Proceedings of ELS 2014

7th European Lisp Symposium

May 5 – 6 2014
IRCAM, Paris, France

ISSN: 2677-3465

Preface

Message from the Programme Chair

Greetings, members of the Lisp Community.

We are a diverse community. Some have been programming in Lisp for some time, some are just starting out, and some are even just peering in curiously from outside—including, in some cases, family and friends.

A variety of dialects are represented as well: Common Lisp, Racket, Clojure and Emacs Lisp to name a few about which we have papers. But we've invited attendance from Scheme, AutoLisp, ISLISP, Dylan, ACL2, ECMAScript, SKILL, and Hop. Even then, I'm sure to have left some out. The important thing is that these are all Lisp. In the work I did with language stands, I recall John McCarthy specifically asking that ANSI and the ISO not name their work product anything that appeared to claim the whole of the name Lisp. He wanted to reserve that for the family of languages, which he felt should continue to freely evolve and add new members. This gathering honors that spirit by inviting that whole extended family.

So, to all of this diverse community: Welcome!

We're all from different backgrounds, but each brings something to contribute in terms of experience, interest, or enthusiasm. We are different, but today we're also all equals, here to exchange ideas and energy. In that spirit, please make sure to visit not just with old friends but also with those you have not met, making them feel that sense of welcome so they'll have reason to return and you'll have even more old friends the next time. Or, if you're new, please don't be shy: Step up and introduce yourself to any among us even if we've slipped and forgotten to reach out to you.

Also, please find a time during the conference to thank Didier Verna and Gérard Assayag for acting as Local Chairs, and to thank members of the program committee for working on a very tight time schedule to select the papers and provide excellent quality feedback to authors. Thanks are also due to IRCAM, which has provided our intellectually stimulating venue as well as practical logistical support. And, of course, the backbone of the conference is the actual content, so make sure to tell the invited speakers, the paper presenters, those performing demonstrations, and those who offer lightning talks that their presence has made a difference. As you can probably see, a lot goes into preparing a conference like this, and no one's getting rich off of it —— at least not in money. But your stated appreciation will go a long way toward making the effort of all of these people seem worthwhile.

Thanks also to you for attending. This conference would be nothing without attendees. I hope you have a great time.

Kent Pitman

Message from the Local Chair

Welcome to Paris!

ELS 2014 is undoubtedly one of the most successful events in the European Lisp Symposium series, with almost 80 delegates from nearly 20 countries, including quite distant ones such as Japan, India and the United States of America. More than just "European", ELS is now indeed fully international.

It is great to see such a crowd gathering in Paris, and I hope you will have a great time meeting fellow lispers from all over the world, visiting the "Lambda Tower" and all the touristic wonders Paris has to offer.

As a part-time researcher, part-time musician, I need to stress that holding the symposium at IRCAM is also a very special pleasure for me. IRCAM is an amazing place for music research and also a nest of hard-core lispers. We couldn't have hoped for a better place. In that regard, I am deeply grateful to my co-local chair, Gérard Assayag, without whom this would simply not have been possible.

In order to organize an event like this one, many other people work in the shadows, although shadows are crucial to make ELS a bright event. I need to extend my warmest thanks to Daniela Becker and Sylvie Benoît, who took care of a lot of things. Like every year, our sponsors also are of a tremendous importance, so thank you, EPITA, CNRS, UPMC, LispWorks, and Franz!

Finally, and this time, not as a local chair but as the president of the European Lisp Symposium's steering committee, I wish to express my deepest gratitude to Kent Pitman, who accepted the role of programme chair this year, perhaps not knowing exactly what he was getting into… From where I stand, working with Kent has been a very smooth experience, one that I would redo anytime.

Let us now hope that you will have as much fun attending the symposium that we had organizing it. In just one sexp…

```
(have :fun #\!)
```

Organization

Programme Chair

- Kent M. Pitman, Hypermeta Inc., USA

Local Chairs

- Didier Verna, EPITA/LRDE, Paris, France
- Gérard Assayag, IRCAM, UMR STMS (CNRS, UPMC), France

Programme Committee

- Marie Beurton-Aimar, LaBRI, University of Bordeaux, France
- Pierre Parquier, IBM France Lab, Paris, France
- Rainer Joswig, Hamburg, Germany
- Guiseppe Attardi, Università di Pisa, Italy
- Taiichi Yuasa, Kyoto University, Japan
- António Leitão, IST/Universidade de Lisboa, Portugal
- Christophe Rhodes, Goldsmiths, University of London, UK
- Olin Shivers, Northeastern University, USA
- Charlotte Herzeel, IMEC, ExaScience Life Lab, Leuven, Belgium

Organizing Committee

- Daniela Becker, EPITA/LRDE, Paris, France
- Sylvie Benoit, IRCAM, Paris, France

Sponsors

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

IRCAM
UMR STMS (CNRS, UPMC)
Institut de Recherche et Coordination
Acoustique / Musique
1, place Igor-Stravinsky
75004 Paris
France www.ircam.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

Contents

Invited Talks

Making Creativity: Software as Creative Partner

Richard P. Gabriel, IBM, Redwood City, CA, USA

Programming, software development, and software engineering: We are taught to solve puzzles and do what we're told. We carry these lessons into our jobs and careers without deliberation. Old fashioned software engineering aims to make no mistakes; agile aims to render programmers compliant, and commands them make money for their bosses. For the past year I've been exploring what creativity means during the act of writing, and I've been doing it by constructing a software partner that acts as a scientific engine of discovery —— a partner that displays a flair for the strange that even the most daring poets can rarely match. I don't have requirements, I don't have specifications, and I normally don't have a plan much beyond a guess. If my program doesn't surprise me, I cry "failure!" and lament.

I'll explore what programming is, how software can act as a collaborator, show you how the agile practices are like training wheels, and explain how a program can astound.

All in Lisp, of course.

Richard P. "Dick" Gabriel overcame a hardscrabble, working-class upbringing in the dreadfully industrialized and famously polluted Merrimack Valley of eastern Massachusetts to become one of the few genuine Renaissance men to emerge from the OO milieu: scholar, scientist, poet, performance artist, entrepreneur, musician, essayist, and yes, hacker…

Though somewhat less well-endowed of the effortless intellectual incandescence, easy charisma, and raw animal magnetism of so many of his generation of future object-oriented luminaries, he was able, with discipline, determination, and hard work, to survive the grueling demands of elite, first-tier academic institutions such as MIT, Stanford and UIUC to earn his PhD and become a leader among the burgeoning legions of Lisp-dom during the early nineties.

However, after a series of the inevitable, endemic startup setbacks that the Internet boom all too often left in its wake, Gabriel grew weary of the cold, cloistered, celibate tedium of engineering culture, and fell willing prey to the lure of the exotic social and intellectual stimulation and blandishments that only the Liberal Arts could offer.

And they, in turn, embraced this gruff emissary from the exotic, intimidating, but newly chic world of technology. Gabriel's dissonant, desiccated, plainchant blank verse was dark, disturbing, distant, candid, calculating, and desperate, at once florid yet monochromatic. It could "cons-up" a soul in a single haunting, searing stanza and remand it remorselessly, insouciantly to the heap in the next. It was like nothing that could be heard on the stale, staid, inbred Writers' Workshop circuits of those times.

But then, as always, poetry alone seldom pays the bills, so the prodigal poet, like a salmon to spawn, returned to his object-oriented roots, proselytizing a newfound artistic sensibility to an aesthetically impoverished community.

His technological audiences, who had subsisted on bland, austere stylistic pabulum born of their collective status as a poor stepchild of mathematics, physics, and engineering, embraced his audacious set-piece guerilla performances and this novel aesthetic dimension in a manner akin to that in which Medieval European palates had embraced the infusion of spices from the East Indies.

His considerable successes in synthesizing the "Two Cultures" in this software setting will likely stand as his enduring legacy.

Gabriel lives in Redwood City, CA, and works for International Business Machines Corporation as a research helper. He likes to unwind by writing a poem every day.

Parallel Programming with Lisp, for Performance

IMEC, ExaScience Life Lab, Leuven, Belgium

This presentation gives an overview of parallel programming constructs and primitives, and how they can be used efficiently from within Common Lisp. The focus of this talk is on taking advantage of multi-core processors for improving the performance of algorithms. For this reason, the most important techniques for achieving efficiency in general will also be covered. The presentation will be based on examples from high performance and life sciences computing.

Pascal Costanza works as a researcher specializing on high-performance computing at the ExaScience Life Lab for Intel Belgium. He maintains Closer, an open source project that provides a compatibility layer for the CLOS MOP across multiple Common Lisp implementations. In the past, he has implemented ContextL, the first programming language extension for Context-oriented Programming based on CLOS, and aspect-oriented extensions for CLOS. More recently, he released Arrow Street, a template library for C++11 to support semi-automatic SIMD-efficient data layouts.

Sending Beams into the Parallel Cube

Gábor Melis, Franz Inc., Hungary

A pop-scientific look through the Lisp lens at machine learning, parallelism, software, and prize fighting

We send probes into the topic hypercube bounded by machine learning, parallelism, software and contests, demonstrate existing and sketch future Lisp infrastructure, pin the future and foreign arrays down.

We take a seemingly random walk along the different paths, watch the scenery of pairwise interactions unfold and piece a puzzle together. In the purely speculative thread, we compare models of parallel computation, keeping an eye on their applicability and lisp support. In the the Python and R envy thread, we detail why lisp could be a better vehicle for scientific programming and how high performance computing is eroding lisp's largely unrealized competitive advantages. Switching to constructive mode, a basic data structure is proposed as a first step.

In the machine learning thread, lisp's unparalleled interactive capabilities meet contests, neural networks cross threads and all get in the way of the presentation.

Gábor Melis is a consultant at Franz Inc. He worked on the SMP thread implementation of both SBCL and AllegroCL and got interested in all things parallel. Had he been a first generation lisper, he would have been in the artificial intelligence bubble of the 80s, but as a second generation lisper all that was left for him is machine learning. There are two kinds of programmers: those who get a kick out of their creation being used by real people to accomplish a goal, and those who first and foremost think in terms of the internal beauty of the object at hand. Most of the time, Gábor is firmly in the latter category and for his practical side to emerge he needs external pressure. Fortunately, external pressure is abundant in contests which he finds and leverages with some success.

Session I: Language and Libraries

CLAUDE – The Common Lisp Library Audience Expansion Toolkit

Nick Levine
Ravenbrook Limited
PO Box 205
Cambridge, CB2 1AN
United Kingdom
ndl@ravenbrook.com

ABSTRACT

CLAUDE is a toolkit for exporting libraries written in Common Lisp, so that applications being developed in other languages can access them. CLAUDE co-operates with foreign runtimes in the management of CLOS objects, records, arrays and more primitive types. Lisp macros make the task of exporting a library simple and elegant; template documentation along with C headers and sample code files relieve some of the burden of explaining such exports to the application programmer.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability—*Distributed objects*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

1. INTRODUCTION

One the the ways in which most Common Lisp implementations extend the ANSI standard is by providing some means of distributing a lisp application as a pre-compiled executable. Another essential extension is the capability to link into libraries written in other languages. The first of these features greatly extends the potential audience of an application, exposing it to end-users who don't ever want to meet a lisp loader (and indeed have no reason to ever know what language the application was written in). The other gives the application writer access to a wealth of foreign libraries and so leaves them freer to get on with solving their core problems.

Less obvious is the possibility of distributing a lisp program as a pre-compiled library, into which authors of applications written in other languages could link. These authors form a vast talent pool, and their users a massive audience base beside which the number of people running applications written in lisp is regrettably less impressive.

In this paper we will discuss a nascent toolkit intended to address this problem. The *Common Lisp Library Audience Expansion Toolkit* (CLAUDE)[1] includes:

- macros for exporting classes and functions;
- shared memory strategies for the allocation and freeing of objects, records, arrays and strings;
- *callbacks* (which in this context means calls from lisp back into the outside world of the application);
- support for UTF-8; high data throughput; full thread-safety; error handling with backtraces visible to the application programmer;
- tests, and templates for generating documentation and C / Python examples.

Authors of libraries written in lisp, in their search for a wider audience, can use CLAUDE to write a thin interface layer on top of their code; the result is saved as a DLL. They should extend the provided header and sample code files and template documentation, to cover their own exported classes and functionality as well as CLAUDE's. Their distribution to application programmers consists then of the DLL plus these extended files. Although the explicit support is for C and Python, there's no particular reason why applications should confine themselves to these two languages: CLAUDE is totally agnostic about the application's implementation choices. In theory CLAUDE can be even used to link two different lisp implementations (we might then, say, drive the LispWorks *CAPI* windowing environment from SBCL); however CLAUDE's interface is deliberately aimed at lower-level languages than Common Lisp and so in practice talking to other lisps won't be its greatest strength.

We will distinguish two roles:

- the *library writer* uses CLAUDE to export their Common Lisp library; and
- the *application programmer* uses that library as part of their (C / Python / other) application.

[1] `http://www.nicklevine.org/claude/`

In the following we will use Python – and to a lesser extent C – to illustrate the foreign (i.e. application) side of several CLAUDE exports.

Hitherto CLAUDE has been deployed only on LispWorks; some LW specifics are present in the code fragments below, exported from the packages SYS and FLI. Other implementations for which a port is possible should be accessible in return for reasonable effort. Indeed where convenient CLAUDE uses Bordeaux Threads (BT) and the Common Foreign Function Interface (CFFI), to help reduce the cost of future ports. However, and this is most unfortunate, most lisps will not be able support CLAUDE, because the facility to save lisp code as a linkable library is not generally available. We expect additional ports to be limited to Allegro and to the lisps which directly embed into other runtimes (ABCL and ECL).

