions are used on short sequences, and by providing good performance in those cases as well, we encourage their use in such cases, often with clearer and more concise code as a result.

In some cases, compiler macros might be able to choose a better implementation strategy for the function in question. An example of this situation is the `butlast` function, where an optional parameter determines how many cons cells to return. The default value for this parameter is 1. For that special case, traversing the list with a single pointer, checking the type of the `cdr`, is a valid implementation strategy. In the general case, two pointers are required, each advancing by `cdr`.

3. TRAVERSING A LIST FROM THE END

Certain functions on sequences take a `:from-end` keyword argument indicating that the sequence should be traversed from the end. When the sequence is a list, many different possible solutions have problems:

- One solution is to use recursion and to process the elements during backtracking. This solution is problematic because available stack space might be exhausted for long lists.
- Another solution is to reverse the list destructively, process the elements, and then reverse the result. This solution is unappealing in the presence of threads because other threads may find that the list structure is temporarily destroyed.
- Yet another possibility is to allocate temporary space, say in the form of a vector or a list with the elements in reverse order, to copy the elements to the temporary space, and to process them from there. This solution is unappealing because of the additional memory required, which can be significant if the list is large. It also prevents destructive modifications to the list such as one might want for the function `delete`.
- Certain functions do not *require* the traversal to be from the end of the sequence, for instance `find`, which only requires that the last element that passes the test be returned. A solution that takes advantage of this possibility consists of traversing the list from the start while remembering the last element that passed the test, and returning it at the end. This solution is unappealing because it might apply the `:test` (or `:test-not`) and the `:key` functions many more times than required to determine the result, and when these functions are costly, performance might be lower than expected.

  Other functions, such as `count` *do* require the elements to be processed from the end, so a general solution to this problem must be found.

The solution adopted by SICL is to trade stack space for multiple traversals of the list as follows:

1. The length of the list is first computed by an initial traversal.
2. If the length of the list is below some maximum n allowed, then it is traversed according to the first method above, using recursion and backtracking.
3. If the list is too long, it is divided into chunks of length n. The list is traversed according to the same method, but by advancing by n elements at a time.

This method is a good trade-off in most cases:

- The stack space used can be bounded (at least if an upper bound on the address space is assumed).
- Computing the `nthcdr` of a list is a relatively cheap operation, and could be made cheaper by realizing that most of the uses do not need to test for the end of the list. Though cache performance might be a problem in some cases.
- In practice, more than two traversals are only required for very long lists, or when implementations have serious restrictions on the recursion depth allowed. Therefore, in most cases, our solution is no worse than the alternative solutions suggested above.

4. INTERNATIONALIZATION

While it is probably the case that most Lisp programmers are able to read English, we believe that for some non-native speakers of English who want to learn to program in Lisp the combination of a foreign language and unfamiliar terminology can be a serious obstacle to understanding error messages and documentation strings.

For that reason, we intend to provide the mechanisms required for internationalization of condition reporting and documentation strings (see below). These mechanisms must obviously be transparent, so that implementations using SICL building blocks remain standard conforming. The exact mechanisms to be used have yet to be determined.

5. DOCUMENTATION STRINGS

Conventional wisdom dictates that documentation strings should be physically close to the entity that is documented, and Common Lisp certainly provides the mechanisms to be used in this case.

However, we believe that this proximity is useful only if the two are likely to evolve in parallel as a result of maintenance. For a body of code implementing a standard that is unlikely to evolve in any significant way, such as an implementation of Common Lisp, documentation strings in association with code are merely noise to the code maintainer. We also believe that this proximity discourages the use of complete and explanatory documentation strings, again because significant documentation strings represent noise to the maintainer.

Furthermore, if internationalization is desired (see above), then it might be better to separate the documentation strings from the documented entities, and instead provide different modules providing documentation strings for different languages. Common Lisp provides mechanisms for separating the two as well, through the use of `(setf documentation)`.

For SICL, we intend to provide documentation strings as part of the distribution of each module for the entities in that module. However, we also intend to provide a separate module with documentation strings for all entities in the standard, thus allowing implementations that simply want to improve their documentation strings to do so without otherwise using SICL code.

6. CONDITION REPORTING

Part of the reason for SICL was the desire for better condition reporting, making it easier for developers to debug their programs.

There are (at least) three reasons why some implementations provide suboptimal condition reporting:

- The exact type of the condition signaled is too general to provide an informative message. For instance, if a improper list was given to a sequence function, then a condition of type `type-error` might be signaled. But the condition type `type-error` is too general for it to be possible to indicate that the problem had to do with an improper list.
- An error is often detected not by the standard function that was directly called by the developer's code, but by some other standard function called by it. For instance, a standard function on sequences might rely on `endp` to signal an error when a non-nil atom is given. Some implementations might then indicate that the problem was detected by `endp` which was never explicitly called by the developer's code, or they might not indicate where the problem was detected at all.
- The *condition reporters* do not provide sufficiently explicit messages for the developer to immediately be able to understand what is meant. The reason for this might be similar to the reason why documentation strings are sometimes uninformative, i.e., that they represent noise to the implementer or the maintainer of the Lisp system.

In SICL, we address these issues as follows:

- Since the standard allows for conditions signaled to be *more specific* than the ones that are required, SICL provides a multitude of such conditions that are specific to each situation, allowing better condition reporting, but also allowing an integrated development environment to assist the user with further information.
- An implementation of a standard function in SICL cannot rely entirely on another standard function, or another function used by several different standard functions, to detect and signal an error. Instead, if a standard function calls another function that might signal an error, it must handle that condition, and report a condition specific to it. Other standard functions can be called without any condition handling only if it is known that they cannot detect and signal an error. This restriction allows SICL code to report errors in terms of the standard function that was called directly by client code.
- As with documentation strings, we separate condition reporters from the definitions of the conditions themselves, and provide more explicit messages. These messages can be more explicit thanks to the fact that conditions are specific, and the fact that it is known which standard function was directly called from client code when the problem was detected.

In the second item above, when a standard function F calls G that might signal an error, it might be necessary for F to establish a condition handler, which could be expensive in terms of performance if the total amount of work to be done by F is small. In that case, we use one of two solutions. Either F contains code that tests that the condition cannot be signaled by G, or G takes additional parameters allowing it to signal a condition specific to F.

As with documentation strings, each SICL module will have a set of associated condition reporters, but we also intend to provide a separate module of condition reporters for implementers who simply want to improve the messages emitted when existing standard conditions are signaled.

7. CODE UNDERSTANDABILITY

A secondary objective of SICL code is that it be easy to read and understand. This objective naturally begs the question "to whom?". The target group should of course include maintainers (including potential future maintainers) of the code. We would like to include in this group a little wider audience, so that programmers looking for information about implementation strategies for Common Lisp systems could read and learn from the code.

This goal is sometimes incompatible with the desire for high performance, because high-performance code might be less clear by necessity. We propose to compensate for this by using extensive commenting in an almost "literate" coding style. For application code, we would normally not advocate this style because it may make perfective maintenance harder, but perfective maintenance is less likely on code that implements an existing standard.

Another direct consequence of this objective is that we decided to avoid the use of macros for generating special versions of functions with many keyword arguments, such as the sequence functions. Macros would make the code more maintainable because of avoided code duplications, but at the cost of making the code very hard to understand. Several examples can be found for instance in the code that implements the sequence functions in SBCL. We thus opted to explicitly write out every special version of such functions.

8. CURRENT STATUS

At the moment, the following has been accomplished:

- The first module was released at the end of 2010. It is an implementation of most functionality of the *conses* dictionary of the HyperSpec. Performance is comparable or better than corresponding functions in SBCL (the system we use for our own development), and in some cases defects in the SBCL code do not exist in the SICL equivalent. For instance the standard requires the `butlast` function to signal an error when given a circular list, whereas SBCL goes into an infinite loop.

- A module containing implementations of standard conditional macros (`cond`, `when`, `unless`, `and`, `or`, `case`, etc.) is almost complete. More tests are required.

- A module containing implementations of standard iteration macros (`dolist`, `dotimes`, `do`, `do*`) is almost complete. More tests are required.

- A module implementing `format` is almost complete. Only floating-point printers remain to be implemented. The implementation compiles the control string into elementary operations.

- A module containing an implementation of the *sequences* dictionary is near complete. This was one of the first modules that were being worked on, and some conventions were established later, requiring significant modifications of this module before a release. As with the module for conses, performance is comparable to or better than corresponding functions of SBCL.

- The implementation of the `loop` macro is able to do syntax analysis and some code generation. More extensive semantic analysis and code generation remain to be written.

- A partial implementation of `read`. This implementation provides special versions for common cases such as when the standard readtable is used, and when the input radix is 10. For the special versions, token parsing is done at the same time as token accumulation, making the reader fast. The reader also provides an additional entry point that returns the form read as an *abstract syntax tree* in the form of instances of classes that contain not only the objects returned by the normal entry point, but also information about source location. This additional entry point can be used by a compiler that tracks source code location, and by other tools requiring such functionality.

9. CONCLUSIONS AND FURTHER WORK

The SICL project allows us to revisit many parts of the Common Lisp standard and examine existing implementations with respect to correctness, modularity, performance, code understandability, and debuggability of client code. By doing this in manageable chunks, we obtain more humane milestones the result of which can be used directly to improve existing implementations or to provide "library" code for new implementations.

Existing implementations undoubtedly contain code that could be extracted, adapted to the SICL requirements and become part of SICL, and we do not exclude this method of creating SICL modules. A potential problem might be the license of existing code. As we already mentioned, we believe that SICL code must be distributed with a license equivalent to "public domain" in order to be widely adopted.

Currently few people are actively working on the project, but by dividing the code into independent modules, we facilitate working in parallel and thus make it possible to have additional participants in the future.

In terms of future work, aside from "finish the remaining modules", there are some interesting questions that need further consideration. One such question concerns bootstrapping issues. We currently have not thought through all possible use cases for each SICL module, and it is possible that different use cases have conflicting requirements.

For instance, the sequences module uses the `loop` macro. When used in a new system, the sequences module will likely be compiled on a bootstrap compiler, in which case the loop macro might use sequence functions in expanded code, which clearly will not work. The SICL `loop` macro will only use low-level primitives such as `tagbody`, so when the SICL `loop` macro is used as well, this will not be a problem. But we do not want it to be a prerequisite to use the SICL `loop` macro in order to use the SICL sequence functions. One way of breaking this dependence cycle would be to distribute a macro-expanded version of the sequences module. Similar issues exist with other SICL modules, and a complete analysis of the possible use cases is required, including a list of potential bootstrapping problems.

Another issue that needs further consideration has to do with trade-offs in implementations that might use SICL modules. Existing SICL modules have been written with execution-time in mind, and no attempt has been made to minimize code size. It might be interesting for SICL to provide alternative modules for implementations with different trade-offs. In particular a version that is specifically written with code size in mind might be significantly easier to test than current SICL modules with many special versions of some functions, all of which need to be tested. Furthermore, a "small" version of a module could be used to test a "fast" version of the same module by comparing results for a large number of combinations of inputs for the two versions, as opposed to manually enumerate expected results.

Using Common Lisp in University Course Administration

Nicolas Neuss
Karlsruhe Institute of Technology (KIT)
neuss@kit.edu

ABSTRACT

In this report, I want to sketch how, during the years 2006-2010, I used Common Lisp to reduce administrative tasks significantly for me and my collaborators. I will stress several important steps which may be important for similar projects. Finally, I will hazard an outlook as to how the future may look for this special application.

Categories and Subject Descriptors

J.1 [**Administrative Data Processing**]: Education

General Terms

Management, Legal aspects, Reliability, Security

Keywords

Course Administration, Web Application, Data Protection

1. INTRODUCTION

Common Lisp is a very powerful programming language. Unfortunately, it is not used much in the software industry so that finding a programming job using Common Lisp is usually not easy. Alternatives are to create and sell your own Common Lisp products or to offer Common Lisp consulting. Both can be successful, as several enterprises show, but you would need to be a very good programmer to develop an entire application in Common Lisp from scratch — even if you can leverage a lot of work from external libraries. You probably also need a rather deep knowledge of the application domain or, alternatively, cooperation partners which complicates the whole project.

It is much easier and less risky to use Common Lisp for solving problems which arise in your everyday non-programming job. There are enormous advantages: first, cooperation partners are not essential, because you yourself are an expert in the domain of application. Second, you can expect to profit from the project from the beginning. Third, and most important, you can rely on a steady income independently of the project.

I have been in this fortunate situation for the last 4-5 years. Since August 2006, I have been employed at the University of Karlsruhe as an "Akademischer Rat" which is more or less equivalent to a "lecturer" in the English-speaking world. It is a permanent position, it involves giving lectures and also supervising diploma and doctoral theses. However, compared with a professor, I have additional administrative duties.

In this report, I want to sketch how, during those years, I used Common Lisp to reduce administrative tasks significantly for me and others. I will stress several important steps which may be important for similar projects. Finally, I will hazard an outlook as to how the future may look for this special application.

2. REQUIREMENTS

When I started to work at the University of Karlsruhe in August 2006, my first duty was not giving lectures. Instead, I was responsible for supervising exercises and managing tutorials run by student assistants. The University of Karlsruhe[1] is a technical university which means that we have engineering subjects, and teaching Mathematics to engineering students represents a significant part of the overall tasks of our Math department. This usually means courses with a large number of participants and, correspondingly, the organisation involved several time-consuming and boring tasks (at least at that time). A non-exhaustive list includes the following tasks:

1. Tutors have to be recruited. Seminar rooms for the tutorials as well as rooms and dates for exams have to be organized.
2. Students have to be registered and then divided up into reasonably sized tutorials (20–30 per tutorial).
3. Exercise sheets together with solutions have to be created and distributed.
4. E-mail correspondence and personal discussions with the tutors and students have to be carried out.
5. Exercise scores have to be collected from the tutors.

[1]In 2009, the University of Karlsruhe merged with a DFG Research Center Karlsruhe to form the KIT.

ELS'2011 TUHH, Hamburg, Germany.
[European Lisp Symposium].

6. Exams have to be created and corrected, the results have to be published.

7. The webpage for the course has to be created and kept up to date.

8. The students' scores have to be passed to the university administration, either in the form of certificates on paper (formerly), or online (now).

Items 2,3,4,5,6 were especially time-consuming.

Early in 2007, it looked as if this would become even worse because I faced having to organize the exercises for the summer term for an even larger course with about 400 participants (Numerical Mathematics for Computer Scientists). However, a little relief could be found in a set of PHP scripts which allowed students to register and those correcting the exercises to enter scores in a database, so that it was not necessary to collect them by e-mail.

Nevertheless, I decided at this point *not* to use these existing scripts but to start a Common Lisp application from scratch. There were several reasons:

1. The PHP scripts were not written by professionals, and their quality was doubtful (as far as I could judge).

2. The functionality was partially acceptable (exercise scores could be entered), partially annoying (every week, some students asked for forgotten passwords), and partially missing. It was therefore clear that I would have to develop the scripts further if I wanted use them.

3. The copyright of the scripts was doubtful; probably it lay either with the institute or the university, but nothing was stated in the code and the programmer had already left the university.

Starting from scratch therefore had significant advantages. I could write Common Lisp, I could keep my own standards for the code, and the complete code was under my control as far as is possible.

3. FIRST IMPLEMENTATION

Nevertheless, there were problems. The main problem was that I had little experience with web applications and databases. Furthermore, time was relatively short, because I had only two months before the start of the courses.

At this point, I decided to get professional help and asked Edi Weitz [8], who is a well-known and impressively skilled Common Lisp programmer. He is the author of several important libraries including his own webserver (Hunchentoot) and the quality of his work is well-known throughout the Common Lisp community. Therefore, I asked him to write a prototype web application for me under the BSD license (which would allow me to combine it with non-BSD code of my own) with the sole functionality of letting students register and change their personal data. Edi did indeed manage to program this in a few hours to my complete satisfaction.

This code gave me something to start from and there was still enough time to improve it to a point where it had more functionality than the PHP script. I incorporated several improvements, especially registering with an e-mail confirmation, automatic handling of forgotten passwords (the password was e-mailed back if an e-mail address to be entered agreed with the one in the database), choice of a preferred tutorial, and the management of exercises. I also combined this with GNU Mailman in such a way that every registered student was automatically enroled in a course mailing list.

When I used the application in this form for the first time in the summer term of 2007, it had eliminated or alleviated several of the tasks described above: I did not have to administer the exercise scores any more, I had to write far fewer e-mails than before, and I had also a function which divided students up into tutorial groups. I had no administrative interface at that time, but, since I was the only user and had access to both the Common Lisp level as well as the database level (Postgresql), this was not a real problem.

4. ADAPTATION

After a reasonable basic functionality was working, I could slowly and incrementally improve the application. The most important change was to incorporate an administrator web interface which allowed others to use it as well. Smaller changes included the handling of exams and certificates. For the latter, a secretary role was introduced which could reprint lost certificates—something which I myself had had to do before.

As Common Lisp allows changes to be incorporated often without shutting down the server, my application suffered almost no downtime while steadily improving. Good practice was to test each change rather carefully offline on another computer before synchronizing the source code with the server host, then attaching to and updating the web server. In the rare cases where I anticipated problems during such an update, I usually waited until the weekend or late night to do an update including a complete restart of the web server. It was also an important advantage that I had already installed an e-mail notification service quite early which sent me every error which was caught by Hunchentoot.

Next, I knew that many people set much store by the appearance of a website. Therefore, in January 2008, I asked a cousin [5] who specializes in art to design a logo. I combined it with a style sheet I had from an earlier application so that the whole application looked quite nice.

In 2009, I carried out another rather fundamental change. For some time, I had been unhappy with GNU Mailman. It was certainly very flexible, but it also introduced another level of complexity for users when they wanted to configure one option or another. Since I had all the student's e-mails in my database anyway, I searched for a simpler, independent solution.

I searched the web and found a software which indeed implemented a simple mail transfer agent (smta) in Common Lisp. It was written by Walter C. Pelissero [6], a software

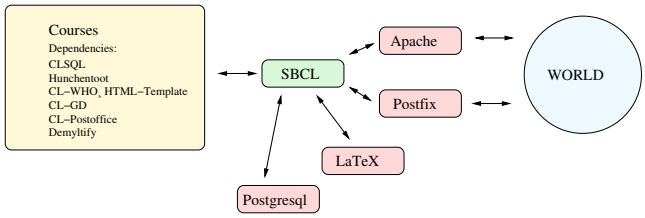


Figure 1: Architecture of the courses system.

consultant living near Frankfurt (Germany), whom I asked per E-Mail if he would consider it a reasonable choice for creating a mail distribution service. He answered that this program was more an exercise than a production quality MTA, and proposed to use his (LGPLed) *milter*[2] software *Demyltify* instead which allows for manipulating mail messages in the Sendmail and Postfix MTAs. I asked him if I could pay him for writing me some code, but he gracefully provided me with a small routine free of charge. Therefore, in the autumn of 2009, I became independent of GNU Mailman, and the whole system looked as shown in Fig. 1.

5. DATA PROTECTION

In March 2008, I sent a mail to my faculty in which I told them about my webservice. They reacted and about a month later, there was a meeting where this issue was discussed. However, the reaction was not very encouraging. Some people had similar web services and did not want to change. Others were afraid because of data protection issues. There were also plans for a campus-wide management system which, in the ideal case, would also have similar functionality. In the end, they decided not to establish another system and to wait for the campus management system which would solve all problems.

This was a small setback, but several positive elements can be seen in that outcome. First, I could devote more time to making the system better. Second, if the faculty was not interested in the software, I was free to do whatever I wanted with it without taking other interests into account. Third, my attention was drawn to the very important issue of data protection.

Even before that meeting I had not been careless about this issue. For example, I had very soon switched to the encrypted https protocol. Therefore, I could always argue that my web service improved on the previous situation where data was often exchanged using unencrypted e-mails. However, if I really wanted to offer a service to others, I had to take another scale into account. The main question was how it was possible to offer such services at all in a legally correct way.

So I started looking around and found (after some detours) that software used at the University of Karlsruhe had to be registered in a certain list which was then synchronized with an institution ("Datenschutzbeauftragter") of the local state government (Baden-Württemberg). To be registered in that list, my software had to be examined by experts from an organisation called ZENDAS [9]. So I contacted ZENDAS and asked for an examination of my software. This turned out to be a really agreeable and stimulating experience, mainly because the people there were very competent. They did not go so far as to examine my source code, but they tested an administrator account I had to make for them. In general, they were really satisfied with my server (saying that they were not used to that quality), and also gave me several hints and orders for improvements (for example, an improved password management). So, after some further detours because of a misunderstanding of mine, my server was taken into this list of "permissible software".

[2]A *Milter* is a mail filter for Sendmail an Postfix.

6. INTEGRATION

After the data protection issue was resolved, I could take a step towards a better integration into the university software infrastructure. One of the main inconveniences was that students had to register with their personal data (name and e-mail) on my server although they already had an university account with all the necessary information. This was an inconvenience which was at the same time an impediment to the further adaption of my service.

Towards the end of 2009, I learned that the KIT (the successor of the University of Karlsruhe) offered a SSO (single-sign-on) service, which means that a "service provider" (in this case my web service) can redirect requests to an "identity provider" (at the KIT computing centre) which performs the authentification and directs the request back again having augmented the necessary data (identification number, name, e-mail address). In 2010, I started to explore this further. First I asked Edi Weitz again, if he could help me there. His answer was that a whole Common Lisp solution would be too much work, but that I might be able to leverage the Shibboleth SSO-module from the Apache webserver. Indeed, this turned out to be the way to go. The people from the KIT computing centre were very competent and helpful, so that this integration turned out to be much less work than I had expected it to be.

7. GROWTH

During 2008 and 2009, several people from my institute used the web service. However, the number of users stayed relatively constant, one reason being that I did not advertise it, since I had to resolve the data protection issues discussed above. However, in the spring of 2010, I received a request from the faculty whether my web service could help them with the registration/distribution of students to seminars. The inclusion of this feature was relatively easy, because it was quite similar to the choice/distribution for tutorials and, since it worked without any problems, it did save both our students and our staff a lot of discomfort.

Now, in the winter term 2010/2011, after the implementation of the SSO authentification mentioned above, two people from other institutes asked for accounts to administer their courses (again, several hundred students). After giving them a short introduction together with a manual, I have not heard any questions or complaints from them, and it looks as if everything works without any problems. For the summer term 2011, two more courses have already asked to be incorporated.

Thus, a steady growth is quite possible as more and more

people learn about the benefits of this service.

8. BENEFITS OF USING LISP

Was using Common Lisp essential for creating this system? Probably not, it was used mainly because I personally know it and like it. On the other hand, there are at least several characteristics of the underlying task which made Lisp highly suitable:

- The specification changed with time, and Common Lisp allowed programming in an incremental style without the need to "throw one version away".
- Some subproblems (like the optimal distribution of students to proseminars or the presentation of simplified interfaces depending on the course) are basically AI problems for which Common Lisp is well suited.
- Having to serve a lot of clients, the system has to be as fail-safe as possible while still allowing change. Here, Common Lisp helps a lot with its elaborate exception handling, and the opportunity of loading patches into the running system.
- The environment is different from place to place so that the perfect system has to adapt to a large range of requirements and tastes. The extreme flexibility of Common Lisp is of help here. For example, I still allow certificate criteria to be specified using a Common Lisp subset.