2. EXAMPLE

The inspiration for the present work was Ravenbrook's *Chart Desktop*: a free, pre-existing, 15k LoC library for the layout and interactive display of large, complex graphs[2]. Here, by way of an introduction to CLAUDE's capabilities, is an example of driving part of the Chart library externally.

On the lisp side, we can say:

```
(defclass-external graph (chart:graph)
  ())

(defun-external (new-graph :result-type object) ()
  (make-instance 'graph))
```

In both C and Python there's quite a bit of overhead before we can get started. The CLAUDE distribution includes all the code for that; once this is out of the way, we can now go:

```
claude_handle_t graph;
CHECK(chart_new_graph(&graph));
```

Here `chart_new_graph()` is an export from the library, automatically created by the `defun-external` form; note the mapping between the lisp and foreign function names. `CHECK` is an unedifying C macro provided with CLAUDE for confirming that the call into lisp returned a success code. After this call, `graph` will contain a CLAUDE *handle*: a value which can be passed back into lisp at any time to refer unambiguously to the CLOS object in question.

In Python it's worth establishing a more sophisticated correspondence with the CLOS object than the raw pointers that C provides:

```
>>> from pyChart import chart
>>> chart.Graph()
<Chart Graph handle=0x20053748>
>>>
```

[2] `http://chart.ravenbrook.com/downloads`

Here's the application writer's Python code to drive this. The CLAUDE function `val()` makes the foreign call (i.e. into lisp), checks for a success code, and dereferences the pointer result; `init()` establishes a correspondence (in a Python dictionary) between the new Graph instance and its Chart handle.

```
class Graph(ClaudeObject):
    def __init__(self):
        handle = val(dll.chart_new_graph)()
        ClaudeObject.init(self, handle)
```

Meanwhile, provided the lisp library was configured when it was saved to open a REPL on restarting, we can check:

```
(in-package claude)
(loop for w being the hash-values of *wrappers*
      return (wrapper-object w))
->
#<Chart Graph handle=0x20053748>
```

Note the similarities in printed representations, between the CLOS object and its Python counterpart.

3. DATA MODEL

CLAUDE functions take arguments of the following types:

- integer (signed and unsigned);
- UTF-8 encoded string;
- *handle* (used to denote a CLOS object within the library);
- *record* (in this paper: a sequence of values whose length and constituent types are determined by the context in which it's being passed);
- *array* (in this paper: a sequence of values whose constituent types are all the same, but whose length is not known in advance);
- pointer to any of the above (for return values);
- pointer to function (for callbacks).

There's no particular reason why floats can't be supported; Chart happened not to need them.

It's vital that the responsibilities for allocating and recycling memory are clearly laid out, and that CLAUDE co-operates transparently with foreign memory managers.

Aggregate values – strings, records (for example, a pair of integers representing x-y co-ordinates) and arrays (for example, a list of such co-ordinates, or a list of CLAUDE objects) – may be generated by either the library or the application.

The contents of any aggregate value created by the application, passed into the library and retained there – for example, a string to be drawn inside some display window, or an

array of records describing a set of new objects to be added to the system – are copied and retained by CLAUDE. So the application may immediately recycle any memory associated with such aggregates, after the library call using them has returned.

An aggregate value created within the library and passed to the application as return data will be retained by CLAUDE, until such time as the application declares that it has no further use for this by calling the library function `claude_free()`. If the application frees a record or array which itself contains any aggregate values, then these values will be freed recursively.

```
CHECK(chart_new_nodes(&nodes, graph, node_defs));
node_0 = nodes->values[0].handle;
node_1 = nodes->values[1].handle;
/* Tell Chart it may free the array in which it
   returned the nodes. */
freeable.array = nodes;
CHECK(claude_free(freeable));
```

A handle may only be generated by the library, by making an instance of an external class (one defined by `defclass-external`) and then returning that instance from a call to `defun-external`, as in the `new-graph` or `chart_new_nodes()` examples above. When the application has no further use for a set of handles, it should call `claude_remove_objects()` to invalidate the CLAUDE objects and permit memory to be recycled on both sides of the fence. Here's the corresponding lisp function for that:

```
(defun-external (remove-objects
                 :result-type (array object
                                     :call 'drop))
    ((array (array object)))
  (remove-duplicates
   (loop for object in array
         append (remove-object object))))
```

Note that both the arguments and the return value of `remove-objects` are (CLAUDE) arrays of objects. This interface is designed to allow the library to:

1. take further action when an object is invalidated;
2. cause the invalidation of one object to invalidate others with it (for example, in Chart, if the application removes a node from the graph then the library will ensure that the node's edges are removed as well); and
3. decline to invalidate an object, if it so chooses.

The calls to `remove-object` allow the library to update its structures and specify secondary removals; supplying `'drop` against the `:call` keyword invokes the following method on each of the dead objects, to recycle lisp memory:

```
(defmethod drop ((self object))
  (let ((wrapper (shiftf (object-wrapper self)
                         nil)))
    (setf (wrapper-object wrapper) nil)
    (remhash wrapper *wrappers*)
    wrapper))
```

Finally, `remove-objects` returns to the application the full set of invalidated handles. The CLAUDE objects corresponding to each of these handles have been turned over for garbage collection; the application should no longer communicate with the library about those objects and any attempt to do so will signal a CLAUDE error. If the application has been mapping the handles into its own objects then these should now be recycled:

```
def _discard(self):
    handle = self.handle
    del _objects[handle]
    self.handle = None
```

4. ARGUMENT TRANSLATION

Every CLAUDE object has a *wrapper* which is a statically allocated structure, doubly linked to the object:

```
(defmethod initialize-instance
          :after ((self object) &key)
  (setf (object-wrapper self) (make-wrapper self)))

(defun make-wrapper (object)
  (let ((wrapper (sys:in-static-area
                  (in-make-wrapper object))))
    (setf (gethash wrapper *wrappers*) wrapper)
    wrapper))

(defstruct (wrapper
            (:constructor
             in-make-wrapper (object)))
  object)
```

The external representation of an object is its wrapper's memory address, guaranteed never to change. The generic-function `box` which retrieves this address is cautious about possible errors (as is its counterpart `unbox` which converts incoming addresses back into the corresponding CLAUDE objects):

```
(defmethod box ((self object)
                &key &allow-other-keys)
  (if (slot-boundp self 'wrapper)
      (let ((wrapper (object-wrapper self)))
        (if wrapper
            (sys:object-address wrapper)
          (error "~a doesn't have a handle."
                 self)))
    (complain "~a has not yet been initialised."
              self)))
```

```
(defmethod box ((self null) &key allow-null)
  (if allow-null
      0
    (call-next-method)))

(defmethod box (self &key &allow-other-keys)
  (complain "~a is not a ~a object and cannot ~
             be boxed."
            *library* self))
```

Note the use of zero to denote null objects, when the library's API permits this.

Boxing and unboxing are handled behind the scenes, by a LispWorks *foreign-converter* object. Such converters make the argument and result handling of `defun-external` elegant (and indeed quite invisible to the library writer).

```
(fli:define-foreign-converter object
    (&key allow-null type) (object address)
  :foreign-type 'uint
  :foreign-to-lisp `(unbox ,address
                           :allow-null ,allow-null
                           :type ,type)
  :lisp-to-foreign `(box ,object
                         :allow-null ,allow-null))
```

We suggest that if possible the application should mirror the above process, if not its elegance, so that it too is dealing with objects rather than raw pointers:

```
_objects = {0:None}

class ClaudeObject(object):
    def __init__(self, handle):
        _objects[handle] = self
        self.handle = handle

    def init(self, handle):
        self.__init__(handle)

    def box(self):
        return self.handle

def unbox(handle):
    return _objects[handle]
```

Depending on the level of introspection exposed by the library, the application's `unbox` might be used interactively as a debugging aid. For example in Chart, faced with the following error...

```
pyChart.invoke.ChartError: Source and destination
are the same node (#<Chart Node handle=0x20053838>),
which is not permitted.
```

...`unbox` can identify the object with handle `0x20053838`, and then `chart_describe_nodes()` can ask the library to provide further information about it:

```
>>> unbox(0x20053838)
<Chart Node handle=0x20053838>
>>> chart.describe_nodes([_])
[(<Chart Graph handle=0x20053748>,
  [], u'Hello World!')]
>>>
```

Alternatively, a little cut-and-paste allows us to access the object on the lisp side:

```
CLAUDE 17 > (unbox #x20053838)
#<Chart Node handle=0x20053838>

CLAUDE 18 > (describe *)

#<Chart Node handle=0x20053838> is a CHART:NODE
WRAPPER           #<WRAPPER Node 2005383B>
GRAPH             #<Chart Graph handle=0x20053748>
LABEL             "Hello"
EDGES             NIL

CLAUDE 19 >
```

A record is passed as a raw sequence of values; an array is implemented as a record whose first member is the array's length, and whose remaining members are the array's values. Corresponding to `box` and `unbox` for objects, we have `construct` and `deconstruct` for records, and `pack` and `unpack` for arrays.

```
def set_callbacks(object, callbacks):
    c_tuples = map(lambda((name, function)): \
                   (c_string(name), function or 0),
                   callbacks)
    records = map(construct, c_tuples)
    array = pack(records)
    boxed = object.box() if object else None
    lib.set_callbacks(boxed, array)
```

In LispWorks both of these aggregate types are handled by foreign-converters, and once again the library writer is isolated from any unpleasantness. For example (decoding of incoming record addresses):

```
(fli:define-foreign-converter record
    (types &key allow-null) (objects address)
  :foreign-type 'uint
  :foreign-to-lisp `(deconstruct ,address ',types
                                 :allow-null
                                 ,allow-null)
  :lisp-to-foreign `(construct ,objects ',types
                               :allow-null
                               ,allow-null))

(defun deconstruct (address types &key allow-null)
  (let ((pointer (cffi:make-pointer address)))
    (if (cffi:null-pointer-p pointer)
        (unless allow-null
          (complain "Null pointer for record ~
```

```
                    which expected ~a"
                   types))
      (loop for type in types
            for index from 0
            collect
            (follow pointer index type)))))

(defun follow (pointer index type)
  (let ((address (cffi:mem-aref pointer 'uint
                                index)))
    (ecase (follow-type type)
      ((array)   (unpack address (cadr type)))
      ((boolean) (not (zerop address)))
      ((object)  (unbox address :type type))
      ((uint)    address)
      ((int)     (if (<= address
                         most-positive-fixnum)
                     address
                   (+ (logand address
                              most-positive-fixnum)
                      most-negative-fixnum)))
      ((record)  (apply 'deconstruct
                        address (cdr type)))
      ((ustring) (from-foreign-string
                  (make-string-pointer
                    address))))))
```

Note incidentally that the CLAUDE package shadows the symbol `ARRAY`.

One particular use of arrays is for reducing the stack switching overhead of making very many calls into or out of lisp, which can otherwise become significant. We strongly recommend that wherever possible the data for such calls should be packed into arrays and the interface desgined so as to enforce fewer calls across the language boundary. For example, Chart supports `chart_new_nodes()` rather than `chart_new_node()`[3].

5. FUNCTION CALLS

The macro `defun-external` defines the library's functional interface. It hides everything that really matters but the programmer never wants to see: argument unmarshalling, checking and result marshalling; foreign name translation; memory management; error handlers. It installs an entry point into lisp (as if by `cffi:defcallback`), arranging for the corresponding C-name to be exported from the library and thus made visible to the application programmer. By hacking into the `defcallback` form's macroexpansion it is able to wrap error handlers around argument deconstruction as well as the body of the function; if for example an argument of the wrong type shows up then CLAUDE can explain the problem to the application:

```
>>> chart.describe_nodes([g])
Traceback (most recent call last):
  [...]
  File "pyChart\invoke.py", line 63, in invoker
    check(func, ctypes.byref(pointer), *args)
  File "pyChart\invoke.py", line 54, in check
    raise ChartError()
pyChart.invoke.ChartError: #<Chart Graph handle=
0x20053748> is a graph, but a node was expected.
>>>
```

The types of all arguments and the result (if any) must be specified, to allow for automatic conversion on the way in and out of the function. For example, this function:

```
(defun-external (display-graph
                 :result-type (object
                               :allow-null t))
    ((display display))
  (chart:display-graph display))
```

takes a display and returns a CLAUDE object, possibly null; and this one:

```
(defun-external set-node-locations
    ((display display)
     (nodes-and-locations (array
                           (record
                            (node
                             (record
                              (int int)
                              :allow-null t))))))
  (loop for (node location) in nodes-and-locations
        do
        (setf (chart:node-location node display)
              (when location
                (apply 'chart:make-location
                       location)))))
```

takes a display and a sequence of (node location) pairs, where the location is either a pair of integers or null, and the function does not return a result.

Passing a variable number of arguments across a foreign function interface can get ugly and we've chosen not to support it. The arguments to an external function are always fixed; the library is free to accept nulls for arguments whose specification was in some sense optional; and an array argument can have variable length.

By default each external call runs in its own thread. This allows for a last-ditch defense against errors in the library (or CLAUDE, or for that matter the lisp implementation): if the thread dies then the call is necessarily over but the program is not[4]. This precaution might not seem very important once the library is believed to be working properly, but during development it's invaluable.