9. FURTHER DEVELOPMENTS

There are several ways in which the service can be extended. For example, exercises could be extracted from a central database and combined into exercise sheets. Another feature would be to incorporate an easy transfer of student grades to the university database. One could also extend it towards an e-learning application, or towards a more complete campus management solution. However, this is questionable, because well-established solutions already exist for these tasks (see next section).

10. RELATED SOFTWARE

Since there are a lot of universities doing similar tasks, it is not surprising that several related systems exist.

The closest is probably a commercial system called ROSELLA [2], which I learned about last year from a brochure. Then educational software like ILIAS [3] or MOODLE [7] can be configured to do similar things. However, the configuration of these very large systems is far from easy, and they usually need extension modules to handle certain local features. Finally, the functionality partly overlaps with whole campus management systems like [1].

11. THE FUTURE

Will this web service be used at some time throughout the Faculty of Mathematics or maybe even across KIT? Maybe, but this depends very much on the campus management system currently being implemented. A 100-page specification has been written, a public tender performed, and one of the quotations was accepted. Now, I am looking forward to seeing when and in which form it will be delivered, and how well it will handle our much smaller range of problems.

Another interesting question is whether there is any commercial value in the software. The answer is surely yes, because —as mentioned above— there is already the commercial software ROSELLA [2] with a rather similar scope. Therefore, I decided at least to reserve a web domain [4] for my service. However, this is a difficult market, also because of the abundance of choices mentioned in the previous section. A possible niche might be to advertise the minimal work for the staff, and also the handling of special features like the distribution of students to seminars.

12. ACKNOWLEDGEMENTS

I want to thank Bettina Nürnberg, Walter C. Pelissero, and Edi Weitz for their help, and Wolfgang Müller, Martin Sauter, and Bernhard Scheffold for useful discussions and suggestions. Finally, I want to thank the referees for their useful remarks.

13. REFERENCES

[1] CAS Software AG. CAS education. `http://www.cas-education.de`.
[2] NAZUMA GmbH. ROSELLA. `http://rosella.nazuma.de`.
[3] ILIAS. Integriertes Lern-, Informations- und Arbeitskooperations-System. `http://www.ilias.de`.
[4] N. Neuss. Vorlesungsverwaltung. `http://www.vorlesungsverwaltung.de`.
[5] B. Nürnberg. Homepage. `http://www.bettinanuernberg.de`.
[6] W. C. Pelissero. Homepage. `http://wcp.sdf-eu.org`.
[7] MOODLE. Modular Object-Oriented Dynamic Learning Environment. `http://www.moodle.org`.
[8] E. Weitz. Homepage. `http://weitz.de`.
[9] ZENDAS. Zentrale für Datenschutz der Universitäten Baden-Württembergs. `http://www.zendas.de`.

Session IV: Scheme & Racket

Bites of Lists - Mapping and Filtering Sublists

Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
normark@cs.aau.dk

ABSTRACT

The idea of applying map and filter functions on consecutive sublists instead of on individual list elements is discussed and developed in this paper. A non-empty, consecutive sublist is called a bite. Both map and filter functions accept a function parameter - a bite function - which is responsible for returning a prefix bite of a list. We develop families of bite functions via a collection of higher-order bite generators. On top of the bite generators, a number of bite mapping and bite filtering functions are introduced. We illustrate the usefulness of bite mapping and filtering via examples drawn from a functional programming library that processes music, represented as Standard MIDI Files.

Categories and Subject Descriptors

D.1.1 [**Applicative (Functional) Programming**]: Lisp, Scheme; E.1 [**Data structures**]: Lists; H.5.5 [**Sound and Music Computing**]: Systems

1. INTRODUCTION

Mapping and filtering represent some of the classical higher-order functions on lists, together with similar functions such as reduction functions. Both the classical mapping and filtering functions deal with individual elements of a list. A mapping function applies a function to each individual element of the list, and it returns the list of these applications. A filter function selects those elements on which a predicate is fulfilled. The predicate of a filtering function is also applied on each individual element of the list.

The idea described in this paper is to apply mapping and filtering on sublists of a list. The Common Lisp function `maplist`, which applies a given function on successive tails of a list, is a simple example of a mapping function that belongs to this genre. In this context, a sublist of a list L= (e_1 e_2 ... e_n) is a non-empty consecutive part of L, (e_i ... e_j), where $i \leq j$, $i \geq 1$, and $j \leq n$. It is easy to see that for a list L of length n, there are $(n+1)(n/2)$ such sublists.

ELS'2011 TUHH, Hamburg, Germany.
[European Lisp Symposium].

Although it is possible, and maybe even useful, to map and filter all possible sublists of a list, most of the work in this paper will deal with map and filter functions that process mutually disjoint sublists that partition the list. In this context, a disjoint partitioning of a list L = (e_1 e_2 ... e_n) is formed by L_1 = (e_{i_1} ... e_{j_1}), L_2 = (e_{i_2} ... e_{j_2}), ..., L_k = (e_{i_k} ... e_{j_k}) where $k \leq n$, $i_1 = 1$, $j_k = n$, $j_m = i_{m+1}$ for $1 \leq m \leq k-1$. In order to simplify the vocabulary, each non-empty sublist in a disjoint partitioning will be called a *bite*. The bites L_1 ... L_k of a list L = (e_1 e_2 ... e_n) are all non-empty, and when they are appended the result is L.

The development on *bites of lists* has been motivated by our previous work on MIDI music programming in Scheme [9]. A piece of music, represented as a MIDI file, consists of a list of discrete MIDI events. The MIDI list of a typical song consists of thousands of such events. When a list of MIDI events is captured from a MIDI instrument, the first job is typically to *impose structure* on the list. The structure will be a music-related division consisting of units, such as bars/measures, song lines, song verses, or chord sequences. The capturing of music related structures in a list of MIDI events was the starting point of the work described in this paper.

In addition to the music-related application area, we will also mention an example of another area in which mapping and filtering of sublists may be useful. Imagine a long list of time-stamped meteorological data objects that describes the weather conditions during a long period of time. Each object may contain information about temperature, air pressure, rain since last reading, and other similar data. In order to extract characteristics about weather conditions at a more coarse-grained level, it may be relevant to partition the list in sublists. These sublists may, for instance, correspond to regular periods of time (days, weeks, month, or years). Sublists with monotone progressions of the air pressure or temperature could also be of interest. Both kinds of sublists can be produced by the functions described in this paper. Systematic search for (maybe overlapping) sublists with certain more detailed properties is also an area which is supported by functions in this work.

In Section 3 we present the higher-order mapping and filtering functions, each of which rely on a bite function. When applied to a list, a bite function returns a prefix of the list. Prior to the discussion of the mapping and filtering functions, we will in Section 2 introduce a collection of *bite*

function generators. It turns out that these generated "biting functions" are the crucial part of the game. In Section 4 we discuss the bite generators relative to the motivating example of this work (introducing structure in a list of MIDI events). Section 5 contains a description of related work. Section 6 presents our conclusions.

The mapping and filtering of sublists has been developed in the context of the R5RS Scheme programming language [5]. Even though Scheme is dynamically typed, we will often describe the functions by means of statically typed signatures.

2. BITE GENERATORS

A bite function `b` is a function which takes a list of elements as parameter, and returns a non-empty prefix of that list. More precisely, a bite function returns a non-empty list prefix when applied on a non-empty list. Thus, the signature of a bite function `b` is List<E> → List<E> for some element type E. Of convenience, and for generalization purposes, a bite function return the empty list when applied on an empty list.

It is usually straightforward to program a *particular* bite function. In this section we will deal with families of similar bite functions, as generated by higher-order *bite generating functions*.

A particularly simple bite generator is `(bite-of-length n)`, which returns a function that takes a bite of length `n`:

```
((bite-of-length 3) '(a b c d e)) => (a b c)
```

If a function, returned by `(bite-of-length n)` is applied on a list with fewer than `n` elements it just returns that list.

Another simple bite generator `bite-while-element` is controlled by an element predicate passed as parameter to the function. When applied on a list, the generated bite function returns the longest prefix of the list whose elements, individually, satisfy the element predicate. Thus, for instance,

```
((bite-while-element even?) '(2 6 7 4)) => (2 6)
```

Elements which violate the element predicate are called *sentinels*. The requirement that successive bites of a list append-accumulate to the original list makes it necessary, one way or another, to include the sentinel elements in the list. Each bite includes at most one sentinel. Sentinels can either start a bite, terminate a bite, or occur alone as singleton bites. These variations are controlled by an optional keyword parameter[1], `sentinel`, of the bite generator. The default value of `sentinel` is `"last"`. However, the example given above assumes that `sentinel` is `"first"`.

[1]Keyword parameters are simulated in a way that corresponds to how LAML [8] handles XML attributes in Scheme functions. Following the conventions of LAML, an attribute name is a symbol and the attribute value must belong to another type (typically a string). In the context of the function `bite-while-element`, this explains why the sentinel role is a string (and not a symbol).

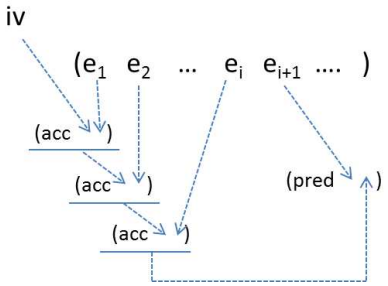


Figure 1: *In the list (e_1 ... e_i e_{i+1} ...) the elements e_1, ..., e_i have been accumulated with use of the function* `acc` *and the initial value* `iv`. *The current element e_{i+1} is passed to the predicate* `pred` *together with the accumulated value.*

The following examples illustrate the role of the `sentinel` parameter:

```
((bite-while-element even? 'sentinel "first")
 '(1 2 6 7 4)) => (1 2 6)

((bite-while-element even? 'sentinel "alone")
 '(1 2 6 7 4)) => (1)

((bite-while-element even? 'sentinel "last")
 '(1 2 6 7 4))  => (1)

((bite-while-element even? 'sentinel "last")
 '(2 6 7 4)) => (2 6 7)
```

The remaining bite generators construct bites based on properties that do not (alone) pertain to individual elements. Functions generated by the expression

```
(bite-while-element-with-accumulation
    pred accumulator init-val)
```

accumulate the elements of the bite. The accumulated value and the 'the current list element' are handed to a predicate `pred`, which controls the extent of the bite. For the sake of the accumulation, a binary accumulator `acc` is needed together with an initial 'getting started value' `iv`. This is illustrated in Figure 1. The generated function returns the longest bite in which each element, together with the accumulation of the "previous elements", fulfill the predicate.

Here follows an example that accumulates elements in an integer list by simple `+` accumulation (using 0 as the initial value). In the example, the predicate states that we are interested in the longest prefix of the list that, successively, has a sum of 5 or below.

```
((bite-while-element-with-accumulation
    (lambda (e s)  (<= (+ e s) 5))
    (lambda (s e) (+ s e))
    0)
 '(1 2 1 1 -1 2 3 4))  => (1 2 1 1 -1)
```

As above, we assume in the general case that the element type of the list is E. The predicate (which is the first parameter shown above) has the signature E × S → bool. The accumulator (the second parameter) has the signature S × E → S. In the predicate, the S-valued parameter is the accumulation of all values before the E-valued 'current element'.

A function generated by `bite-while-element-with-accumulation` always 'consumes' the first element in the list without passing it to the predicate. As a consequence, the first element in the bite does not necessarily fulfill the predicate. Without this special case, bite functions generated by `bite-while-element-with-accumulation`, will be able to return empty bites. Recall from Section 1 that empty bites are illegal (unless taken from an empty list). Successive "biting" with bite a function generated by `bite-while-element-with-accumulation` is illustrated in Section 3.

A function generated by `(bite-while-compare er)` returns the longest bite where elements, pair-wise, fulfill a binary element relation defined by the function `er`. Here is an example where we identify the longest increasing prefix of a list:

```
((bite-while-compare <=)
 '(2 6 6 7 1 2))  => (2 6 6 7)
```

It comes out naturally that bite functions, generated by use of `bite-while-compare`, return non-empty bites when applied on non-empty input. It should be noticed that the effect of `bite-while-compare` can be achieved by a (rather clumsy) application of `bite-while-element-with-accumulation` with an accumulator that just returns the previous elements.

The last bite generation function that we will discuss in this section is `bite-while-monotone`, which can be seen as a convenient generalization of `bite-while-compare`. Based on an element comparator, which follows the C conventions of comparison functions[2], a function generated by `bite-while-monotone` returns the longest monotone bite of the list. More precise, the function returns the longest list prefix where successive pairs of elements have the same value when passed to the comparator function. Here is an example:

```
((bite-while-monotone (make-comparator < >))
 '(1 2 3 2 1 0 4 4 4 1 2 1))
  => (1 2 3)
```

The element comparator is constructed by `make-comparator`, which as input receives the *greater than* and the *less than* functions. Five "successive bitings" with the function in the example produces the bites `(1 2 3)`, `(2 1 0)`, `(4 4 4)`, `(1 2)`, and `(1)` respectively. Such "successive biting" can easily be realized with use of `map-bites` which will be introduced in the following section.

As explained above, all generated bite functions have the signature List<E> → List<E> for some element type E.

[2]In the scope of this paper `(compare-to x y)` is -1 if `x` is less than `y`, 0 if `x` is equal to `y`, and 1 if `x` is greater than `y`.

In some situations it is useful to know where a given bite belongs relative to neighboring bites. For this reason, all generated bite functions accept a second integer parameter that informs the bite function about the *current bite number*, in contexts where bites are generated successively (as introduced in Section 1). In Scheme, this is handled by requiring that all generated bite functions have a rest parameter, like in `(lambda (lst . rest) ...)`, where the first element in `rest` will be bound to the current bite number. Examples are provided when we discuss the bite mapping and bite filtering functions in Section 3.

Finally, most bite generators accept an optional predicate, called a *noise predicate*. Elements that fulfill the noise predicate are passed unconditionally to the resulting bite. Noise elements are not counted (in the context of `bite-of-length`), are not taken into consideration by the predicate of `bite-while-element`, and are not accumulated by `bite-while-element-with-accumulation`. In Section 4 we will see practical examples that reveal the usefulness of noise predicates.

3. BITE MAPPING AND BITE FILTERING

We will now discuss a number of higher-order functions that successively applies a bite function to a list, and which processes the resulting bites in various ways. As already mentioned, some bite functions can be generated by one of the functions described in Section 2. More specialized bite functions will have to be explicitly programmed relative to the specific needs.

3.1 The map-bites function

The function `map-bites` is the natural counterpart to the classic `map` function in both Scheme and Common Lisp. `map-bites` applies a *bite transformation function* on each bite of a list (relative to repeated application of a given bite function):

```
(map-bites bite-function bite-transf lst)
```

If applied on a bite of type List<E>, the transformation function `bite-transf` is supposed to return another list of type List<F>. The lists produced by the bite transformation function are *spliced* (`append` accumulated) by `map-bites`. In that way `(map-bites bf id lst)`, where `id` is the identity function, is equal to `lst` for any bite function `bf`.

Let us first use the Scheme function `list` as the bite transformation function, with the purpose of explicitly identifying the bites successively delivered by a given biting function. This particular transformation illustrate the typical need of somehow revealing/representing the individual bites in the results returned by the mapping or filtering functions.

```
(map-bites (bite-while-element number?) list
  '(1 -1 1 a 3 4 b 6 1 2 3))  =>
((1 -1 1 a) (3 4 b) (6 1 2 3))
```

The bites taken by `(bite-while-element number?)` places a sentinel value as the last element of the bite (because, as explained in Section 2, the default value of the optional

sentinel parameter is "last"). In the example, a sentinel element is an element which is not a number. If the intention is to get rid of sentinel elements after applying the bite function, it may be better to isolate them using the "alone" sentinel option:

```
(map-bites
  (bite-while-element number? 'sentinel "alone")
   list
   '(1 -1 1 a b c 3 4 b 6 1 2 3))  =>
((1 -1 1) (a) (b) (c) (3 4) (b) (6 1 2 3))
```

In this example it is straightforward to get rid of singular, non-numeric elements from this list by ordinary (element) filtering.

Let us also illustrate map-bites relative to other bite functions, as produced by the generators discussed in Section 2.

```
(define sum-at-most-5
  (bite-while-element-with-accumulation
   (lambda (e v)  (<= (+ e v) 5))
   (lambda (v e) (+ v e))
   0))

(map-bites sum-at-most-5 list
   '(1 2 1 1 -1 7 -1 3 1 4))   =>
((1 2 1 1 -1) (7) (-1 3 1) (4))

(define increasing
   (bite-while-compare <=))

(map-bites increasing list
   '(2 6 6 7 5 1 -3 1 8 9))  =>
((2 6 6 7) (5) (1) (-3 1 8 9))

(define monotone-ints
   (bite-while-monotone
     (make-comparator <= >=)))

(map-bites monotone-ints list
   '(2 6 6 7 5 1 -3 1 8 9))   =>
((2 6 6 7) (5 1 -3) (1 8 9))
```

3.2 The bite filtering functions

Bite filtering, as provided by the function filter-bites, has the following parameter profile:

```
(filter-bites bite-function bite-predicate lst)
```

After generation of each bite with use of bite-function the bite is passed to a bite-predicate, which decides if the bite should be part of the output list. The bites accepted by the bite predicate are spliced together, in the same way as in map-bites. The non-accepted bites are discarded. The following example, which works on bites of length 3 (whenever possible), filters the bites that start with a number:

```
(filter-bites
   (bite-of-length 3)
   (lambda (bite) (number? (car bite)))
   '(1 -1 1 a b c 3 4 b 6 1 2 3))  =>
(1 -1 1 3 4 b 6 1 2 3)
```

The following bites are taken out of the sample list: (1 -1 1), (a b c), (3 4 b), (6 1 2), and (3). The predicate only discards (a b c), because the first element of this bite is not a number. The remaining bites are spliced together and returned by filter-bites.

It is often convenient to apply a bite transformation function bite-transf just after filtering with bite-predicate:

```
(filter-map-bites bite-function bite-predicate
                  bite-transf lst)
```

This function first chunks the list lst with use of bite-function. Each resulting bite is passed to bite-predicate, and the accepted bites are transformed by bite-transf (to a value which must be a list). The lists returned by the bite transformations are finally spliced together.

In simple cases we can (just as illustrated for map-bites above) use the bite transformation function list for identification of the bites that have survived the filtering.

```
(filter-map-bites
  (bite-of-length 3)
  (lambda (bite) (number? (car bite)))
  list
  '(1 -1 1 a b c 3 4 b 6 1 2 3))  =>
((1 -1 1) (3 4 b) (6 1 2) (3))
```

The result of this filtering is similar to the previous example, but the bite structure is preserved in the output.

The implementations of map-bites and filter-bites are simple and straightforward. The function map-bites is implemented as a tail recursive function that collects the transformed bites in a list, which is reversed and append-accumulated as the very last step. The function filter-bites is implemented in a similar way.

In general, the bite functions are implemented by means of tail recursive functions which collect the bite elements in a parameter, which is reversed before a bite is returned. Because a bite is prefix of a list, say of length n, we need to allocate n new cons-cells for it. (This would not have been necessary if we worked on suffixes of a list, like the Common Lisp function maplist). It is crucial for our approach that the bites are materialized as separate lists, but it is also quite expensive to allocate (and deallocate) memory for these intermediate structures.

3.3 The step-and-map-bites function

We will now discuss step-and-map-bites which is a slightly more complex variant of filter-map-bites. The parameter lists of step-and-map-bites and filter-map-bites are

basically the same. `filter-bites` and `filter-map-bites` both discard a whole bite b, if b is not accepted by the bite predicate. Let us assume that the length of a bite b is n. It b is accepted by the bite predicate in `step-and-map-bites` it is transformed just like it is done by `filter-map-bites`, and we step n elements forward from the beginning of the current bite before we take the next bite of the list. If `b` is not accepted by `bite-predicate`, we do not discard n elements when using `step-and-map-bites`. Instead, the predicate gives back a step length s (typically 1), and the next bite taken into consideration starts s elements ahead. Elements stepped over are not discarded (as in filtering functions). Such elements are passed directly and untransformed to the output.

In the setup just described, the value returned by the bite predicate has two purposes: (1) The accepting purpose (a boolean view of the value) and (2) the stepping length purpose (an integer view of the value). Therefore the bite predicate of `step-and-map-bites` returns an integer. A positive integer p is considered as an accepting value, and p is the stepping length. The number p is typically (but not necessarily) the length of the most recent bite. A negative integer n is a non-accepting value, and -n is the stepping length. In most practical cases n is -1.

The execution of the functions `map-bites`, `filter-bites`, and `filter-map-bites` are linear in the length of the input list, if the bite functions, together with the other function parameters, are linear in their processing of the prefixes of the list. In contrast, `step-and-map-bites` is able to regress the "biting process", hereby processing the same elements of the input list several times.

As another difference, `map-bites` and `filter-bites` process disjoint "bite" partitionings of a list, as described in Section 1. In contrast, and as discussed above, `step-and-map-bites` processes selected disjoint[3] bites that may be separated by list elements, which are unaffected by the mapping process. We may consider these "in between elements" as intermediate bites. Using this interpretation, the processing done by `step-and-map-bites` can be thought of as operating on disjoint bites (those selected together with the intermediate bites) that append-accumulate to the original list.

In the following example, the biting function takes bites of length 3 out of a list of integers. A bite is accepted if the sum of the elements of the bite is even. If the bite is not accepted, we take a single step forward, and recurses from there.

```
(step-and-map-bites
  (bite-of-length 3)
  (lambda (bite) (if (even? (apply + bite))
                     (length bite)
                      -1))
  list
  '(0 1 2 1 2 3 4 0 -2 1 3 4 5))   =>
(0 (1 2 1) 2 3 (4 0 -2) (1 3 4) 5)
```

The first bite of length 3 is `(0 1 2)`, and its element sum is odd. The stepping mechanism causes `step-and-map-bites` to output the element 0, and consider the next bite `(1 2 1)`. The element sum of `(1 2 1)` is even, and it is accepted and transformed (with the function `list`). We therefore step 3 elements forward. The following (overlapping) bites `(2 3 4)` and `(3 4 0)` are not accepted, because their element sums are non-even. The elements `2` and `3` are consequently transferred the output list. The next bite of length 3, which is `(4 0 -2)`, is accepted, etc.

If the "predicate" of `step-and-map-bites` (the second parameter) returns a positive integer smaller than the length of the bite, overlapping bites will be processed. The following variant of the expression shown above returns all possible consecutive triples of a list with even element sum:

```
(filter list?
  (step-and-map-bites
    (bite-of-length 3)
    (lambda (bite) (if (even? (apply + bite))
                       1    ; <-- The difference
                      -1))
    list
    '(0 1 2 1 2 3 4 0 -2 1 3 4 5)))   =>
((1 2 1) (1 2 3) (4 0 -2) (-2 1 3)
 (1 3 4) (3 4 5))
```

At the outer level of the expression we disregard all non-list elements with use of the ordinary element `filter` function.

4. MAPPING AND FILTERING BITES OF MIDI SEQUENCES

As already mentioned in Section 1, the development on *bites of lists* was motivated by our work on MIDI programming in Scheme. In this section we discuss a number of MIDI programming problems and solutions facilitated by bite mapping and filtering. First, however, we give some background on our approach to MIDI programming in Scheme.