[3] A simple experiment indicates that if `chart_new_node()` were made available, it would generate 1000 nodes almost an order of magnitude slower than its array counterpart.

[4] LispWorks executes all incoming calls in its "thread created by foreign code"; if this thread is killed (say by an unguarded `"Quit process"` restart) then no further incoming calls can be handled in that lisp session.

```
(let ((result +error+)
      (name (format nil "Safe call to ~s"
                    function)))
  (flet ((safe-run ()
           (when (with-simple-restart
                     (abort "Abandon call to ~a"
                            function)
                   (let ((*debugger-hook*
                          'debugger-hook))
                     (apply function arguments))
                   t)
             (setf result +success+))))
    (bt:join-thread (bt:make-thread #'safe-run
                                    :name name))
    result))
```

Function calls from the CLAUDE library into the application are referred to as *callbacks* (we're describing life from the application writer's perspective). Their uses range from notification of mouse events in a graphical toolkit to the handling of asynchronous errors. In lisp they are equivalent to invocations of `cffi:foreign-funcall-pointer`; LispWorks has a declarative macro `fli:define-foreign-funcallable` for handling this, a call to which is assembled and explicitly compiled first time a callback is invoked.

Callbacks are established and removed by calls to `claude_set_callbacks()`; they can be associated with a specific object and/or with none at all.

```
@ctypes.WINFUNCTYPE(ctypes.c_void_p,
                    ctypes.c_uint,
                    ctypes.c_uint)
def advise_condition(foreignRef, report):
    lib.raise_error(report)

set_callbacks(None, (("claude_advise_condition",
                      advise_condition),))
```

If the library wishes to invoke a callback on an object and none is currently established – either on that object or non-specifically – then nothing happens (i.e. it's not an error). It's assumed that the application programmer doesn't care to be informed about this particular event.

```
(invoke-callback
 '(:void
   (display (display :allow-null t))
   (address uint))
 display
 'advise-condition
 (when (and display (object-wrapper display))
   display)
 report)
```

6. ERROR HANDLING

The functions which CLAUDE exports all return a success/fail code (and so any "result" which functions wish to communicate must be via a pointer argument for the application to dereference).

```
enum {
  CLAUDE_RES_OK   =  0,        /* success */
  CLAUDE_RES_FAIL = -1         /* failure */
};
```

If the application spots a fail code, it can call `claude_last_error()` to find out more.

```
(defun-external (last-error :result-type ustring)
    ()
  (shift-last-error nil))

(defun debugger-hook (condition encapsulation)
  (declare (ignore encapsulation))
  (shift-last-error condition)
  (abort))

(defun shift-last-error (condition)
  (shiftf *last-error*
          (when condition
            (with-output-to-string (report)
              (report condition report)))))
```

There are methods on `report` for handling errors, warnings, and *complaints* (application pilot errors spotted by the library). The method on errors produces a full lisp backtrace; the library can choose whether to pass the whole thing on or to chop it (after the first line, say); if the application does anything intelligent with this string and its end users are sufficiently well trained then the backtrace could even be forwarded to the library writer for assistance. CLAUDE contains an elementary patch loader (which has been described elsewhere[5]) and so field patching of the lisp library is generally possible. Note that in LispWorks, at least, it is not possible to export new functions from the library without resaving the image, and so the introduction of additional `defun-external` forms is one thing that cannot be patched.

An asynchronous error can't be handled as above: the option of returning a `CLAUDE_RES_FAIL` isn't available when we're not actually inside a `defun-external` call. So it's advised that all threads which the library creates should install an error handler which invokes the `claude_advise_condition` callback, and that the application should handle this callback. When the application is notified of an asynchronous error, it can either deal with this directly, or make a synchronous call to `claude_raise_error()` and allow the usual error handlers to take over:

```
(defun-external raise-error ((report ustring))
  (shift-last-error report)
  (abort))
```

7. THE EXPORT PROCESS

After downloading the toolkit, the first step in exporting a library is to call `claude-setup:configure`, supplying the library's name and a working directory. For example, with a library called `wombat`, the following files and subdirectories are established:

[5]See `http://www.nicklevine.org/play/patching-made-easy.html`

```
wombat/
  build/save-dll.lisp
  claude/
    claude.asd
    src/...
  examples/C/...
  include/wombat.h
  manual/...
  pyWombat/...
  src/defsys.lisp
  wombat.asd
```

Even though no library is yet present we can easily try out the vanilla framework, by using the build script `save-dll.lisp` to save a DLL and then running some tests through the Python interface:

```
C:\home\wombat> lisp -init save-dll.lisp
  [...]                          ; saves wombat.dll
C:\home\wombat> python
Python 2.7.1
>>> from pyWombat import wombat
>>> wombat.Wombat()
<Wombat Wombat handle=0x200538b0>
>>> wombat.objects.communications_test()
True
>>>
```

Source files defining the library's external interface can later be added to the `src/` directory; these files along with dependencies for loading the library itself should be listed in the LispWorks `defsys` file (or, for those who'd rather, in the equivalent `.asd` file; the build script should then be set to load this instead). As each external class or function appears in lisp, its counterpart should be added to the `pyWombat` package. Whether or not the library is being targetted at Python application writers, Python is a reasonable test environment for it. Only once features are known to be working is it worth adding them to the C header and test files and finally to the manual.

The library, and in particular its CLAUDE layer, should be developed using lisp images saved to include a full lisp IDE (for instance, the LispWorks environment, or SLIME). CLAUDE supports this, via command-line arguments supplied when `save-dll.lisp` is loaded. With an IDE in place all the standard lisp development tools are immediately available. In particular it's straightforward to edit and recompile library code; depending on the framework being used to test the library, it may (Python) or clearly will not (C) be possible to interactively update the calling code without frequent halts. As already noted, one other cause for closing down and resaving the lisp image is addition of exported functions (new calls to `defun-external`).

8. ANOTHER EXAMPLE

Let's see what's involved in exporting another library: one not written by the present author. We choose Edi Weitz's ".NET layer for Common Lisp": *RDNZL*. We don't have space for the whole thing here, so we pick the following short example, for accessing web pages, out of RDNZL documentation and work on exporting that.

```
(import-types "System" "Net.WebClient")

(defun download-url (url)
  (let ((web-client (new "System.Net.WebClient")))
    [GetString (new "System.Text.ASCIIEncoding")
               [DownloadData web-client url]]))
```

Note the reader-macro on `#\[`, which case-preserves the next symbol and expands into a call to `rdnzl:invoke`. Two minor problems are presented, and we'll have to work around them:

- the function `rdnzl:new` returns an instance of the *structure* type `rdnzl:container`; and
- neither the argument types of `invoke`'s variable number of arguments nor that of its return value are preordained.

Let's deal with the RDNZL structure instances first, wrapping externalised objects around them[6]. In general `new` takes a variable number of arguments, but that feature won't be needed for this cut-down example.

```
(defclass-external container ()
  ((container :reader container-container
              :initarg :container)))

(defun contain (container)
  (make-instance 'container :container container))

(defun-external (new :result-type container)
  ((type ustring))
  (contain (rdnzl:new type)))
```

The next two functions are straightforward. Each RDNZL container has a *type-name* which is used in its `print-object` method, and it's worth exporting that; `rdnzl:import-types` takes a list of strings and doesn't return a value.

```
(defun-external import-types
  ((types (array ustring)))
  (apply 'rdnzl:import-types types))

(defun-external (type-name :result-type ustring)
  ((container object))
  (rdnzl::get-type-name
   (container-container container)))
```

The only real annoyance is `rdnzl:invoke`. Its first two arguments are a (RDNZL) container and a string, but after that all bets are off. A simple (if inelegant) solution for handling this flexibility is to export a number of functions, one for each signature that's needed. Here are the two we want. We have to translate between RDNZL's structure objects and our externalised CLOS instances, but otherwise the going is clear.

[6]CLAUDE does support the direct export of structure-objects, via its `defstruct-external` macro, but using that here would involve modifying RDNZL source, so we'll investigate this other option.

```
(defun-external (invoke-container-string
                 :result-type container)
  ((container object)
   (method ustring)
   (arg ustring))
  (contain
   (rdnzl:invoke (container-container container)
                 method arg)))

(defun-external (invoke-string-container
                 :result-type ustring)
  ((container object)
   (method ustring)
   (arg container))
  (rdnzl:invoke (container-container container)
                method
                (container-container arg)))
```

Now for the Python. We start by writing a stub for simplifying access to each of the external functions, for instance:

```
def new(type):
    return lib.rdnzl_new(type)
```

We can then implement a higher-level interface. Omitting here some of the details (string conversion functions; and a `__repr__` method which uses `rdnzl_type_name()` to generate printed representations), we have:

```
def import_types(types):
    c_strings = map(c_string, types)
    lib.import_types(objects.pack(c_strings))

class Container(objects.RdnzlObject):
    def __init__(self, type=None, handle=None):
        if handle is None:
            handle = lib.new(type)
        objects.RdnzlObject.init(self, handle)

    def invoke_container_string(self, method, arg):
        invoke = lib.invoke_container_string
        handle = invoke(self.box(),
                        c_string(method),
                        c_string(arg))
        return Container(handle=handle)
```

and similarly for `invoke_string_container()`.

Finally, we can try it all out, stepping our way through the `download-url` function with which we started:

```
>>> from pyRdnzl import rdnzl
>>> rdnzl.import_types(["System", "Net.WebClient"])
>>> c=rdnzl.Container("System.Net.WebClient")
>>> d=c.invoke_container_string("DownloadData",
      "http://nanook.agharta.de/")
>>> e=rdnzl.Container("System.Text.ASCIIEncoding")
>>> (c,d,e)
(<Rdnzl Container System.Net.WebClient
      handle=0x20053748>,
 <Rdnzl Container System.Byte[] handle=0x20053838>,
 <Rdnzl Container System.Text.ASCIIEncoding
      handle=0x200538b0>)
>>> print e.invoke_string_container("GetString", d)
<html>
  <head>
    <meta http-equiv="refresh"
          content="0;url=http://weitz.de/">
  </head>
  <body>
  </body>
</html>
>>> rdnzl.objects.remove_objects([c,d,e])
>>>
```

9. CONCLUDING REMARKS

This toolkit has only been tried on one lisp implementation and exercised by one individual. Indeed it came about as a quite unforseen consequence of publishing Chart and it's clearly yet to meet its full general potential.

Preparing a Common Lisp library for use in the outside world means abandoning some of CL's grace: no `with-interesting-state` macro can cross the divide; interfaces which depend on any number of keyword arguments will have to be rethought, as will multiple return values, closures, and undoubtedly various other features among CL's great strengths. If these restrictions are not excessive then libraries can be exported in exchange for very reasonable effort. CLAUDE is an abstraction layer whose task is to simplify that effort.

Our audience can be expanded.

ASDF 3, or Why Lisp is Now an Acceptable Scripting Language

François-René Rideau
Google
tunes@google.com

Abstract

ASDF, the *de facto* standard build system for Common Lisp, has been vastly improved between 2012 and 2014. These and other improvements finally bring Common Lisp up to par with "scripting languages" in terms of ease of writing and *deploying* portable code that can access and "glue" together functionality from the underlying system or external programs. "Scripts" can thus be written in Common Lisp, and take advantage of its expressive power, well-defined semantics, and efficient implementations. We describe the most salient improvements in ASDF 3 and how they enable previously difficult and portably impossible uses of the programming language. We discuss past and future challenges in improving this key piece of software infrastructure, and what approaches did or didn't work in bringing change to the Common Lisp community.

Introduction

As of 2013, one can use Common Lisp (CL) to *portably* write the programs for which one traditionally uses so-called "scripting" languages: one can write small scripts that glue together functionality provided by the operating system (OS), external programs, C libraries, or network services; one can scale them into large, maintainable and modular systems; and one can make those new services available to other programs via the command-line as well as via network protocols, etc.

The last barrier to making that possible was the lack of a portable way to build and deploy code so a same script can run *unmodified* for many users on one or many machines using one or many different compilers. This was solved by ASDF 3.

ASDF has been the *de facto* standard build system for portable CL software since shortly after its release by Dan Barlow in 2002 (Barlow 2004). **The purpose of a build system is to enable division of labor in software development**: source code is organized in separately-developed components that depend on other components, and the build system transforms the transitive closure of these components into a working program.

ASDF 3 is the latest rewrite of the system. Aside from fixing numerous bugs, it sports a new portability layer. One can now use ASDF to write Lisp programs that may be invoked from the command line or may spawn external programs and capture their output ASDF can deliver these programs as standalone executable files; moreover the companion script `cl-launch` (see section 2.9) can create light-weight scripts that can be run unmodified on many different kinds of machines, each differently configured. These features make portable scripting possible. Previously, key parts of a program had to be configured to match one's specific CL implementation, OS, and software installation paths. Now, all of one's usual scripting needs can be entirely fulfilled using CL, benefitting from its efficient implementations, hundreds of software libraries, etc.

In this article, we discuss how the innovations in ASDF 3 enable new kinds of software development in CL. In section 1, we explain what ASDF is about; we compare it to common practice in the C world. In section 2, we describe the improvements introduced in ASDF 3 and ASDF 3.1 to solve the problem of software delivery; this section requires some familiarity with CL. In section 3, we discuss the challenges of evolving a piece of community software, concluding with lessons learned from our experience.

This is the short version of this article. It sometimes refers to appendices present only in the extended version (Rideau 2014), that also includes a few additional examples and footnotes.

1. What ASDF is

1.1 ASDF: Basic Concepts

1.1.1 Components

ASDF is a build system for CL: it helps developers divide software into a hierarchy of *component*s and automatically generates a working program from all the source code.

Top components are called *system*s in an age-old Lisp tradition, while the bottom ones are source *file*s, typically written in CL. In between, there may be a recursive hierarchy of *module*s.

Users may then `operate` on these components with various build *operation*s, most prominently compiling the source code (operation `compile-op`) and loading the output into the current Lisp image (operation `load-op`).

Several related systems may be developed together in the same source code *project*. Each system may depend on code from other systems, either from the same project or from a different project. ASDF itself has no notion of projects, but other tools on top of ASDF do: Quicklisp (Beane 2011) packages together systems from a project into a *release*, and provides hundreds of releases as a *distribution*, automatically downloading on demand required systems and all their transitive dependencies.

Further, each component may explicitly declare a *dependency* on other components: whenever compiling or loading a component relies on declarations or definitions of packages, macros, variables, classes, functions, etc., present in another component, the programmer must declare that the former component *depends-on* the latter.

1.1.2 Example System Definition

Below is how the `fare-quasiquote` system is defined (with elisions) in a file `fare-quasiquote.asd`. It contains three files, `packages`, `quasiquote` and `pp-quasiquote` (the `.lisp` suffix is automatically added based on the component class; see Appendix C). The latter files each depend on the first file, because this former file defines the CL packages[1]:

[1] Packages are namespaces that contain symbols; they need to be created before the symbols they contain may even be read as valid syntax.

```
(defsystem "fare-quasiquote" ...
  :depends-on ("fare-utils")
  :components
  ((:file "packages")
   (:file "quasiquote"
    :depends-on ("packages"))
   (:file "pp-quasiquote"
    :depends-on ("quasiquote"))))
```

Among the elided elements were metadata such as `:license "MIT"`, and extra dependency information `:in-order-to ((test-op (test-op "fare-quasiquote-test")))`, that delegates testing the current system to running tests on another system. Notice how the system itself *depends-on* another system, `fare-utils`, a collection of utility functions and macros from another project, whereas testing is specified to be done by `fare-quasiquote-test`, a system defined in a different file, `fare-quasiquote-test.asd`, within the same project.

1.1.3 Action Graph

The process of building software is modeled as a Directed Acyclic Graph (DAG) of *action*s, where *each action is a pair of an operation and a component.* The DAG defines a partial order, whereby each action must be *perform*ed, but only after all the actions it (transitively) *depends-on* have already been performed.

For instance, in `fare-quasiquote` above, the *loading* of (the output of compiling) `quasiquote` *depends-on* the *compiling* of `quasiquote`, which itself depends-on the *loading* of (the output of compiling) `package`, etc.

Importantly, though, this graph is distinct from the preceding graph of components: the graph of actions isn't a mere refinement of the graph of components but a transformation of it that also incorporates crucial information about the structure of operations.

`ASDF` extracts from this DAG a *plan*, which by default is a topologically sorted list of actions, that it then *perform*s in order, in a design inspired by Pitman (Pitman 1984).

Users can extend `ASDF` by defining new subclasses of `operation` and/or `component` and the methods that use them, or by using global, per-system, or per-component hooks.

1.1.4 In-image

`ASDF` is an "in-image" build system, in the Lisp `defsystem` tradition: it compiles (if necessary) and loads software into the current CL image, and can later update the current image by recompiling and reloading the components that have changed. For better and worse, this notably differs from common practice in most other languages, where the build system is a completely different piece of software running in a separate process.[2] On the one hand, it minimizes overhead to writing build system extensions. On the other hand, it puts great pressure on `ASDF` to remain minimal.

Qualitatively, `ASDF` must be delivered as a single source file and cannot use any external library, since it itself defines the code that may load other files and libraries. Quantitatively, `ASDF` must minimize its memory footprint, since it's present in all programs that are built, and any resource spent is paid by each program.

For all these reasons, `ASDF` follows the minimalist principle that **anything that can be provided as an extension should be provided as an extension and left out of the core**. Thus it cannot afford to support a persistence cache indexed by the cryptographic digest of build expressions, or a distributed network of workers, etc. However, these could conceivably be implemented as `ASDF` extensions.

[2] Of course, a build system could compile CL code in separate processes, for the sake of determinism and parallelism: our XCVB did (Brody 2009); so does the Google build system.

1.2 Comparison to C programming practice

Most programmers are familiar with C, but not with CL. It's therefore worth contrasting `ASDF` to the tools commonly used by C programmers to provide similar services. Note though how these services are factored in very different ways in CL and in C.

To build and load software, C programmers commonly use `make` to build the software and `ld.so` to load it. Additionally, they use a tool like `autoconf` to locate available libraries and identify their features. In many ways these C solutions are better engineered than `ASDF`. But in other important ways `ASDF` demonstrates how these C systems have much accidental complexity that CL does away with thanks to better architecture.

- Lisp makes the full power of runtime available at compile-time, so it's easy to implement a Domain-Specific Language (DSL): the programmer only needs to define new functionality, as an extension that is then seamlessly combined with the rest of the language, including other extensions. In C, the many utilities that need a DSL must grow it onerously from scratch; since the domain expert is seldom also a language expert with resources to do it right, this means plenty of mutually incompatible, misdesigned, power-starved, misimplemented languages that have to be combined through an unprincipled chaos of expensive yet inexpressive means of communication.
- Lisp provides full introspection at runtime and compile-time alike, as well as a protocol to declare *features* and conditionally include or omit code or data based on them. Therefore you don't need dark magic at compile-time to detect available features. In C, people resort to horribly unmaintainable configuration scripts in a hodge podge of shell script, `m4` macros, C preprocessing and C code, plus often bits of `python`, `perl`, `sed`, etc.
- `ASDF` possesses a standard and standardly extensible way to configure where to find the libraries your code depends on, further improved in `ASDF` 2. In C, there are tens of incompatible ways to do it, between `libtool`, `autoconf`, `kde-config`, `pkg-config`, various manual `./configure` scripts, and countless other protocols, so that each new piece of software requires the user to learn a new *ad hoc* configuration method, making it an expensive endeavor to use or distribute libraries.
- `ASDF` uses the very same mechanism to configure both runtime and compile-time, so there is only one configuration mechanism to learn and to use, and minimal discrepancy.[3] In C, completely different, incompatible mechanisms are used at runtime (`ld.so`) and compile-time (unspecified), which makes it hard to match source code, compilation headers, static and dynamic libraries, requiring complex "software distribution" infrastructures (that admittedly also manage versioning, downloading and precompiling); this at times causes subtle bugs when discrepancies creep in.

Nevertheless, there are also many ways in which `ASDF` pales in comparison to other build systems for CL, C, Java, or other systems:

- `ASDF` isn't a general-purpose build system. Its relative simplicity is directly related to it being custom made to build CL software only. Seen one way, it's a sign of how little you can get away with if you have a good basic architecture; a similarly simple solution isn't available to most other programming languages, that require much more complex tools to achieve a similar purpose. Seen another way, it's also the CL community

[3] There is still discrepancy *inherent* with these times being distinct: the installation or indeed the machine may have changed.

failing to embrace the outside world and provide solutions with enough generality to solve more complex problems.

- At the other extreme, a build system for CL could have been made that is much simpler and more elegant than ASDF, if it could have required software to follow some simple organization constraints, without much respect for legacy code. A constructive proof of that is quick-build (Bridgewater 2012), being a fraction of the size of ASDF, itself a fraction of the size of ASDF 3, and with a fraction of the bugs — but none of the generality and extensibility (See section 2.10).
- ASDF it isn't geared at all to build large software in modern adversarial multi-user, multi-processor, distributed environments where source code comes in many divergent versions and in many configurations. It is rooted in an age-old model of building software in-image, what's more in a traditional single-processor, single-machine environment with a friendly single user, a single coherent view of source code and a single target configuration. The new ASDF 3 design is consistent and general enough that it could conceivably be made to scale, but that would require a lot of work.

2. ASDF 3: A Mature Build

2.1 A Consistent, Extensible Model

Surprising as it may be to the CL programmers who used it daily, there was an essential bug at the heart of ASDF: it didn't even try to propagate timestamps from one action to the next. And yet it worked, mostly. The bug was present from the very first day in 2001, and even before in mk-defsystem since 1990 (Kantrowitz 1990), and it survived till December 2012, despite all our robustification efforts since 2009 (Goldman 2010). Fixing it required a complete rewrite of ASDF's core.

As a result, the object model of ASDF became at the same time more powerful, more robust, and simpler to explain. The dark magic of its traverse function is replaced by a well-documented algorithm. It's easier than before to extend ASDF, with fewer limitations and fewer pitfalls: users may control how their operations do or don't propagate along the component hierarchy. Thus, ASDF can now express arbitrary action graphs, and could conceivably be used in the future to build more than just CL programs.

The proof of a good design is in the ease of extending it. And in CL, extension doesn't require privileged access to the code base. We thus tested our design by adapting the most elaborate existing ASDF extensions to use it. The result was indeed cleaner, eliminating the previous need for overrides that redefined sizable chunks of the infrastructure. Chronologically, however, we consciously started this porting process in interaction with developing ASDF 3, thus ensuring ASDF 3 had all the extension hooks required to avoid redefinitions.

See the entire story in Appendix F.

2.2 Bundle Operations

Bundle operations create a single output file for an entire system or collection of systems. The most directly user-facing bundle operations are compile-bundle-op and load-bundle-op: the former bundles into a single compilation file all the individual outputs from the compile-op of each source file in a system; the latter loads the result of the former. Also lib-op links into a library all the object files in a system and dll-op creates a dynamically loadable library out of them. The above bundle operations also have so-called *monolithic* variants that bundle all the files in a system *and all its transitive dependencies*.

Bundle operations make delivery of code much easier. They were initially introduced as asdf-ecl, an extension to ASDF specific to the implementation ECL, back in the day of ASDF 1. asdf-ecl was distributed with ASDF 2, though in a way that made upgrade slightly awkward to ECL users, who had to explicitly reload it after upgrading ASDF, even though it was included by the initial (require "asdf"). In 2012, it was generalized to other implementations as the external system asdf-bundle. It was then merged into ASDF during the development of ASDF 3: not only did it provide useful new operations, but the way that ASDF 3 was automatically upgrading itself for safety purposes (see Appendix B) would otherwise have broken things badly for ECL users if the bundle support weren't itself bundled with ASDF.

In ASDF 3.1, using deliver-asd-op, you can create both the bundle from compile-bundle-op and an .asd file to use to deliver the system in binary format only.

2.3 Understandable Internals

After bundle support was merged into ASDF (see section 2.2 above), it became trivial to implement a new concatenate-source-op operation. Thus ASDF could be developed as multiple files, which would improve maintainability. For delivery purpose, the source files would be concatenated in correct dependency order, into the single file asdf.lisp required for bootstrapping.

The division of ASDF into smaller, more intelligible pieces had been proposed shortly after we took over ASDF; but we had rejected the proposal then on the basis that ASDF must not depend on external tools to upgrade itself from source, another strong requirement (see Appendix B). With concatenate-source-op, an external tool wasn't needed for delivery and regular upgrade, only for bootstrap. Meanwhile this division had also become more important, since ASDF had grown so much, having almost tripled in size since those days, and was promising to grow some more. It was hard to navigate that one big file, even for the maintainer, and probably impossible for newcomers to wrap their head around it.

To bring some principle to this division, we followed the principle of one file, one package, as demonstrated by faslpath (Etter 2009) and quick-build (Bridgewater 2012), though not yet actively supported by ASDF itself (see section 2.10). This programming style ensures that files are indeed providing related functionality, only have explicit dependencies on other files, and don't have any forward dependencies without special declarations. Indeed, this was a great success in making ASDF understandable, if not by newcomers, at least by the maintainer himself; this in turn triggered a series of enhancements that would not otherwise have been obvious or obviously correct, illustrating the principle that **good code is code you can understand, organized in chunks you can each fit in your brain**.

2.4 Package Upgrade

Preserving the hot upgradability of ASDF was always a strong requirement (see Appendix B). In the presence of this package refactoring, this meant the development of a variant of CL's defpackage that plays nice with hot upgrade: define-package. Whereas the former isn't guaranteed to work and may signal an error when a package is redefined in incompatible ways, the latter will update an old package to match the new desired definition while recycling existing symbols from that and other packages.

Thus, in addition to the regular clauses from defpackage, define-package accepts a clause :recycle: it attempts to recycle each declared symbol from each of the specified packages in the given order. For idempotence, the package itself must be the first in the list. For upgrading from an old ASDF, the :asdf package is always named last. The default recycle list consists in a list of the package and its nicknames.

New features also include :mix and :reexport. :mix mixes imported symbols from several packages: when multiple

packages export symbols with the same name, the conflict is automatically resolved in favor of the package named earliest, whereas an error condition is raised when using the standard `:use` clause. `:reexport` reexports the same symbols as imported from given packages, and/or exports instead the same-named symbols that shadow them. ASDF 3.1 adds `:mix-reexport` and `:use-reexport`, which combine `:reexport` with `:mix` or `:use` in a single statement, which is more maintainable than repeating a list of packages.

2.5 Portability Layer

Splitting ASDF into many files revealed that a large fraction of it was already devoted to general purpose utilities. This fraction only grew under the following pressures: a lot of opportunities for improvement became obvious after dividing ASDF into many files; features added or merged in from previous extensions and libraries required new general-purpose utilities; as more tests were added for new features, and were run on all supported implementations, on multiple operating systems, new portability issues cropped up that required development of robust and portable abstractions.