MIDI is a protocol for exchange of music-related events. The MIDI protocol emphasizes sequences of discrete music events (such as `NoteOn` and `NoteOff` events) in contrast to an audio representation of the music. A piece of music can be represented as a *Standard MIDI file* in a compact binary representation. A Standard MIDI file[4] is basically a long sequence of MIDI events. We have developed a Scheme representation of a Standard MIDI file, which relies on LAML[5] for representation of MIDI sequences. We call it MIDI LAML [9].

In MIDI LAML we work on long lists of MIDI events (typically several thousands events). The main goal of our system

[3]Disjointness is only assured if the stepping length of the integer-valued bite predicate returns a positive number which is not less than the length of the accepted bite. The last part of Section 3.3 shows an example where the stepping length is 1. This leads to processing of overlapping bites.

[4]A Standard MIDI file of format 0 is a sequence of MIDI events. Format 1 and format 2 midi files are structured in tracks and songs respectively.

[5]LAML [8] represents our approach and suite of tools to deal with XML documents in Scheme.

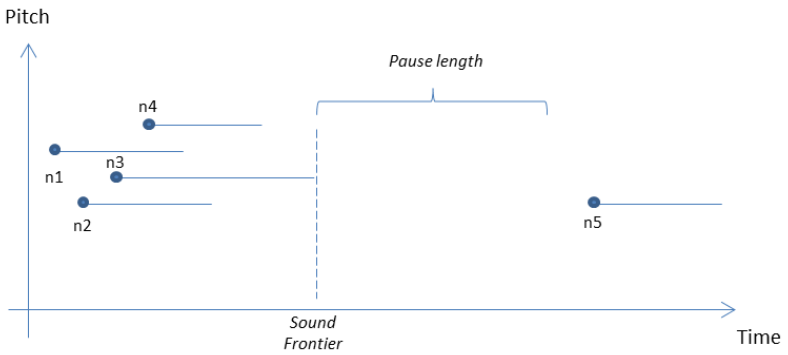


Figure 2: *A piano roll presentation of a few notes n1 .. n5 and their (horizontal) durations. The bite formed by the notes n1 .. n4 is followed by a pause, because n5 starts after the absolute time* Sound Frontier + Pause Length.

is to do useful work on music via functionally *programmed solutions* - in contrast to *interactive operations* in a MIDI sequencer environment. Instead of always working on individual MIDI events it is often attractive to work on selected, consecutive sublists of MIDI sequences. Such sublists are bites of MIDI events.

When a list of MIDI events is captured from a MIDI instrument, the first job is typically to *impose structure* on the list. The structure will consist of music-related divisions of the MIDI list. Many such structures can be captured by application of bite mapping. We will now describe how it can be done by use of the functions from Section 2 and 3. It is recommended that the reader consults the detailed program listings in the appendix while reading the subsections below.

4.1 Bars

For temporally strict music[6] the bar/measure structure can be captured by mapping a *next-bar bite function* over the music using `map-bites`. A relevant bite function can be generated by means of either `bite-while-element-with-accumulation` (for delta timed MIDI sequences) or `bite-while-element` (for absolutely timed MIDI sequences). A simple transformation of `map-bites` is to insert a bar division meta message into the stream of notes in order to emphasize the bar structure of the music. The function `map-bars`, as outlined in Appendix A.1, wraps these pieces together.

As a more interesting application, it is possible via the transformation function of `map-bites` to affect the characteristics of selected bars, for instance the tempo, the velocity (playing strength), or the left/right panning. In the last part of Appendix A.1 we show how to gradually slide the tempo of every fourth bar of a song with use of `map-bars`, which is programmed on top of `map-bites`.

4.2 Pauses

A song is often composed by parts separated by pauses. A *pause* is a period of time *pl* during which no `NoteOn` messages appear. In addition, we will require that all previously activated notes have ended before entering the pause of length *pl*.

It is obviously useful to identify pauses in a song, because it will provide a natural top-level structure in many kinds of music. We provide a function called `map-paused-sections` which maps some function `f` on sections of MIDI message that are separated by pauses. This function is implemented in terms of `map-bite`, which in turn uses a bite function generated by `bite-while-element-with-accumulation`. Please consult Appendix A.2 where the implementation of `map-paused-sections` is presented.

The use of the generated bite function is illustrated in Figure 2. Let us assume that we look for pauses of at least *pl* time ticks. The accumulator keeps track of a point in time called the *sound frontier sf* where all previous notes have been ended. The predicate examines if the next note starts at a time after *sf* + *pl*. If this is the case a pause has been identified, and this ends the current bite of the MIDI sequence.

The actual implementation of `map-paused-sections`, as it is shown in Appendix A.2, uses a variant of `map-bites` called `map-n-bites`, which passes the bite-number to the transformation function. With use of this variation, it is easy to insert numbered markers into the MIDI sequence.

4.3 Sustain intervals

When playing a piano, one of the pedals is used to hold the notes after they have been released on the keyboard. This is called *sustain*. In a MIDI representation, sustain is dealt with by particular `ControlChange` messages that represent the level of the pedal. We are interested in identifying the monotone sustain regions, such as R1, ..., R7 shown Figure 3. A single bite of the MIDI sequence can be identified with a function generated by `bite-while-monotone` from Section 2. When such a bite function is mapped over the entire sequence of MIDI messages with `map-bites`, we can conveniently process the regions shown in Figure 3.

We have programmed a function on top of these applica-

[6]Music played by metronome, or music captured from a source which quantizes the start and duration of notes to common note lengths, is here called *temporally strict music*.

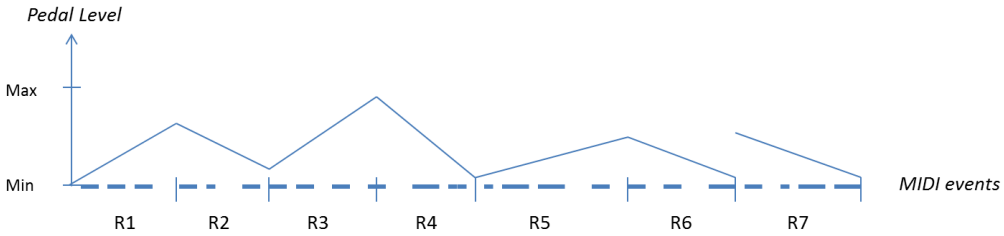


Figure 3: *Seven regions of MIDI events that represent 'pedal down" and 'pedal up' intervals. Region 6 and 7 are both pedal down intervals. The pedal moved quickly from a low value to a high value. It is not a requirement that monotone bites alternate between being increasing and decreasing.*

tions called `map-sustain-intervals`, which passes knowledge about the monotonicity to the transformation function of `map-bites`. (Please consult Appendix A.3 for details). This can, for instance, be used for fastening the downward pedal movements (in the regions R2, R4, R6, and R7 of Figure 3), such that sustained notes rings out steeper or earlier than in the original. Without use of a generated bite function, and without use of `map-bites`, it would be a fairly difficult task to program such a transformation (let alone the effort of realizing the change interactively in a conventional MIDI sequencer).

4.4 Chord identification

As the final example from the domain of MIDI music we will discuss how to identify the chords in a piece music. Chord recognition [12] is much more difficult to handle than the other music related examples we have discussed above. In the context of the work described in this paper, it is not the ambition to come up with a high-quality chord recognition algorithm. Rather, our goal is to find out which chords can be identified based on rather straightforward use of the functions described in Section 2 and 3.

A chord is, in a formulation due to Wikipedia, a set of three or more notes that is heard as if sounding simultaneously [17]. The notes in a chord are not necessarily initiated at the exact same time. At top-level, chords are identified by a function `map-chords`, which is shown in Appendix A.4. This is a higher-order function which applies a given function on every sublist identified as a chord. Internally, `map-chords` calls `step-and-map-bites` which is one of the functions we described in Section 3. We use `step-and-map-bites` together with a bite function generated with `bite-while--element-with-accumulation`. This bite function takes a bite of notes, where each note is temporally relatively close to the previous note. The predicate of `step-and-map-bites` (the second parameter) asserts if the relevant notes of the bite fulfill a chord formula. If it is not the case, the stepping mechanism of `step-and-map-bites` is activated. This implies that a new bite of temporally close notes is taken, and so on. Please take a look in Appendix A.4 for additional details.

4.5 Noise elements

In all but the first example in this section, it is useful to zoom in on certain MIDI events, and to be able to disregard all other events. This is possible by use of the so-called *noise predicate* mentioned briefly in Section 2. The noise predicate can be applied on elements of the list, from which bites are taken. Elements that satisfy the noise predicate are disregarded while identifying a bite (in predicates, comparison, accumulation, etc.), but noise elements appear in the resulting bite.

In the function that locates pauses, `map-paused-sections`, which is shown in Appendix A.2, all non-`NoteOn` events (such as instrument selection, sustain and tempo changes) are considered as noise. In addition, `map-paused-sections` relies on a *relevance function* (third parameter) which at a detailed level points out the MIDI events which should be taken into account when looking for pauses. This may, for instance, be all notes in a particular channel above a certain note value (pitch value). The negation of the *relevance function* is used as noise function of the bite function "under the hood". This separation of concern turns out to be very useful: The filtering of relevant messages takes place in the generation of the bite function (`bite-while-element-with-accumulation`), totally separated from the logic that deals with the rules for pauses in the music.

In the function that identifies sustain intervals, all non-sustain messages (such as `NoteOn` messages) are noise elements. In the chord identification function only `NoteOn` events in a given channel are relevant. All other messages are considered as noise.

4.6 Discussion

As mentioned in the introduction to Section 4, the primary use of bite functions in a music related context is to *identify structures* in the music. Some structures, such as the use of *tracks*, are already manifest in the representation of some Standard MIDI files. The bar structuring can also appear explicitly in the MIDI LAML representation of Standard MIDI files.

Some structures are very difficult to identify automatically. Although we may attempt to capture song lines and song verses via use of bite functions, it is our experience that it is not always realistic to accomplish this with success. Therefore, the MIDI LAML environment contains a number of facilities for manual introduction of additional structures. This includes manual insertions of MIDI markers (meta events), and systematic transformation of events in

a dedicated channel to MIDI markers. See the paper about MIDI programming in Scheme [9] for additional details.

The programming technique explored in this paper relies, to large extent, on higher-order function that receives and generates functions. As a typical scenario, a bite mapping function receives both a bite function *bf* and a bite transformation function *btf*. *bf* may be generated by one of the bite generator functions from Section 2 on the basis of several other functions, such as a list element predicate, an accumulator, and a noise function.

It turns out to be a typical situation that the information established by one function, such as the bite function *bf*, also is needed by another function, such as the bite transformation function *btf*. As a concrete example, the direction of the pedal in Section 4.3 and Appendix A.3 is identified in the bite function, but it is also needed in bite transformation function. It would clutter everything if we attempted to pass this "additional information" as the function result together with the "main function result", for instance with use of multiple valued functions. In pure functional programming, passing the information via an output parameter is not feasible either. We choose to re-calculate the information in the bite transformation function - as the least evil way out of the problem. It is possible to abstract the duplicated parts to a common lambda expressions at an outer scope level, but this solution does not necessarily lower the complexity of the program. As an alternative, it could be tempting to let the bite function fuse the needed information into the bite, hereby transferring it to the bite transformation function. This solution requires that it is possible to associate extra information with the bites.

5. RELATED WORK

In this section we will discuss existing work which is related to our work on bites of lists.

The idea of capturing recursive patterns using "Functions with Functions as Arguments" first appeared in John McCarthy's seminal paper about recursive functions and symbolic expressions from 1960 [7]. In this paper, the `maplist` function (as mentioned in Section 1) appears together with a linear search function. It soon became clear that a large class of list-related problems can be solved by a few applications of `map` and `filter`, typically followed by some reduction. During fifty years, the use of mapping and filtering functions together with reduction functions have played a role in almost any textbook about Lisp, Scheme, and other functional languages.