The portability layer, after it was fully documented, ended up being slightly bigger than the rest of ASDF. Long before that point, ASDF was thus formally divided in two: this portability layer, and the `defsystem` itself. The portability layer was initially dubbed `asdf-driver`, because of merging in a lot of functionality from `xcvb-driver`. Because users demanded a shorter name that didn't include ASDF, yet would somehow be remindful of ASDF, it was eventually renamed UIOP: the Utilities for Implementation- and OS- Portability[4] It was made available separately from ASDF as a portability library to be used on its own; yet since ASDF still needed to be delivered as a single file `asdf.lisp`, UIOP was *transcluded* inside that file, now built using the `monolithic-concatenate-source-op` operation. At Google, the build system actually uses UIOP for portability without the rest of ASDF; this led to UIOP improvements that will be released with ASDF 3.1.1.

Most of the utilities deal with providing sane pathname abstractions (see Appendix C), filesystem access, sane input/output (including temporary files), basic operating system interaction — many things for which the CL standard lacks. There is also an abstraction layer over the less-compatible legacy implementations, a set of general-purpose utilities, and a common core for the ASDF configuration DSLs.[5] Importantly for a build system, there are portable abstractions for compiling CL files while controlling all the warnings and errors that can occur, and there is support for the life-cycle of a Lisp image: dumping and restoring images, initialization and finalization hooks, error handling, backtrace display, etc. However, the most complex piece turned out to be a portable implementation of `run-program`.

2.6 `run-program`

With ASDF 3, you can run external commands as follows:

```
(run-program '("cp" "-lax" "--parents"
               "src/foo" ,destination))
```

On Unix, this recursively hardlinks files in directory `src/foo` into a directory named by the string `destination`, preserving the prefix `src/foo`. You may have to add `:output t :error-output t` to get error messages on your `*standard-output*` and `*error-output*` streams, since the default value, `nil`, designates `/dev/null`. If the invoked program returns an error code, `run-program` signals a structured CL `error`, unless you specified `:ignore-error-status t`.

This utility is essential for ASDF extensions and CL code in general to portably execute arbitrary external programs. It was a challenge to write: Each implementation provided a different underlying mechanism with wildly different feature sets and countless corner cases. The better ones could fork and exec a process and control its standard-input, standard-output and error-output; lesser ones could only call the `system(3)` C library function. Moreover, Windows support differed significantly from Unix. ASDF 1 itself actually had a `run-shell-command`, initially copied over from `mk-defsystem`, but it was more of an attractive nuisance than a solution, despite our many bug fixes: it was implicitly calling `format`; capturing output was particularly contrived; and what shell would be used varied between implementations, even more so on Windows.

ASDF 3's `run-program` is full-featured, based on code originally from XCVB's `xcvb-driver` (Brody 2009). It abstracts away all these discrepancies to provide control over the program's standard output, using temporary files underneath if needed. Since ASDF 3.0.3, it can also control the standard input and error output. It accepts either a list of a program and arguments, or a shell command string. Thus your previous program could have been:

```
(run-program
 (format nil "cp -lax --parents src/foo ~S"
         (native-namestring destination))
 :output t :error-output t)
```

where (UIOP)'s `native-namestring` converts the `pathname` object `destination` into a name suitable for use by the operating system, as opposed to a CL `namestring` that might be escaped somehow.

You can also inject input and capture output:

```
(run-program '("tr" "a-z" "n-za-m")
 :input '("uryyb, jbeyq") :output :string)
```

returns the string `"hello, world"`. It also returns secondary and tertiary values `nil` and `0` respectively, for the (non-captured) error-output and the (successful) exit code.

`run-program` only provides a basic abstraction; a separate system `inferior-shell` was written on top of UIOP, and provides a richer interface, handling pipelines, `zsh` style redirections, splicing of strings and/or lists into the arguments, and implicit conversion of pathnames into native-namestrings, of symbols into downcased strings, of keywords into downcased strings with a `--` prefix. Its short-named functions `run`, `run/nil`, `run/s`, `run/ss`, respectively run the external command with outputs to the Lisp standard and error output, with no output, with output to a string, or with output to a stripped string. Thus you could get the same result as previously with:

```
(run/ss '(pipe (echo (uryyb ", " jbeyq))
               (tr a-z (n-z a-m))))
```

Or to get the number of processors on a Linux machine, you can:

```
(run '(grep -c "^processor.:"
           (< /proc/cpuinfo))
     :output #'read)
```

2.7 Configuration Management

ASDF always had minimal support for configuration management. ASDF 3 doesn't introduce radical change, but provides more usable replacements or improvements for old features.

[4] U, I, O and P are also the four letters that follow QWERTY on an anglo-saxon keyboard.

[5] ASDF 3.1 notably introduces a `nest` macro that nests arbitrarily many forms without indentation drifting ever to the right. It makes for more readable code without sacrificing good scoping discipline.

For instance, ASDF 1 had always supported version-checking: each component (usually, a system) could be given a version string with e.g. :version "3.1.0.97", and ASDF could be told to check that dependencies of at least a given version were used, as in :depends-on ((:version "inferior-shell" "2.0.0")). This feature can detect a dependency mismatch early, which saves users from having to figure out the hard way that they need to upgrade some libraries, and which.

Now, ASDF always required components to use "semantic versioning", where versions are strings made of dot-separated numbers like 3.1.0.97. But it didn't enforce it, leading to bad surprises for the users when the mechanism was expected to work, but failed. ASDF 3 issues a warning when it finds a version that doesn't follow the format. It would actually have issued an error, if that didn't break too many existing systems.

Another problem with version strings was that they had to be written as literals in the .asd file, unless that file took painful steps to extract it from another source file. While it was easy for source code to extract the version from the system definition, some authors legitimately wanted their code to not depend on ASDF itself. Also, it was a pain to repeat the literal version and/or the extraction code in every system definition in a project. ASDF 3 can thus extract version information from a file in the source tree, with, e.g. :version (:read-file-line "version.text") to read the version as the first line of file version.text. To read the third line, that would have been :version (:read-file-line "version.text" :at 2) (mind the off-by-one error in the English language). Or you could extract the version from source code. For instance, poiu.asd specifies :version (:read-file-form "poiu.lisp" :at (1 2 2)) which is the third subform of the third subform of the second form in the file poiu.lisp. The first form is an in-package and must be skipped. The second form is an (eval-when (...) body...) the body of which starts with a (defparameter *poiu-version* ...) form. ASDF 3 thus solves this version extraction problem for all software — except itself, since its own version has to be readable by ASDF 2 as well as by who views the single delivery file; thus its version information is maintained by a management script using regexps, of course written in CL.

Another painful configuration management issue with ASDF 1 and 2 was lack of a good way to conditionally include files depending on which implementation is used and what features it supports. One could always use CL reader conditionals such as #+(or sbcl clozure) but that means that ASDF could not even see the components being excluded, should some operation be invoked that involves printing or packaging the code rather than compiling it — or worse, should it involve cross-compilation for another implementation with a different feature set. There was an obscure way for a component to declare a dependency on a :feature, and annotate its enclosing module with :if-component-dep-fails :try-next to catch the failure and keep trying. But the implementation was a kluge in traverse that short-circuited the usual dependency propagation and had exponential worst case performance behavior when nesting such pseudo-dependencies to painfully emulate feature expressions.

ASDF 3 gets rid of :if-component-dep-fails: it didn't fit the fixed dependency model at all. A limited compatibility mode without nesting was preserved to keep processing old versions of SBCL. As a replacement, ASDF 3 introduces a new option :if-feature in component declarations, such that a component is only included in a build plan if the given feature expression is true during the planning phase. Thus a component annotated with :if-feature (:and :sbcl (:not :sb-unicode)) (and its children, if any) is only included on an SBCL without Unicode support. This is more expressive than what preceded, without requiring inconsistencies in the dependency model, and without pathological performance behavior.

2.8 Standalone Executables

One of the bundle operations contributed by the ECL team was program-op, that creates a standalone executable. As this was now part of ASDF 3, it was only natural to bring other ASDF-supported implementations up to par: CLISP, Clozure CL, CMUCL, LispWorks, SBCL, SCL. Thus UIOP features a dump-image function to dump the current heap image, except for ECL and its successors that follow a linking model and use a create-image function. These functions were based on code from xcvb-driver, which had taken them from cl-launch.

ASDF 3 also introduces a defsystem option to specify an entry point as e.g. :entry-point "my-package:entry-point". The specified function (designated as a string to be read after the package is created) is called without arguments after the program image is initialized; after doing its own initializations, it can explicitly consult *command-line-arguments*[6] or pass it as an argument to some main function.

Our experience with a large application server at ITA Software showed the importance of hooks so that various software components may modularly register finalization functions to be called before dumping the image, and initialization functions to be called before calling the entry point. Therefore, we added support for image life-cycle to UIOP. We also added basic support for running programs non-interactively as well as interactively: non-interactive programs exit with a backtrace and an error message repeated above and below the backtrace, instead of inflicting a debugger on end-users; any non-nil return value from the entry-point function is considered success and nil failure, with an appropriate program exit status.

Starting with ASDF 3.1, implementations that don't support standalone executables may still dump a heap image using the image-op operation, and a wrapper script, e.g. created by cl-launch, can invoke the program; delivery is then in two files instead of one. image-op can also be used by all implementations to create intermediate images in a staged build, or to provide ready-to-debug images for otherwise non-interactive applications.

2.9 cl-launch

Running Lisp code to portably create executable commands from Lisp is great, but there is a bootstrapping problem: when all you can assume is the Unix shell, how are you going to portably invoke the Lisp code that creates the initial executable to begin with?

We solved this problem some years ago with cl-launch. This bilingual program, both a portable shell script and a portable CL program, provides a nice colloquial shell command interface to building shell commands from Lisp code, and supports delivery as either portable shell scripts or self-contained precompiled executable files.

Its latest incarnation, cl-launch 4 (March 2014), was updated to take full advantage of ASDF 3. Its build specification interface was made more general, and its Unix integration was improved. You may thus invoke Lisp code from a Unix shell:

```
cl -sp lisp-stripper \
   -i "(print-loc-count \"asdf.lisp\")"
```

You can also use cl-launch as a script "interpreter", except that it invokes a Lisp compiler underneath:

[6] In CL, most variables are lexically visible and statically bound, but *special* variables are globally visible and dynamically bound. To avoid subtle mistakes, the latter are conventionally named with enclosing asterisks, also known in recent years as *earmuffs*.

```
#!/usr/bin/cl -sp lisp-stripper -E main
(defun main (argv)
  (if argv
      (map () 'print-loc-count argv)
      (print-loc-count *standard-input*)))
```

In the examples above, option `-sp`, shorthand for `--system-package`, simultaneously loads a system using ASDF during the build phase, and appropriately selects the current package; `-i`, shorthand for `--init` evaluates a form at the start of the execution phase; `-E`, shorthand for `--entry` configures a function that is called after init forms are evaluated, with the list of command-line arguments as its argument.[7] As for `lisp-stripper`, it's a simple library that counts lines of code after removing comments, blank lines, docstrings, and multiple lines in strings.

`cl-launch` automatically detects a CL implementation installed on your machine, with sensible defaults. You can easily override all defaults with a proper command-line option, a configuration file, or some installation-time configuration. See `cl-launch --more-help` for complete information. Note that `cl-launch` is on a bid to homestead the executable path `/usr/bin/cl` on Linux distributions; it may slightly more portably be invoked as `cl-launch`.

A nice use of `cl-launch` is to compare how various implementations evaluate some form, to see how portable it is in practice, whether the standard mandates a specific result or not:

```
for l in sbcl ccl clisp cmucl ecl abcl \
         scl allegro lispworks gcl xcl ; do
 cl -l $l -i \
 '(format t "'$l': ~S~%" `#5(1 ,@`(2 3)))' \
 2>&1 | grep "^$l:" # LW, GCL are verbose
done
```

`cl-launch` compiles all the files and systems that are specified, and keeps the compilation results in the same output-file cache as ASDF 3, nicely segregating them by implementation, version, ABI, etc. Therefore, the first time it sees a given file or system, or after they have been updated, there may be a startup delay while the compiler processes the files; but subsequent invocations will be faster as the compiled code is directly loaded. This is in sharp contrast with other "scripting" languages, that have to slowly interpret or recompile everytime. For security reasons, the cache isn't shared between users.

2.10 `package-inferred-system`

ASDF 3.1 introduces a new extension `package-inferred-system` that supports a one-file, one-package, one-system style of programming. This style was pioneered by `faslpath` (Etter 2009) and more recently `quick-build` (Bridgewater 2012). This extension is actually compatible with the latter but not the former, for ASDF 3.1 and `quick-build` use a slash `"/"` as a hierarchy separator where `faslpath` used a dot `"."`.

This style consists in every file starting with a `defpackage` or `define-package` form; from its `:use` and `:import-from` and similar clauses, the build system can identify a list of packages it depends on, then map the package names to the names of systems and/or other files, that need to be loaded first. Thus package name `lil/interface/all` refers to the file `interface/all.lisp` under the hierarchy registered by system `lil`, defined as follows in `lil.asd` as using class `package-inferred-system`:

```
(defsystem "lil" ...
  :description "LIL: Lisp Interface Library"
  :class :package-inferred-system
  :defsystem-depends-on ("asdf-package-system")
  :depends-on ("lil/interface/all"
               "lil/pure/all" ...)
  ...)
```

The `:defsystem-depends-on ("asdf-package-system")` is an external extension that provides backward compatibility with ASDF 3.0, and is part of Quicklisp. Because not all package names can be directly mapped back to a system name, you can register new mappings for `package-inferred-system`. The `lil.asd` file may thus contain forms such as:

```
(register-system-packages :closer-mop
 '(:c2mop :closer-common-lisp :c2cl ...))
```

Then, a file `interface/order.lisp` under the `lil` hierarchy, that defines abstract interfaces for order comparisons, starts with the following form, dependencies being trivially computed from the `:use` and `:mix` clauses:

```
(uiop:define-package :lil/interface/order
  (:use :closer-common-lisp
   :lil/interface/definition
   :lil/interface/base
   :lil/interface/eq :lil/interface/group)
  (:mix :fare-utils :uiop :alexandria)
  (:export ...))
```

This style provides many maintainability benefits: by imposing upon programmers a discipline of smaller namespaces, with explicit dependencies and especially explicit forward dependencies, the style encourages good factoring of the code into coherent units; by contrast, the traditional style of "everything in one package" has low overhead but doesn't scale very well. ASDF itself was rewritten in this style as part of ASDF 2.27, the initial ASDF 3 pre-release, with very positive results.

Since it depends on ASDF 3, `package-inferred-system` isn't as lightweight as `quick-build`, which is almost two orders of magnitude smaller than ASDF 3. But it does interoperate perfectly with the rest of ASDF, from which it inherits the many features, the portability, and the robustness.

2.11 Restoring Backward Compatibility

ASDF 3 had to break compatibility with ASDF 1 and 2: all operations used to be propagated *sideway* and *downward* along the component DAG (see Appendix F). In most cases this was undesired; indeed, ASDF 3 is predicated upon a new operation `prepare-op` that instead propagates *upward*.[8] Most existing ASDF extensions thus included workarounds and approximations to deal with the issue. But a handful of extensions did expect this behavior, and now they were broken.

Before the release of ASDF 3, authors of all known ASDF extensions distributed by Quicklisp had been contacted, to make their code compatible with the new fixed model. But there was no way to contact unidentified authors of proprietary extensions, beside sending an announcement to the mailing-list. Yet, whatever message was sent didn't attract enough attention. Even our co-maintainer Robert Goldman got bitten hard when an extension used at work stopped working, wasting days to figure out the issue.

Therefore, ASDF 3.1 features enhanced backward-compatibility. The class `operation` implements sideway and downward propagation on all classes that do not explicitly inherit from any

[7] Several systems are available to help you define an evaluator for your command-line argument DSL: `command-line-arguments`, `clon`, `lisp-gflags`.

[8] Sideway means the action of operation `o` on component `c` *depends-on* the action of `o` (or another operation) on each of the declared dependencies of `c`. Downward means that it *depends-on* the action of `o` on each of `c`'s children; upward, on `c`'s parent (enclosing module or system).

of the propagating mixins `downward-operation`, `upward-operation`, `sideway-operation` or `selfward-operation`, unless they explicitly inherit from the new mixin `non-propagating-operation`. `ASDF` 3.1 signals a `warning` at runtime when an operation class is instantiated that doesn't inherit from any of the above mixins, which will hopefully tip off authors of a proprietary extension that it's time to fix their code. To tell `ASDF` 3.1 that their operation class is up-to-date, extension authors may have to define their non-propagating operations as follows:

```
(defclass my-op (#+asdf3.1 non-propagating-
operation operation) ())
```

This is a case of "negative inheritance", a technique usually frowned upon, for the explicit purpose of backward compatibility. Now `ASDF` cannot use the CLOS Meta-Object Protocol (MOP), because it hasn't been standardized enough to be portably used without using an abstraction library such as `closer-mop`, yet `ASDF` cannot depend on any external library, and this is too small an issue to justify making a sizable MOP library part of `UIOP`. Therefore, the negative inheritance is implemented in an *ad hoc* way at runtime.

3. Code Evolution in a Conservative Community

3.1 Feature Creep? No, Mission Creep

Throughout the many features added and tenfold increase in size from `ASDF` 1 to `ASDF` 3, `ASDF` remained true to its minimalism — but the mission, relative to which the code remains minimal, was extended, several times: In the beginning, `ASDF` was the simplest extensible variant of `defsystem` that builds CL software (see Appendix A). With `ASDF` 2, it had to be upgradable, portable, modularly configurable, robust, performant, usable (see Appendix B). Then it had to be more declarative, more reliable, more predictable, and capable of supporting language extensions (see Appendix D). Now, `ASDF` 3 has to support a coherent model for representing dependencies, an alternative one-package-per-file style for declaring them, software delivery as either scripts or binaries, a documented portability layer including image life-cycle and external program invocation, etc. (see section 2).

3.2 Backward Compatibility is Social, not Technical

As efforts were made to improve `ASDF`, a constant constraint was that of *backward compatibility*: every new version of `ASDF` had to be compatible with the previous one, i.e. systems that were defined using previous versions had to keep working with new versions. But what more precisely is backward compatibility?

In an overly strict definition that precludes any change in behavior whatsoever, even the most uncontroversial bug fix isn't backward-compatible: any change, for the better as it may be, is incompatible, since by definition, some behavior has changed!

One might be tempted to weaken the constraint a bit, and define "backward compatible" as being the same as a "conservative extension": a *conservative extension* may fix erroneous situations, and give new meaning to situations that were previously undefined, but may not change the meaning of previously defined situations. Yet, this definition is doubly unsatisfactory. On the one hand, it precludes any amendment to previous bad decisions; hence, the jest **if it's not backwards, it's not compatible**. On the other hand, even if it only creates new situations that work correctly where they were previously in error, some existing analysis tool might assume these situations could never arise, and be confused when they now do.

Indeed this happened when `ASDF` 3 tried to better support *secondary systems*. `ASDF` looks up systems by name: if you try to load system `foo`, `ASDF` will search in registered directories for a file call `foo.asd`. Now, it was common practice that programmers may define multiple "secondary" systems in a same `.asd` file, such as a test system `foo-test` in addition to `foo`. This could lead to "interesting" situations when a file `foo-test.asd` existed, from a different, otherwise shadowed, version of the same library, resulting in a mismatch between the system and its tests. To make these situations less likely, `ASDF` 3 recommends that you name your secondary system `foo/test` instead of of `foo-test`, which should work just as well in `ASDF` 2, but with reduced risk of clash. Moreover, `ASDF` 3 can recognize the pattern and automatically load `foo.asd` when requested `foo/test`, in a way guaranteed not to clash with previous usage, since no directory could contain a file thus named in any modern operating system. In contrast, `ASDF` 2 has no way to automatically locate the `.asd` file from the name of a secondary system, and so you must ensure that you loaded the primary `.asd` file before you may use the secondary system. This feature may look like a textbook case of a backward-compatible "conservative extension". Yet, it's the major reason why Quicklisp itself still hasn't adopted `ASDF` 3: Quicklisp assumed it could always create a file named after each system, which happened to be true in practice (though not guaranteed) before this `ASDF` 3 innovation; systems that newly include secondary systems using this style break this assumption, and will require non-trivial work for Quicklisp to support.

What then, is backward compatibility? It isn't a technical constraint. **Backward compatibility is a social constraint**. The new version is backward compatible if the users are happy. This doesn't mean matching the previous version on all the mathematically conceivable inputs; it means improving the results for users on all the actual inputs they use; or providing them with alternate inputs they may use for improved results.

3.3 Weak Synchronization Requires Incremental Fixes

Even when some "incompatible" changes are not controversial, it's often necessary to provide temporary backward compatible solutions until all the users can migrate to the new design. Changing the semantics of one software system while other systems keep relying on it is akin to changing the wheels on a running car: you cannot usually change them all at once, at some point you must have both kinds active, and you cannot remove the old ones until you have stopped relying on them. Within a fast moving company, such migration of an entire code base can happen in a single checkin. If it's a large company with many teams, the migration can take many weeks or months. When the software is used by a weakly synchronized group like the CL community, the migration can take years.

When releasing `ASDF` 3, we spent a few months making sure that it would work with all publicly available systems. We had to fix many of these systems, but mostly, we were fixing `ASDF` 3 itself to be more compatible. Indeed, several intended changes had to be forsaken, that didn't have an incremental upgrade path, or for which it proved infeasible to fix all the clients.

A successful change was notably to modify the default encoding from the uncontrolled environment-dependent `:default` to the *de facto* standard `:utf-8`; this happened a year after adding support for encodings and `:utf-8` was added, and having forewarned community members of the future change in defaults, yet a few systems still had to be fixed (see Appendix D).

On the other hand, an unsuccessful change was the attempt to enable an innovative system to control warnings issued by the compiler. First, the `*uninteresting-conditions*` mechanism allows system builders to hush the warnings they know they don't care for, so that any compiler output is something they care for, and whatever they care for isn't drowned into a sea of uninteresting output. The mechanism itself is included in `ASDF` 3, but disabled by default, because there was no consensually agreeable value except an empty set, and no good way (so far) to configure it

both modularly and without pain. Second, another related mechanism that was similarly disabled is `deferred-warnings`, whereby `ASDF` can check warnings that are deferred by SBCL or other compilers until the end of the current *compilation-unit*. These warnings notably include forward references to functions and variables. In the previous versions of `ASDF`, these warnings were output at the end of the build the first time a file was built, but not checked, and not displayed afterward. If in `ASDF` 3 you `(uiop:enable-deferred-warnings)`, these warnings are displayed and checked every time a system is compiled or loaded. These checks help catch more bugs, but enabling them prevents the successful loading of a lot of systems in Quicklisp that have such bugs, even though the functionality for which these systems are required isn't affected by these bugs. Until there exists some configuration system that allows developers to run all these checks on new code without having them break old code, the feature will have to remain disabled by default.

3.4 Underspecification Creates Portability Landmines

The CL standard leaves many things underspecified about pathnames in an effort to define a useful subset common to many then-existing implementations and filesystems. However, the result is that portable programs can forever only access but a small subset of the complete required functionality. This result arguably makes the standard far less useful than expected (see Appendix C). The lesson is **don't standardize partially specified features**. It's better to standardize that some situations cause an error, and reserve any resolution to a later version of the standard (and then follow up on it), or to **delegate specification to other standards**, existing or future.

There could have been one pathname protocol per operating system, delegated to the underlying OS via a standard FFI. Libraries could then have sorted out portability over N operating systems. Instead, by standardizing only a common fragment and letting each of M implementations do whatever it can on each operating system, libraries now have to take into account N*M combinations of operating systems and implementations. In case of disagreement, it's much better to let each implementation's variant exist in its own, distinct namespace, which avoids any confusion, than have incompatible variants in the same namespace, causing clashes.

Interestingly, the aborted proposal for including `defsystem` in the CL standard was also of the kind that would have specified a minimal subset insufficient for large scale use while letting the rest underspecified. The CL community probably dodged a bullet thanks to the failure of this proposal.

3.5 Safety before Ubiquity

Guy Steele has been quoted as vaunting the programmability of Lisp's syntax by saying: *If you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases.* Unhappily, if he were speaking about CL specifically, he would have had to add: *but it can't be the same as anyone else's*.

Indeed, syntax in CL is controlled via a fuzzy set of global variables, prominently including the `*readtable*`. Making non-trivial modifications to the variables and/or tables is possible, but letting these modifications escape is a serious issue; for the author of a system has no control over which systems will or won't be loaded before or after his system — this depends on what the user requests and on what happens to already have been compiled or loaded. Therefore in absence of further convention, it's always a bug to either rely on the syntax tables having non-default values from previous systems, or to inflict non-default values upon next systems. What is worse, changing syntax is only useful if it also happens at the interactive REPL and in appropriate editor buffers. Yet these interactive syntax changes can affect files built interactively, including, upon modification, components that do not depend on the syntax support, or worse, that the syntax support depends on; this can cause catastrophic circular dependencies, and require a fresh start after having cleared the output file cache. Systems like `named-readtables` or `cl-syntax` help with syntax control, but proper hygiene isn't currently enforced by either CL or `ASDF`, and remains up to the user, especially at the REPL.

Build support is therefore strongly required for safe syntax modification; but this build support isn't there yet in `ASDF` 3. For backward-compatibility reasons, `ASDF` will not enforce strict controls on the syntax, at least not by default. But it is easy to enforce hygiene by binding read-only copies of the standard syntax tables around each action. A more backward-compatible behavior is to let systems modify a shared readtable, and leave the user responsible for ensuring that all modifications to this readtable used in a given image are mutually compatible; `ASDF` can still bind the current `*readtable*` to that shared readtable around every compilation, to at least ensure that selection of an incompatible readtable at the REPL does not pollute the build. A patch to this effect is pending acceptance by the new maintainer.

Until such issues are resolved, even though the Lisp ideal is one of ubiquitous syntax extension, and indeed extension through macros is ubiquitous, extension though reader changes are rare in the CL community. This is in contrast with other Lisp dialects, such as Racket, that have succeeded at making syntax customization both safe and ubiquitous, by having it be strictly scoped to the current file or REPL. **Any language feature has to be safe before it may be ubiquitous**; if the semantics of a feature depend on circumstances beyond the control of system authors, such as the bindings of syntax variables by the user at his REPL, then these authors cannot depend on this feature.