Common Lisp supports functions on *sequences* [15]. A Common Lisp sequence is a generalization of lists and one-dimensional arrays. Many of the Common Lisp sequence functions are higher-order functions. Some functional arguments are passed as required parameters, others are passed as keyword parameters. The processing of successive bites, as proposed in this paper, can be handled by use of the `:start` and `:end` keyword parameters in many of the sequence functions. The `:start` and `:end` keyword parameters delimit a sublist which subsequently can be processed in various ways (removed, substituted). By use of these keyword parameters a Common Lisp programmer can do simple bite processing. The Common Lisp sequence functions operate at the level of list elements. Only element testing and element transformation is provided for. In contrast, the bite mapping and filtering functions in this paper work on sublists as such. The higher level of abstraction in the bite-related functions may be convenient and powerful in some contexts, but it is also quite expensive. The cost comes primarily from copying prefixes of a list, in order to form the bites.

There exists a variant of Common Lisp sequences called *series* (see appendix A of [15]). In this work the main focus is on automatic transformation of series to efficient iterative looping constructs [16]. In our current work it would be interesting and useful to consider similar techniques for obtaining more efficient mapping and filtering of bites. A number of functions in the series package are oriented towards splitting of a list into one or more sublists (`split`, `split-if`, `subseries`, `chunk`). As such, it is plausible that some of the bite processing programs discussed in this paper can be converted to use functions from the series library.

R5RS Scheme [5] is quite minimalistic with respect to list supporting functions. The repertoire of list functions in the R6RS Scheme standard libraries [14] is more comprehensive. In relation to R5RS, additional list functions are supported by SRFIs, most notable the SRFI 1 List Library [13]. This library supports the `drop` and `take` functions. The expression `(take lst i)` returns the first `i` elements of `lst`, and it corresponds to the `((bite-of-length i) lst)`. The SRFI 1 function `take-while` correspond to the function `bite-while-element`, as described in Section 2.

Modern object-oriented programming languages handle a variety of different collection types via so-called *iterators*. An iterator is a mutable object that manages the traversal of a collection. The LINQ framework (well described in [1]) of C# [2] is a good example of a system which handles the processing of data collections via use of iterators. There exists a number of LINQ query operators that are related to sub-collections (`Take`, `TakeWhile`, `Skip`, `SkipWhile`, `GroupBy`). It remains to be seen to which degree the existing query operators can be used directly for the purposes that are discussed in this paper. If this is not the case, it should be noticed that it is easy to define new, specialized query operators (as extension methods in static classes) that carry out specialized operations of collections.

The concept of *bites*, as introduced in this paper, is known as *slices* in other contexts. Some programming languages have special notation for slicing. A good example is Python [6] which generalizes the classical subscripting notation `seq[i]` to slicing notation `seq[i:j]`. In this expression both `i` and `j` are optional. Therefore, the Python expression `seq[:j]`, which extracts the first `j` elements of a sequence object, corresponds to `((bite-of-length j) seq)` using the bite generator `bite-of-length` from Section 2 of this paper.

As much more advanced notation, known as *list comprehension*, is supported in many programming languages (such as in Haskell [4] or Python [6]). List comprehension, which is inspired by conventional mathematical set building notation, is a syntactic abstraction over applications of mapping and filtering functions. Therefore, list comprehension may

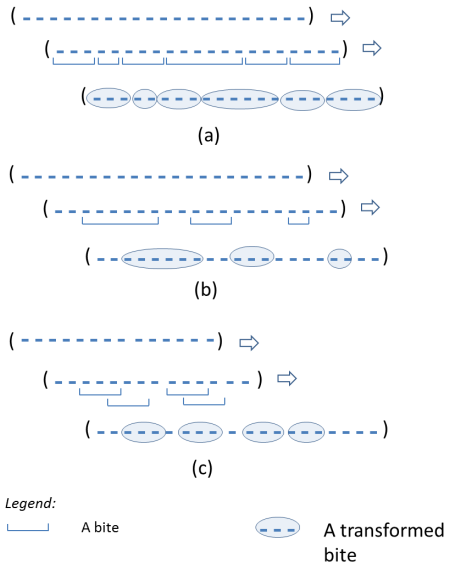


Figure 4: *Three different situations of bite taking and bite transformation.*

be used as an alternative to explicit mapping and filtering when we wish to process sublists of list.

During the last few years, the words "map" and "reduce" have been used to characterize a particular kind of parallel processing of very large collections of data. MapReduce [3], as used by Google, works on key/value pairs, and (in part) because of that, its relation to the original work on mapping, filtering and reduction is relatively weak. In the scope of this paper, it may be interesting to notice that the initial chunking of data (as a preparation for the parallel mapping in MapReduce) may be realized by taking bites of a list. The initial chunking is, however, not really a central part of MapReduce.

In an earlier paper about mapping and filtering in functional and object-oriented programming [11] we have described the idea of *general mapping*. General mapping is characterized by (1) element selection, (2) element ordering, (3) function selection (selection of function(s) to apply on the selected elements), (4) calculation (which transformation to apply), and (5) the result of the mapping. Relative to this understanding, the current paper contributes to the first aspect, namely a more elaborate way of selecting the part of list to be transformed in the mapping process.

6. CONCLUSIONS

The abstractions introduced in this paper are useful in situations where it is necessary to process selected sublists of a list, in contrast to individual elements of a list. As illustrated in Section 4 the generated bite functions are, together with the bite mapping functions, useful for discovering structures among the elements in a list. As a use case, we have demonstrated how a number of important music related structures can be captured in Standard MIDI Files.

Figure 4 illustrates three typical bite mapping scenarios, supported by our bite mapping functions. The most regular scenario, as supported by `map-bites`, is a *complete, disjoint chunking of a list* followed by *processing of the chunks*, as shown in Figure 4(a). With use of `step-and-map-bites` we can approach application areas in which we more exhaustively search for certain "sequences of consecutive elements" which together not necessarily span the entire list. This situation is sketched in Figure 4(b). As mentioned briefly in Section 3, it is also possible to extract and process overlapping sublists with use of `step-and-map-bites`. This situation is shown in Figure 4(c).

The organization of the sublisting facilities as higher-order functions has been the primary focus of this paper. We have striven for natural generalizations of the classical `map` and `filter` functions, which are well-known in most functional programming languages. Thus, the main emphasis in this paper has been to provide mapping and filtering of sublists via a few functions (such as `map-bites` and `filter-bites`) which take other functions are parameters. The bite functions introduced in Section 2 are of particular importance among these function parameters. We have explored how a number of useful bite functions can be produced by bite function generators, such as `bite-of-length`, and `bite-while-element`.

The functions discussed in this paper and their API documentation are available on the web [10].

7. ACKNOWLEDGEMENT

I would like to thank Prof. Dr. Christian Wagenknecht from Hochschule Zittau/Görlitz for helpful comments to an earlier version of this paper. I also thank the anonymous reviewers for very useful feedback to the version of the paper, which was submitted to the symposium.

8. REFERENCES

[1] Joseph Albahari and Ben Albahari. *C# 4.0 in a nutshell*. O'Reilly, 2010.

[2] Microsoft Corporation. The C# language specification version 4.0, 2010. http://www.microsoft.com/downloads/.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communication of the ACM*, 51:107–113, January 2008.

[4] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM Sigplan Notices*, 27(5), May 1992.

[5] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[6] Mark Lutz. *Programming Python*. O'Reilly and Associates, Inc, 2006.

[7] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3:184–195, April 1960.

[8] Kurt Nørmark. Web programming in Scheme with LAML. *Journal of Functional Programming*, 15(1):53–65, January 2005.

[9] Kurt Nørmark. MIDI programming in Scheme - supported by an Emacs environment. Proceedings of the European Lisp Workshop 2010 (ELW 2010), June 2010. http://www.cs.aau.dk/∼normark/laml/papers/-midi-laml-paper.pdf.

[10] Kurt Nørmark. Bites of lists - API documentation and source code. http://www.cs.aau.dk/∼normark/bites-of-lists/, March 2011.

[11] Kurt Nørmark, Bent Thomsen, and Lone Leth Thomsen. Mapping and visiting in functional and object-oriented programming. *Journal of Object Technology*, 7(7), September-October 2008.

[12] Ricardo Scholz and Geber Ramalho. COCHONUT: Recognizing complex chords from MIDI guitar sequences. In *ISMIR*, pages 27–32, 2008. `http://ismir2008.ismir.net/papers/ISMIR2008_200.pdf`.

[13] Olin Shivers. SRFI 1: List library. http://srfi.schemers.org/srfi-1/.

[14] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(Supplement S1):1–301, 2009.

[15] Guy L. Steele. *Common Lisp, the language, 2nd Edition*. Digital Press, 1990.

[16] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. Program. Lang. Syst.*, 13:52–98, January 1991.

[17] Wikipedia. Chord (music) — Wikipedia, the free encyclopedia, 2011. [Online; accessed 25-February-2011].

APPENDIX
A. DETAILED MIDI PROGRAMS

In this appendix we present more detailed examples related to the MIDI application area. The examples are all introduced and discussed at an overall level in the subsections of Section 4.

A.1 Bars

The MIDI function library contains a function `map-bars`, which activates `map-bites` with an appropriate bite function and bite transformation function. Here is a sample application of `map-bars` on some temporally strict music referred to as `SOME-MIDI-EVENTS`:

```
(map-bars
  (lambda (messages n st et) (list (midi-marker-abs-time st "Bar" n) messages))
  480                                                              ; Pulses Per Quarter Note.
  '(4 4)                                                           ; Time signature is 4:4.
  SOME-MIDI-EVENTS
)
```

In addition to the `messages` in the bar, the function mapped over the bar (the lambda expression shown above) receives a bar number `n`, the start time `st`, and the end time `et` of the bar. In the body of the lambda expression, we see that the messages in the bar are being prefixed with a MIDI marker.

In `absTime` mode, the function `map-bars` is implemented in terms of `map-bites` using a bite function generated by `bite--while-element`. Here is an outline of the use of `map-bites` in `map-bars`:

```
(define (map-bars f ppqn time-signature . messages)
  ...
  (map-bites
    (lambda (lst . rest)                                           ; The bite function.
     (let* ((start-time-first-mes (midi 'absTime (first lst)))
            (bar-number (quotient start-time-first-mes ticks-per-bar))     ; Zero based.
            (bar-start-time (* bar-number ticks-per-bar))
            (bar-end-time (+ bar-start-time ticks-per-bar))
           )
       ((bite-while-element (lambda (mes) (< (midi 'absTime mes) bar-end-time)) 'sentinel "first") lst)))
    (lambda (bite)                                                 ; The bite transformation
       (let* ((start-time-first-mes (midi 'absTime (first bite)))  ; function.
              (bar-number (quotient start-time-first-mes ticks-per-bar))
              (bar-start-time (* bar-number ticks-per-bar))
              (bar-end-time (+ bar-start-time ticks-per-bar))
             )
         (f bite (+ bar-number 1) bar-start-time (- bar-end-time 1)) ))   ; Activation of f on the bar.
     messages))
```

The `bar-number`, `bar-start-time` and `bar-end-time` are needed for taking a bar bite from the MIDI messages. As it appears, these values are recalculated in the bite transformation function, as a service to the function `f` being mapped to the bars of the music. These calculations can be lifted out of the `map-bites` application to a multi-valued function. But due to the unpacking of these values in both lambda expressions, the modified programs is not shorter, not simpler, and probably not more efficient than the version with the recalculations shown above. As discussed in Section 4.6, it seems to be typical that information calculated in the bite function also is useful in the bite transformation function.