3.6 Final Lesson: Explain it

While writing this article, we had to revisit many concepts and pieces of code, which led to many bug fixes and refactorings to `ASDF` and `cl-launch`. An earlier interactive "`ASDF` walkthrough" via Google Hangout also led to enhancements. Our experience illustrates the principle that you should always **explain your programs**: having to intelligibly verbalize the concepts will make *you* understand them better.

Bibliography

Daniel Barlow. ASDF Manual. 2004. `http://common-lisp.net/project/asdf/`

Zach Beane. Quicklisp. 2011. `http://quicklisp.org/`

Alastair Bridgewater. Quick-build (private communication). 2012.

François-René Rideau and Spencer Brody. XCVB: an eXtensible Component Verifier and Builder for Common Lisp. 2009. `http://common-lisp.net/projects/xcvb/`

Peter von Etter. faslpath. 2009. `https://code.google.com/p/faslpath/`

François-René Rideau and Robert Goldman. Evolving ASDF: More Cooperation, Less Coordination. 2010. `http://common-lisp.net/project/asdf/doc/ilc2010draft.pdf`

Mark Kantrowitz. Defsystem: A Portable Make Facility for Common Lisp. 1990. `ftp://ftp.cs.rochester.edu/pub/archives/lisp-standards/defsystem/pd-code/mkant/defsystem.ps.gz`

Kent Pitman. The Description of Large Systems. 1984. `http://www.nhplace.com/kent/Papers/Large-Systems.html`

François-René Rideau. ASDF3, or Why Lisp is Now an Acceptable Scripting Language (extended version). 2014. `http://fare.tunes.org/files/asdf3/asdf3-2014.html`

Generalizers: New Metaobjects for Generalized Dispatch

Christophe Rhodes
Department of Computing
Goldsmiths, University of London
London SE14 6NW
c.rhodes@gold.ac.uk

Jan Moringen
Universität Bielefeld
Technische Fakultät
33594 Bielefeld
jmoringe@techfak.uni-bielefeld.de

David Lichteblau
ZenRobotics Ltd
Vilhonkatu 5 A
FI-00100 Helsinki
david@lichteblau.com

ABSTRACT

This paper introduces a new metaobject, the generalizer, which complements the existing specializer metaobject. With the help of examples, we show that this metaobject allows for the efficient implementation of complex non-class-based dispatch within the framework of existing metaobject protocols. We present our modifications to the generic function invocation protocol from the *Art of the Metaobject Protocol*; in combination with previous work, this produces a fully-functional extension of the existing mechanism for method selection and combination, including support for method combination completely independent from method selection. We discuss our implementation, within the SBCL implementation of Common Lisp, and in that context compare the performance of the new protocol with the standard one, demonstrating that the new protocol can be tolerably efficient.

Report-No.: `http://eprints.gold.ac.uk/id/eprint/9924`

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Object-oriented Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design

Keywords

generic functions, specialization-oriented programming, method selection, method combination

1. INTRODUCTION

The revisions to the original Common Lisp language [14] included the detailed specification of an object system, known as the Common Lisp Object System (CLOS), which was eventually standardized as part of the ANSI Common Lisp standard [10]. The object system as presented to the standardization committee was formed of three chapters. The first two chapters covered programmer interface concepts and the functions in the programmer interface [15, Chapter 28] and were largely incorporated into the final standard; the third chapter, covering a Metaobject Protocol (MOP) for CLOS, was not.

Nevertheless, the CLOS MOP continued to be developed, and the version documented in [7] has proven to be a reasonably robust design. While many implementations have derived their implementations of CLOS from either the Closette illustrative implementation in [7], or the Portable Common Loops implementation of CLOS from Xerox Parc, there have been largely from-scratch reimplementations of CLOS (in CLISP[1] and CCL[2], at least) incorporating substantial fractions of the Metaobject Protocol as described.

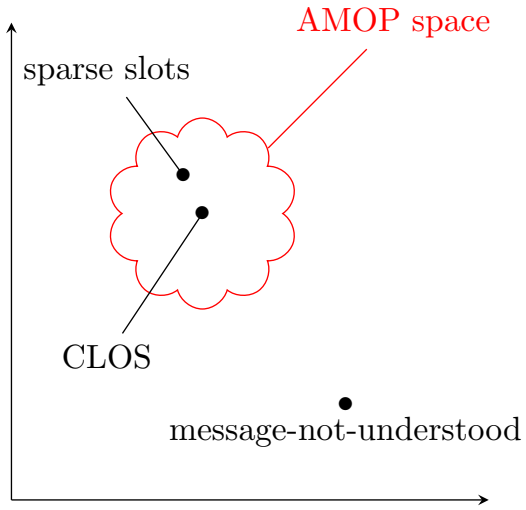


Figure 1: MOP Design Space

Although it has stood the test of time, the CLOS MOP is neither without issues (e.g. semantic problems with `make-method-lambda` [2]; useful functions such as `compute-effective-slot-definition-initargs` being missing from

[1]GNU CLISP, at `http://www.clisp.org/`
[2]Clozure Common Lisp, at `http://ccl.clozure.com/`

the standard) nor is it a complete framework for the metaprogrammer to implement all conceivable variations of object-oriented behaviour. While metaprogramming offers some possibilities for customization of the object system behaviour, those possibilities cannot extend arbitrarily in all directions (conceptually, if a given object system is a point in design space, then a MOP for that object system allows exploration of a region of design space around that point; see figure 1). In the case of the CLOS MOP, there is still an expectation that functionality is implemented with methods on generic functions, acting on objects with slots; it is not possible, for example, to transparently implement support for "message not understood" as in the message-passing paradigm, because the analogue of messages (generic functions) need to be defined before they are used.

Nevertheless, the MOP is flexible, and is used for a number of things, including: documentation generation (where introspection in the MOP is used to extract information from a running system); object-relational mapping and other approaches to object persistence; alternative backing stores for slots (hash-tables or symbols); and programmatic construction of metaobjects, for example for IDL compilers and model transformations.

One area of functionality where there is scope for customization by the metaprogrammer is in the mechanics and semantics of method applicability and dispatch. While in principle AMOP allows customization of dispatch in various different ways (the metaprogrammer can define methods on protocol functions such as `compute-applicable-methods`, `compute-applicable-methods-using-classes`), for example, in practice implementation support for this was weak until relatively recently[3].

Another potential mechanism for customizing dispatch is implicit in the class structure defined by AMOP: standard specializer objects (instances of `class` and `eql-specializer`) are generalized instances of the `specializer` protocol class, and in principle there are no restrictions on the metaprogrammer constructing additional subclasses. Previous work [9] has explored the potential for customizing generic function dispatch using extended specializers, but there the metaprogrammer must override the entirety of the generic function invocation protocol (from `compute-discriminating-function` on down), leading to toy implementations and duplicated effort.

This paper introduces a protocol for efficient and controlled handling of new subclasses of `specializer`. In particular, it introduces the `generalizer` protocol class, which generalizes the return value of `class-of` in method applicability computation, and allows the metaprogrammer to hook into cacheing schemes to avoid needless recomputation of effective methods for sufficiently similar generic function arguments (See Figure 2).

The remaining sections in this paper can be read in any order. We give some motivating examples in section 2, including reimplementations of examples from previous work,

[3]the Closer to MOP project, at `http://common-lisp.net/project/closer/`, attempts to harmonize the different implementations of the metaobject protocol in Common Lisp.

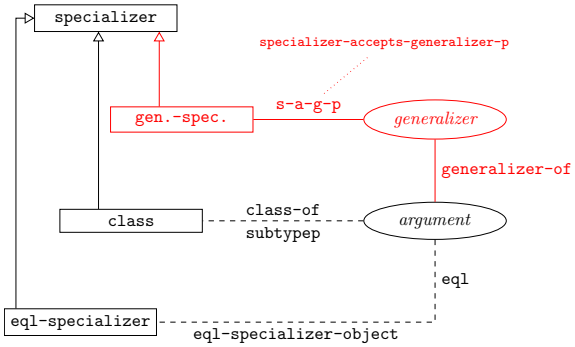


Figure 2: Dispatch Comparison

as well as examples which are poorly supported by previous protocols. We describe the protocol itself in section 3, describing each protocol function in detail and, where applicable, relating it to existing protocol functions within the CLOS MOP. We survey related work in more detail in section 4, touching on work on customized dispatch schemes in other environments. Finally, we draw our conclusions from this work, and indicate directions for further development, in section 5; reading that section before the others indicates substantial trust in the authors' work.

2. EXAMPLES

In this section, we present a number of examples of dispatch implemented using our protocol, which we describe in section 3. For reasons of space, the metaprogram code examples in this section do not include some of the necessary support code to run; complete implementations of each of these cases, along with the integration of this protocol into the SBCL implementation [11] of Common Lisp, are included in the authors' repository[4].

A note on terminology: we will attempt to distinguish between the user of an individual case of generalized dispatch (the "programmer"), the implementor of a particular case of generalized dispatch (the "metaprogrammer"), and the authors as the designers and implementors of our generalized dispatch protocol (the "metametaprogammer", or more likely "we").

2.1 CONS specializers

One motivation for the use of generalized dispatch is in an extensible code walker: a new special form can be handled simply by writing an additional method on the walking generic function, seamlessly interoperating with all existing methods. In this use-case, dispatch is performed on the first element of lists. Semantically, we allow the programmer to specialize any argument of methods with a new kind of specializer, `cons-specializer`, which is applicable if and only if the corresponding object is a `cons` whose `car` is `eql` to the symbol associated with the `cons-specializer`; these specializers are more specific than the `cons` class, but less specific than an `eql-specializer` on any given `cons`.

[4]the tag `els2014-submission` in `http://christophe.rhodes.io/git/specializable.git` corresponds to the code repository at the point of submitting this paper.

The programmer code using these specializers is unchanged from [9]; the benefits of the protocol described here are: that the separation of concerns is complete – method selection is independent of method combination – and that the protocol allows for efficient implementation where possible, even when method selection is customized. In an application such as walking source code, we would expect to encounter special forms (distinguished by particular atoms in the `car` position) multiple times, and hence to dispatch to the same effective method repeatedly. We discuss the efficiency aspects of the protocol in more detail in section 3.1.2; we present the metaprogrammer code to implement the `cons-specializer` below.

```
(defclass cons-specializer (specializer)
  ((%car :reader %car :initarg :car)))
(defclass cons-generalizer (generalizer)
  ((%car :reader %car :initarg :car)))
(defmethod generalizer-of-using-class
    ((gf cons-generic-function) arg)
  (typecase arg
    ((cons symbol)
     (make-instance 'cons-generalizer
                    :car (car arg)))
    (t (call-next-method))))
(defmethod generalizer-equal-hash-key
    ((gf cons-generic-function)
     (g cons-generalizer))
  (%car g))
(defmethod specializer-accepts-generalizer-p
    ((gf cons-generic-function)
     (s cons-specializer)
     (g cons-generalizer))
  (if (eql (%car s) (%car g))
      (values t t)
      (values nil t)))
(defmethod specializer-accepts-p
    ((s cons-specializer) o)
  (and (consp o) (eql (car o) (%car s))))
```

The code above shows a minimal use of our protocol. We have elided some support code for parsing and unparsing specializers, and for handling introspective functions such as finding generic functions for a given specializer. We have also elided methods on the protocol functions `specializer<` and `same-specializer-p`; for `cons-specializer` objects, specializer ordering is trivial, as only one `cons-specializer` (up to equality) can ever be applicable to any given argument. See section 2.3 for a case where specializer ordering is non-trivial.

As in [9], the programmer can use these specializers to implement a modular code walker, where they define one method per special operator. We show two of those methods below, in the context of a walker which checks for unused bindings and uses of unbound variables.

```
(defgeneric walk (form env stack)
  (:generic-function-class cons-generic-function))
(defmethod walk
    ((expr (cons lambda)) env call-stack)
  (let ((lambda-list (cadr expr))
        (body (cddr expr)))
    (with-checked-bindings
        ((bindings-from-ll lambda-list)
         env call-stack)
      (dolist (form body)
        (walk form env (cons form call-stack))))))
(defmethod walk
    ((expr (cons let)) env call-stack)
  (flet ((let-binding (x)
           (walk (cadr x) env
                 (cons (cadr x) call-stack))
           (cons (car x)
                 (make-instance 'binding))))
    (with-checked-bindings
        ((mapcar #'let-binding (cadr expr))
         env call-stack)
      (dolist (form (cddr expr))
        (walk form env (cons form call-stack))))))
```

Note that in this example there is no strict need for `cons-specializer` and `cons-generalizer` to be distinct classes. In standard generic function dispatch, the `class` functions both as the specializer for methods and as the generalizer for generic function arguments; we can think of the dispatch implemented by `cons-specializer` objects as providing for subclasses of the `cons` class distinguished by the `car` of the `cons`. This analogy also characterizes those use cases where the metaprogrammer could straightforwardly use filtered dispatch [3] to implement their dispatch semantics. We will see in section 2.3 an example of a case where filtered dispatch is incapable of straightforwardly expressing the dispatch, but first we present our implementation of the motivating case from [3].

2.2 SIGNUM specializers

Our second example of the implementation and use of generalized specializers is a reimplementation of one of the examples in [3]: specifically, the factorial function. Here, dispatch will be performed based on the `signum` of the argument, and again, at most one method with a `signum` specializer will be applicable to any given argument, which makes the structure of the specializer implementation very similar to the `cons` specializers in the previous section.