As an example of a more elaborate use of `map-bars`, we will show how it is possible to slide the tempo of every fourth bar down to half speed, and back again to normal speed:

```
(map-bars
   (lambda (messages n st et)
     (if (and (> n 0) (= (remainder n 4) 0))                              ; Every fourth bar.
         (list
            (tempo-scale-1 20                                             ; Use tempo scaling.
               120
               (make-scale-function-by-xy-points                          ; A scaling function
                   (from-percent-points '((0 100) (50 50) (100 100))))    ; use for tempo scaling.
               120                                                        ; Base tempo is 120 BPM.
              messages
            )
            (midi-marker-abs-time st "Bar" n)                             ; Still inserting markers.
          )
          (list messages (midi-marker-abs-time st "Bar" n)))
   )
   480
  '(4 4)
   SOME-MIDI-EVENTS)
```

As it appears in the lambda expression shown above, bars with bar numbers divisible by 4 are tempo scaled by use of the function `tempo-scale-1`. The details of the tempo scaling is not relevant for this paper.

A.2 Pauses

At the top level, pauses are captured in a similar way as we located the bars in Appendix A.1.

```
(map-paused-sections
  (lambda (n mes-lst)
     (list (midi-marker "Start of paused section" n "P") mes-lst))
  130                                                                     ; Pause time ticks.
  (lambda (ast) (and (NoteOn? ast) (= (midi 'channel ast) 1)))            ; The relevance function.
  SOME-MIDI-EVENTS)
```

The function `map-paused-sections` has been implemented with use of `map-n-bites` and a bite function generated by `bite-while-element-with-accumulation`:

```
(define (map-paused-sections f pause-ticks relevance-predicate . messages)
 (map-n-bites
   (bite-while-element-with-accumulation
      (lambda (mes sound-frontier-time)                                   ; The predicate.
         (not (and (> (midi 'absTime mes) sound-frontier-time)
                   (> (- (midi 'absTime mes) sound-frontier-time)
                         pause-ticks))))
      (lambda (sound-frontier-time NoteOnMes)                             ; The accumulator.
         (max sound-frontier-time
             (+ (midi 'absTime NoteOnMes) (midi 'duration NoteOnMes))))
      0                                                                   ; The initial value.
      (lambda (x)                                                         ; The noise function.
        (and (ast? x)
             (or (not (relevance-predicate x)) (not (NoteOn? x)))))
   )
   (lambda (midi-messages-bite n)                                         ; The bite transformation
     (f n midi-messages-bite))                                            ; function.
    messages))
```

A.3 Sustain intervals

The following application of `map-sustain-intervals` illustrates how to obtain a faster release of the sustain pedal, without affecting the way the pedal is moved downwards.

```
(map-sustain-intervals
  1                                                          ; The channel affected.
  (lambda (messages n direction)                             ; The function mapped on
    (cond ((eq? direction 'increasing)                       ; intervals of messages that
           messages)                                         ; are monotone in sustain
          ((eq? direction 'decreasing)                       ; control messages.
             (scale-attribute-by-factor-1
                (lambda (ast) (ControlChange? ast 64 1))
                'value
                0.75
                messages))
          ((eq? direction 'constant)
             messages)
          (else (laml-error "Should not happen"))))
  SOME-MIDI-EVENTS)
```

Only in intervals with decreasing pedal movement, the `value` attributes of the appropriate `ControlChange` messages are scaled by the factor of 0.75.

The function `map-sustain-intervals` is implemented with use of `map-n-bites`. The bite function is generated by `bite--while-monotone`.

```
(define (map-sustain-intervals channel f . mes)
 (let ((cc-val-comp
         (make-comparator
            (lambda (cc1 cc2) (< (midi 'value cc1) (midi 'value cc2)))
            (lambda (cc1 cc2) (> (midi 'value cc1) (midi 'value cc2)))))
       (noise-fn (lambda (x) (not (ControlChange? x 64 channel))))
      )
  (map-n-bites
    (bite-while-monotone                                     ; The bite function generated
       cc-val-comparator                                     ; with bite-while-monotone.
       noise-fn)
    (lambda (mes bite-number)                                ; The bite transformation
       (f mes bite-number                                    ; function.
          (cond ((increasing-list-with-noise? cc-val-comp noise-fn mes)
                    'increasing)
                ((decreasing-list-with-noise? cc-val-comp noise-fn mes)
                    'decreasing)
                (else 'constant))))
    mes)))
```

As it appears, the sustain interval map function `f` gets information about the monotonicity of the MIDI message interval. This information has already been established in the bite function, but it needs to be re-calculated in the bite transformation function (via the two calls of `increasing-list-with-noise`). Without this information, we would not have been able to accomplish the task of dimming only the release of the pedal.

A.4 Chords

Chord identification makes use of `step-and-map-bites` instead of `map-bites`. Hereby the chords are identified in a more elaborate searching process than we have seen in the other examples. At top level, we search for chords in channel 1 of a piece of music in this way:

```
(map-chords
   1                                                         ; Channel number.
   40                                                        ; Max chord note distance.
   chord-marker                                              ; Chord markup function.
   SOME-MIDI-EVENTS)
```

The function `chord-marker` inserts markers (MIDI meta events) around a chord. In addition to a "chordal bite", `chord-marker` receives the channel number, the bite number, the successful chord formula, and the chord name.

Here follows the function `map-chords` in order to illustrate the use of `step-and-map-n-bites` and the bite function generated by `bite-while-element-with-accumulation`.

```
(define (map-chords channel max-time-diff f . messages)
 (let ((normalized-note-val (lambda (noteon-mes) (remainder (midi 'note noteon-mes) 12)))
       (relevant-message? (lambda (x) (and (NoteOn? x) (= channel (midi 'channel x)))))
      )
  (step-and-map-n-bites
     (bite-while-element-with-accumulation
      (lambda (mes prev-time)                                          ; Keep going while
          (if prev-time                                                ; notes are dense.
              (if (< (- (time-of-message mes) prev-time) max-time-diff)
                  #t
                  #f)
               #t))
      (lambda (time mes)                                               ; Accumulate time of
        (time-of-message mes))                                         ; previous note.
      #f                                                               ; Initial accumulation value
      (negate relevant-message?)                                       ; The noise function.
     )
     (lambda (bite)                                                    ; The int returning
       (let ((chord-list                                               ; predicate ...
              (map (lambda (no) (normalized-note-val no))
                   (filter relevant-message? bite))))
         (if (chord-match? (normalize-chord-list chord-list))          ; ... that determines a
             (length bite)                                             ; chord match
             -1)))                                                     ; or a stepping value.
     (lambda (bite n)                                                  ; The function applied on a
       (let ((normalized-chord-list                                    ; a chordal bite. Prepares
              (normalize-chord-list                                    ; calling f with useful
               (map (lambda (no) (normalized-note-val no))             ; information.
                    (filter relevant-message? bite)))))
         (f bite channel n normalized-chord-list
            (chord-name-of-normalized-note-list normalized-chord-list))))

  messages)))
```

As it appears, we locate chords in a given `channel`, among notes with `max-time-diff` time ticks between them. Only `NoteOn` messages in the given `channel` are taken into account. The noise function, formed by generating the negation of `relevant-message?` shown in line 3 of the fragment above, is important for disregarding MIDI events, which are irrelevant to the chord recognition process.

The Scheme Natural Language Toolkit (SNLTK)

NLP libraries for R6RS and Racket

D. Ćavar, T. Gulan,
D. Kero, F. Pehar,
P. Valerjev
University of Zadar

ABSTRACT

The Scheme Natural Language Toolkit (SNLTK) is a collection of procedures, example scripts and programs for natural language processing (NLP). The SNLTK library is fully documented and includes implementations of common data-structures and algorithms for text processing, text mining, and linguistic, as well as statistic analysis of linguistic data. Additionally, it also provides basic language data, word lists and corpus samples from various languages.

The SNLTK target group is Scheme and Lisp enthusiasts, computational linguists, linguists and language technology software developers. It is aiming at researchers and teachers, as well as students, who are interested in language related cognitive science, psychology, and linguistics.

Categories and Subject Descriptors

I.2.7 [**Computing Methodologies**]: Natural Language Processing; D.2.8 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms

Theory

Keywords

Scheme, Racket, NLP, SNLTK

1. INTRODUCTION

The SNLTK project started as a joint activity of faculty, assistants, and students from various departments at the University of Zadar, i.e. the Schemers in Zadar:

`ling.unizd.hr/~schemers`

The Schemers in Zadar is an open group of Scheme and Lisp enthusiasts. The goals of the group include, among others, the development of practical tools for computational linguistics, language related informatics and cognitive science in Scheme and Lisp. The resulting material should also serve as educational material for courses in the domain of Natural Language Processing, statistical language analysis and machine learning models.

The SNLTK is an outcome of joint NLP coding activities, and an attempt to aggregate the developed code and examples in an openly available general and specific text and language processing library.

`www.snltk.org`

The SNLTK is a collection of Scheme modules for various tasks in natural language processing (NLP), text mining, language related machine learning and statistical analysis of linguistic data.

The core libraries are written in R6RS Scheme, as well as in Racket (racket-lang.org). The code is tested for compatibility with common interpreters and compilers, e.g. Larceny.

The libraries are kept independent of external extensions and modules as much as possible, using the SRFI libraries where necessary. Additional programs, libraries and scripts are made available based on Racket. Racket is the recommended working and learning environment.

2. EXISTING TOOLKITS

Numerous other natural language processing tools, libraries and resources are implemented and available in various programming languages, e.g. Java, Perl, Python. Given the vast amount of NLP components and toolkits, we cannot discuss all the existing tools and libraries here. We will focus on the three most prominent packages and toolkits available for Java, Perl, and Python that are related to SNLTK and some of its goals.

In general, we should differentiate between speech and language processing. These two domains differ with respect to their nature and formal properties. While speech is concerned with the spoken signal, a non-discrete continuous event or phenomenon along the time axis, with specific issues related to its digitization, feature recognition and extraction, and consequently specific technologies, approaches, and algorithms, language refers to the indirectly observable properties of natural language that are related to combinatory and order relations and restrictions of sound groups, syllables, morphemes, words, and sentences. It is the lan-

ELS'2011 TUHH, Hamburg, Germany.
[European Lisp Symposium].

guage domain that the SNLTK is concerned with, and textual representations of natural language, rather than speech and signal processing.

Among the most popular of NLP toolkits is the Python Natural Language Toolkit (NLTK) [1]. The Python NLTK is a large collection of many common and popular algorithms for various text and natural language processing tasks. It contains algorithms for statistical NLP, as well as for rule-based analysis using common grammar types and symbolic approaches. Among the available libraries and tools it is the one widely used in educational computational linguistics programs worldwide. It is well documented, and significant amounts of teaching material and examples for it are freely available online. It contains implementations of the most important algorithms used in computational linguistics, as well as samples of common and valuable language data and corpora.

For Perl, a rich collection of tools and algorithms can be found in the CPAN archive (see www.cpan.org). Numerous sub-modules are available under Lingua::* module, including parsers for Link Grammars [9], WordNet ([14], [6]) access, and many other useful text mining and language processing tasks. Numerous valuable introductions to Perl for text processing and computational linguistic tasks are freely available online, or published as books, see e.g. [10].

Various tools and algorithms implemented in Java can be found online. Among the most prominent might be the OpenNLP (http://incubator.apache.org/opennlp/) collection of NLP tools, and the Stanford NLP software (http://nlp.stanford.edu/software/).

The coverage of the Python NLTK is most impressive, as well as the quality of the implementation. The mentioned Perl modules do as well offer impressive functionalities for various NLP oriented tasks and problems. Nevertheless, these implementations lack overall transparency at the implementation level, and provide less documentation and instructions related to the efficient and ideal implementation of particular algorithms and tools in the particular languages. It appears that they seem to have been designed with a clear usage orientation, rather than focusing on the educational goals of a deeper understanding of the particular algorithms and their implementation in the respective language or programming language paradigm.

Besides all the different implementations of computational linguistic algorithms for text processing, language and speech analysis, there are also numerous frameworks for the integration of specific language processing components for text mining. Among the most popular environments are Gate (cf. [21], [11]) and UIMA . These environments do not focus on particular language processing tasks, but provide an architecture for handling of sub-components in a language processing chain.

3. SNLTK GOALS

As mentioned in the previous section, for various computational linguistic tasks and problems, related to language, or even speech processing, many tools, libraries and components can be found online. The SNLTK is not intended to compete with these tools for applied computational linguistics tasks. It does not even intend to provide better, faster, or new solutions for common problems, or new types of implementation strategies for known or new algorithms. Its main goal is in fact an educational, experimental, and research oriented one. In addition, it provides alternative functional implementations of common, and potentially also new NLP algorithms.

One the one hand, many algorithms that we implemented for research and experimental purposes in the past, have been generalized and added to the library, some have been simplified in order to be easier understandable and analyzable. Many more will be prepared and added in the near future. Thus, an initial set of algorithms and tools in SNLTK is based on implementations from experiments and research projects that had a potential of being usable elsewhere.

On the other hand, we have chosen Scheme as our development and educational language (in addition to, and also replacing Python) for various reasons. Scheme is a very simple, but powerful language. It is easy and fast to learn, and simple to use. Further, various tools, in particular the intuitive IDE DrRacket (former DrScheme) for learning Scheme is available as a cross-platform environment, free of charge, and without runtime restrictions. It contains various learning packages, and freely available books and tutorials are distributed with it, and also available elsewhere online. DrRacket appeared to be the ideal environment for educational purposes. It is used in many programs in computer science, and large amounts of teaching material and examples are available online. However, it has not been widely used for computation linguistic courses, which made it necessary to collect and also re-implement algorithms for educational purposes.

In addition to being a very good educational and research language, with very good development tools like DrRacket, there are many useful tools for Scheme that allow it to be used for professional software development as well. In particular, compilers and interpreters exist that generate binaries, or code translation into other languages (e.g. C), or very good connectivity between languages like Java and C♯. Among the most interesting implementations of such interpreters and compilers we should mention Gambit-C (e.g. [5]), Larceny (e.g. [2]), Bigloo ([20], 1995; [19]), and Chicken Scheme (see www.call-cc.org). The possibility to generate standalone binaries, or translate code automatically into other programming languages is rather limited or non-existent in some of the other languages (e.g. Python and Perl) that natural language toolkits exist for.

Various books and educational material for computational linguistics based on Common Lisp are already available. One of the seminal publications on Lisp and Computational Linguistics is [8]. In addition, [18] offer many computational linguistics related Lisp implementations of algorithms. While we consider these textbooks extremely valuable and important, they nevertheless lack a discussion of modern approaches and implementation strategies in the domain of computational linguistics.

While we focus on the implementation of Scheme libraries,

future releases or parallel versions might be geared towards ANSI Common Lisp. The use of CUSP and Eclipse as a development environment is a possible path, since affordable commercial Lisp development environments tend to be outside of the range for the academic and research community.

4. LIBRARY CONTENT

In its current state the SNLTK contains basic procedures and data from domains like:
– Finite State Automata
– Parsing with Context Free Grammars
– N-gram models
– Vector Space Models and algorithms for classification and clustering
– Basic language data for various languages
– Additional components

Specific statistical procedures for the processing of N-gram models are being developed, as well as document classification and clustering functionality.

In the following we shall describe some of the subcomponents and functionalities implemented in the SNLTK.

4.1 Finite State Automata

Finite State technologies for spell checkers, morphological analyzers, part of speech taggers, shallow parsers, and various other approaches to NLP are well known and discussed in the literature, see for example [15], and [16]. Finite State Automata are used for spell checkers, and shallow parsers, wherever regular grammars or languages are sufficient for natural language processing.

We made various implementations of FSAs for lexicon compression, morphological analyzes (Croatian morphological analyzer), and simple word class recognition available in the SNLTK. The current implementation makes use of table based FSA implementations. Basic functionalities include the generation of acyclic deterministic FSAs (ADFSA) from finite word lists, as well as common operations like unions, minimization, and concatenations over ADFSAs. Missing functionalities include conversions of non-deterministic automata to deterministic ones, the construction of transducers or Moore/Mealy machines ([17], [12]), and various other optimizations and code export.

Currently ADFSAs can be exported as DOT definitions for visualization and interoperability (e.g. code generation, transformation using Graphviz and related tools). A future version should be able to export C code definitions of automata, maybe even directly assembler.

4.2 Parsing with Context Free Grammars

Natural language syntax is formally beyond regular languages and the coverage of regular grammars. Thus, for syntactic processing, various parsing algorithms are implemented that use context free grammars (CFG). Simple algorithms like bottom-up or top-down parsers are part of the parser library, as well as an Earley Parser [4], and other types of chart parsers, using agenda-based processing strategies.

In addition to the simple parser implementations, graphical visualization widgets for Racket have been implemented that display balanced syntactic parse trees.

For higher level syntactic processing that makes use of lexical feature structures, first versions of unification parsers are included in the library as well, see [3].

4.3 N-gram models

Various language processing algorithms make use of statistical models of distributional properties of linguistic units, e.g. sounds, morphemes, words and phrases. An approximation of such distributional properties can be achieved using N-gram models.

Various tools for different types of N-gram models are included in the SNLTK. It is possible to create character-based N-gram models from textual data, that are useful for language identification and approximations of syllable structure. In addition, word token-based N-gram model generators exist as well, that extract statistical distributional models over word co-occurrences.

Besides token co-occurrence patterns, bags of tokens or types can be generated from N-gram models as well. Frequency profiles can be calculated in terms of absolute and relative frequencies. Some other language tools which include filtering of functional words, lemmatization and normalization can be applied in different stages of N-gram model generation. Once a N-gram model is generated frequencies can be weighted by tf-idf weight (term frequencyâĂŞinverse document frequency) before generation of vector space models.

Information theoretic measures are included as well. The Mutual Information score for example can be calculated for models and individual N-grams, which is useful for finding correlation between uni-grams or for bi-gram weighting. Other implemented statistical testing functions include chi-squares and t-tests for testing similarities or differences between models.

4.4 Vector Space Models and algorithms for classification and clustering

For various models of distributional properties of tokens and words in text, as well as complex text models, vector space models appear to be very useful. Not only multivariate statistical properties can be processed using such models, but these models are also language independent, i.e. generalize well over multiple languages with fundamentally different properties at various linguistic levels (e.g. syllable structure, syntactic word order restrictions).

One the one hand, simple mapping algorithms of raw text and N-gram models to Vector Space Models are provided in the SNLTK.

For classification and clustering tasks various similarity metrics algorithms are implemented, that are used among others in a K-Means algorithm for text clustering. As similarity metrics, various distance measures like Euclidean Distance and Cosine Similarity are provided. Further scaling algorithms for vector models are provided, including re-ranking on the basis of tf-idf, as well as statistical significance testing

on the basis of e.g. chi-square tests.

4.5 Basic language data for various languages

Besides example corpora and word lists for specific languages, in particular stop word lists for various languages are provided. The creation of additional stop word lists is facilitated by a library which exports them as lists or a hashtable data structure. In addition to hashtable based lookup lists, finite state automata can be generated from word lists.

4.5.1 Example corpora

Text collections for various languages are being prepared that facilitate language model training and tuning, as well as testing of existing algorithms.

4.6 Additional components

4.6.1 Basic tree visualization (Racket)

The SNLTK also allow us to use Racket interpreter to generate N-gram models and produce DOT representations of directed graphs based on the underlying N-grams that can be visualized with Graphviz or any other alternative tool for DOT-based graph visualization. The current n-gram2dot release represents frequencies via heavier weights and length of edges and also different weight of nodes [7]. Graphviz can generate different output formats for graphs like PDF, PNG, SVG.

4.7 Documentation

The documentation of the SNLTK library and tools is provided initially in English. All documents are translated to German, Polish and Croatian as well.

The SNLTK documentation contains the library description, as well as example documents related to the use of specific library functions, and algorithm implementation issues. One of the main goals of SNLTK is to provide a detailed documentation of the algorithms, their implementation, as well as different examples of use.

5. CONCLUSION

On the one hand, from our experience, we can conclude that Scheme and current Scheme development environments like DrRacket are very useful for the development of NLP tools, as well as for learning and education purposes.

The SNLTK in its current form is subject to further development and extension. Its further directions are guided by the research and educational interests of the participating researchers and developers.

Future releases of the SNLTK will most likely include more NLP algorithms, as well as machine learning algorithms, related, but not restricted to the domain of cognitive modeling and language learning.

References

6. REFERENCES

[1] Bird, S., Klein, E., and Loper, E. 2009. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit.* O'Reilly Media.

[2] Clinger, W.D. 2005. Common Larceny. In the *Proceedings of the 2005 International Lisp Conference*, June 2005, pages 101–107.

[3] Dybvig, R.K. 2009. *The Scheme Programming Language*. 4th edition. Cambridge, MA: MIT Press.

[4] Earley, J. 1970. An efficient context-free parsing algorithm, *Communications of the Association for Computing Machinery* 13.2, 94–102.

[5] Feeley, M. 2010. *Gambit-C*. Online documentation, http://www.iro.umontreal.ca/ gambit/doc/gambit-c.html.

[6] Fellbaum, C. 1998. *WordNet: An Electronic Lexical Database*. Cambridge, MA: MIT Press.

[7] Gansner, E., Koutsofios, E., and North, S. 2006. *Drawing graphs with dot. dot UserâĂŹs Manual*, 1–40.

[8] Gazdar, G., Mellish, C. 1990. *Natural Language Processing in LISP: An Introduction to Computational Linguistics*. Boston, MA: Addison-Wesley Longman Publishing.

[9] Grinberg, D., Lafferty, J., Sleator, D. 1995. A robust parsing algorithm for link grammars, *Carnegie Mellon University Computer Science technical report* CMU-CS-95-125.

[10] Hammond, M. 2003. *Programming for Linguists: Perl for Language Researchers*, Blackwell.

[11] Konchady, M. 2008. *Building Search Applications: Lucene, LingPipe, and Gate*. Mustru Publishing.

[12] Mealy, G.H. 1955. A Method to Synthesizing Sequential Circuits. *Bell Systems Technical Journal* 34. 1045–1079.

[13] Sperber, M., Dybvig, R.K., Flatt, M., Straaten, A.v., Findler, R., and Matthews, J. 2010. Revised [6] Report on the Algorithmic Language Scheme. Cambridge University Press.

[14] Miller, G.A. 1995. WordNet: A Lexical Database for English. *Communications of the ACM* Vol. 38, No. 11: 39–41.

[15] Mohri, M. 1996. On Some Applications of Finite-State Automata Theory to Natural Language Processing. *Natural Language Engineering* 1.1.

[16] Mohri, M. 1997. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics* 23.2, 269–312.

[17] Moore, E.F. 1956. Gedanken-experiments on Sequential Machines. Automata Studies, *Annals of Mathematical Studies* 34, 129–153.

[18] Russell, S., and Norvig, N. 2009. *Artificial Intelligence: A Modern Approach*. 3rd edition. Upper Saddle River, NJ: Prentice Hall.

[19] Serrano, M. 1996. *Bigloo user's manual*. Technical Report, Inria.

[20] Serrano, M., and Weis, P. 1995. Bigloo: a portable and optimizing compiler for strict functional languages, *SAS* 95, 366–381.

[21] Wilcock, G., and Hirst, G. 2008. *Introduction to Linguistic Annotation and Text Analytics* (Synthesis Lectures on Human Language Technologies). Claypool Publishers.