The metaprogrammer has chosen in the example below to compare signum values using `=`, which means that a method with specializer `(signum 1)` will be applicable to positive floating-point arguments (see the first method on `specializer-accepts-generalizer-p` and the method on `specializer-accepts-p` below). This leads to one subtle difference in behaviour compared to that of the `cons` specializers: in the case of `signum` specializers, the *next* method after any `signum` specializer can be different, depending on the class of the argument. This aspect of the dispatch is handled by the second method on `specializer-accepts-generalizer-p` below.

```
(defclass signum-specializer (specializer)
  ((%signum :reader %signum :initarg :signum)))
(defclass signum-generalizer (generalizer)
```

```
  ((%signum :reader %signum :initarg :signum)))
(defmethod generalizer-of-using-class
    ((gf signum-generic-function) (arg real))
  (make-instance 'signum-generalizer
                 :signum (signum arg)))
(defmethod generalizer-equal-hash-key
    ((gf signum-generic-function)
     (g signum-generalizer))
  (%signum g))
(defmethod specializer-accepts-generalizer-p
    ((gf signum-generic-function)
     (s signum-specializer)
     (g signum-generalizer))
  (if (= (%signum s) (%signum g))
      (values t t)
      (values nil t)))

(defmethod specializer-accepts-generalizer-p
    ((gf signum-generic-function)
     (s specializer)
     (g signum-generalizer))
  (specializer-accepts-generalizer-p
   gf s (class-of (%signum g))))

(defmethod specializer-accepts-p
    ((s signum-specializer) o)
  (and (realp o) (= (%signum s) (signum o))))
```

Given these definitions, and once again some more straightforward ones elided for reasons of space, the programmer can implement the factorial function as follows:

```
(defgeneric fact (n)
  (:generic-function-class signum-generic-function))
(defmethod fact ((n (signum 0))) 1)
(defmethod fact ((n (signum 1))) (* n (fact (1- n))))
```

The programmer does not need to include a method on `(signum -1)`, as the standard `no-applicable-method` protocol will automatically apply to negative real or non-real arguments.

2.3 Accept HTTP header specializers

In this section, we implement a non-trivial form of dispatch. The application in question is a web server, and specifically to allow the programmer to support RFC 2616 [5] content negotiation, of particular interest to publishers and consumers of REST-style Web APIs.

The basic mechanism in content negotiation is as follows: the web client sends an HTTP request with an `Accept` header, which is a string describing the media types it is willing to receive as a response to the request, along with numerical preferences. The web server compares these stated client preferences with the resources it has available to satisfy this request, and sends the best matching resource in its response.

For example, a graphical web browser might send an `Accept` header of `text/html,application/xml;q=0.9,*/*;q=0.8` for a request of a resource typed in to the URL bar. This should be interpreted as meaning that: if the server can provide content of type `text/html` (i.e. HTML) for that resource, then it should do so. Otherwise, if it can provide `application/xml` content (i.e. XML of any schema), then that should be provided; failing that, any other content type is acceptable.

In the case where there are static files on the filesystem, and the web server must merely select between them, there is not much more to say. However, it is not unusual for a web service to be backed by some other form of data, and responses computed and sent on the fly, and in these circumstances the web server must compute which of its known output formats it can use to satisfy the request before actually generating the best matching response. This can be modelled as one generic function responsible for generating the response, with methods corresponding to content-types – and the generic function must then perform method selection against the request's `Accept` header to compute the appropriate response.

The `accept-specializer` below implements this dispatch. It depends on a lazily-computed `tree` slot to represent the information in the accept header (generated by `parse-accept-string`), and a function `q` to compute the (defaulted) preference level for a given content-type and `tree`; then, method selection and ordering involves finding the `q` for each `accept-specializer`'s content type given the `tree`, and sorting them according to the preference level.

```
(defclass accept-specializer (specializer)
  ((media-type :initarg :media-type :reader media-type)))
(defclass accept-generalizer (generalizer)
  ((header :initarg :header :reader header)
   (tree)
   (next :initarg :next :reader next)))
(defmethod generalizer-equal-hash-key
    ((gf accept-generic-function)
     (g accept-generalizer))
  `(accept-generalizer ,(header g)))
(defmethod specializer-accepts-generalizer-p
    ((gf accept-generic-function)
     (s accept-specializer)
     (g accept-generalizer))
  (values (q (media-type s) (tree g)) t))
(defmethod specializer-accepts-generalizer-p
    ((gf accept-generic-function)
     (s specializer)
     (g accept-generalizer))
  (specializer-accepts-generalizer-p
   gf s (next g)))

(defmethod specializer<
    ((gf accept-generic-function)
     (s1 accept-specializer)
     (s2 accept-specializer)
     (g accept-generalizer))
  (let ((m1 (media-type s1))
        (m2 (media-type s2))
        (tree (tree g)))
    (cond
      ((string= m1 m2) '=)
      (t (let ((q1 (q m1 tree)))
```

```
      (q2 (q m2 tree))))
(cond
  ((= q1 q2) '=)
  ((< q1 q2) '>)
  (t '<))))))
```

The metaprogrammer can then add support for objects representing client requests, such as instances of the `request` class in the Hunchentoot[5] web server, by translating these into `accept-generalizer` instances. The code below implements this, by defining the computation of a `generalizer` object for a given request, and specifying how to compute whether the specializer accepts the given request object (`q` returns a number between 0 and 1 if any pattern in the `tree` matches the media type, and `nil` if the media type cannot be matched at all).

```
(defmethod generalizer-of-using-class
    ((gf accept-generic-function)
     (arg tbnl:request))
  (make-instance 'accept-generalizer
                 :header (tbnl:header-in :accept arg)
                 :next (call-next-method)))
(defmethod specializer-accepts-p
    ((s accept-specializer)
     (o tbnl:request))
  (let* ((accept (tbnl:header-in :accept o))
         (tree (parse-accept-string accept))
         (q (q (media-type s) tree)))
    (and q (> q 0))))
```

This dispatch cannot be implemented using filtered dispatch, except by generating anonymous classes with all the right mime-types as direct superclasses in dispatch order; the filter would generate

```
(ensure-class nil :direct-superclasses
 '(text/html image/webp ...))
```

and dispatch would operate using those anonymous classes. While this is possible to do, it is awkward to express content-type negotiation in this way, as it means that the dispatcher must know about the universe of mime-types that clients might declare that they accept, rather than merely the set of mime-types that a particular generic function is capable of serving; handling wildcards in accept strings is particularly awkward in the filtering paradigm.

Note that in this example, the method on `specializer<` involves a non-trivial ordering of methods based on the `q` values specified in the accept header (whereas in sections 2.1 and 2.2 only a single extended specializer could be applicable to any given argument).

Also note that the accept specializer protocol is straightforwardly extensible to other suitable objects; for example, one simple debugging aid is to define that an `accept-specializer` should be applicable to `string` objects. This can be done in a modular fashion (see the code below, which can be completely disconnected from the code for Hunchentoot request objects), and generalizes to dealing with multiple web server libraries, so that content-negotiation methods are applicable to each web server's request objects.

[5]Hunchentoot is a web server written in Common Lisp, allowing the user to write handler functions to compute responses to requests; `http://weitz.de/hunchentoot/`

```
(defmethod generalizer-of-using-class
    ((gf accept-generic-function)
     (s string))
  (make-instance 'accept-generalizer
                 :header s
                 :next (call-next-method)))
(defmethod specializer-accepts-p
    ((s accept-specializer) (o string))
  (let* ((tree (parse-accept-string o))
         (q (q (media-type s) tree)))
    (and q (> q 0))))
```

The `next` slot in the `accept-generalizer` is used to deal with the case of methods specialized on the classes of objects as well as on the acceptable media types; there is a method on `specializer-accepts-generalizer-p` for specializers that are not of type `accept-specializer` which calls the generic function again with the next generalizer, so that methods specialized on the classes `tbnl:request` and `string` are treated as applicable to corresponding objects, though less specific than methods with `accept-specializer` specializations.

3. PROTOCOL

In section 2, we have seen a number of code fragments as partial implementations of particular non-standard method dispatch strategies, using `generalizer` metaobjects to mediate between the methods of the generic function and the actual arguments passed to it. In section 3.1, we go into more detail regarding these `generalizer` metaobjects, describing the generic function invocation protocol in full, and showing how this protocol allows a similar form of effective method cacheing as the standard one does. In section 3.2, we show the results of some simple performance measurements on our implementation of this protocol in the SBCL implementation [11] of Common Lisp to highlight the improvement that this protocol can bring over a naïve implementation of generalized dispatch, as well as to make the potential for further improvement clear.

3.1 Generalizer metaobjects

3.1.1 Generic function invocation

As in the standard generic function invocation protocol, the generic function's actual functionality is provided by a discriminating function. The functionality described in this protocol is implemented by having a distinct subclass of `standard-generic-function`, and a method on `compute-discriminating-function` which produces a custom discriminating function. The basic outline of the discriminating function is the same as the standard one: it must first compute the set of applicable methods given particular arguments; from that, it must compute the effective method by combining the methods appropriately according to the generic function's method combination; finally, it must call the effective method with the arguments.

Computing the set of applicable methods is done using a pair of functions: `compute-applicable-methods`, the standard metaobject function, and a new function `compute-applicable-methods-using-generalizers`. We define a custom method on `compute-applicable-methods` which tests the applicability of a particular specializer against a given argument using `specializer-accepts-p`, a new protocol function with default implementations on `class` and `eql-specializer` to implement the expected behaviour. To order the methods, as required by the protocol, we define a pairwise comparison operator `specializer<` which defines an ordering between specializers for a given generalizer argument (remembering that even in standard CLOS the ordering between `class` specializers can change depending on the actual class of the argument).

The new `compute-applicable-methods-using-generalizers` is the analogue of the MOP's `compute-applicable-methods-using-classes`. Instead of calling it with the `class-of` each argument, we compute the generalizers of each argument using the new function `generalizer-of-using-class` (where the `-using-class` refers to the class of the generic function rather than the class of the object), and call `compute-applicable-methods-using-generalizers` with the generic function and list of generalizers. As with `compute-applicable-methods-using-classes`, a secondary return value indicates whether the result of the function is definitive for that list of generalizers.

Thus, in generic function invocation, we first compute the generalizers of the arguments; we compute the ordered set of applicable methods, either from the generalizers or (if that is not definitive) from the arguments themselves; then the normal effective method computation and call can occur. Unfortunately, the nature of an effective method function is not specified, so we have to reach into implementation internals a little in order to call it, but otherwise the remainder of the generic function invocation protocol is unchanged from the standard one. In particular, method combination is completely unchanged; programmers can choose arbitrary method combinations, including user-defined long form combinations, for their generic functions involving generalized dispatch.

3.1.2 Effective method memoization

The potential efficiency benefit to having `generalizer` metaobjects lies in the use of `compute-applicable-methods-using-generalizers`. If a particular generalized specializer accepts a variety of objects (such as the `signum` specializer accepting all reals with a given sign, or the `accept` specializer accepting all HTTP requests with a particular `Accept` header), then there is the possibility of cacheing and reusing the results of the applicable and effective method computation. If the computation of the applicable method from `compute-applicable-methods-using-generalizers` is definitive, then the ordered set of applicable methods and the effective method can be cached.

One issue is what to use as the key for that cache. We cannot use the generalizers themselves, as two generalizers that should be considered equal for cache lookup will not compare as `equal` – and indeed even the standard generalizer, the `class`, cannot easily be used as we must be able to invalidate cache entries upon class redefinition. The issue of `class` generalizers we can solve as in [8] by using the `wrapper` of a class, which is distinct for each distinct (re)definition of a class; for arbitrary generalizers, however, there is *a priori* no good way of computing a suitable hash key automatically, so we allow the metaprogrammer to specify one by defining a method on `generalizer-equal-hash-key`, and combining the hash keys for all required arguments in a list to use as a key in an `equal` hash-table.

3.2 Performance

We have argued that the protocol presented here allows for expressive control of method dispatch while preserving the possibility of efficiency. In this section, we quantify the efficiency that the memoization protocol described in section 3.1.2 achieves, by comparing it both to the same protocol with no memoization, as well as with equivalent dispatch implementations in the context of methods with regular specializers (in an implementation similar to that in [8]), and with implementation in straightforward functions.

In the case of the `cons-specializer`, we benchmark the walker acting on a small but non-trivial form. The implementation strategies in the table below refer to: an implementation in a single function with a large `typecase` to dispatch between all the cases; the natural implementation in terms of a standard generic function with multiple methods (the method on `cons` having a slightly reduced `typecase` to dispatch on the first element, and other methods handling `symbol` and other atoms); and three separate cases using `cons-specializer` objects. As well as measuring the effect of memoization against the full invocation protocol, we can also introduce a special case: when only one argument participates in method selection (all the other required arguments only being specialized on `t`), we can avoid the construction of a list of hash keys and simply use the key from the single active generalizer directly.

implementation	time (µs/call)	overhead
function	3.17	
standard-gf/methods	3.6	+14%
cons-gf/one-arg-cache	7.4	+130%
cons-gf	15	+370%
cons-gf/no-cache	90	+2700%

The benchmarking results from this exercise are promising: in particular, the introduction of the effective method cache speeds up the use of generic specializers in this case by a factor of 6, and the one-argument special case by another factor of 2. For this workload, even the one-argument special case only gets to within a factor of 2-3 of the function and standard generic function implementations, but the overall picture is that the memoizability in the protocol does indeed drastically reduce the overhead compared with the full invocation.

For the `signum-specializer` case, we choose to benchmark the computation of 20!, because that is the largest factorial whose answer fits in SBCL's 63-bit fixnums – in an attempt to measure the worst case for generic dispatch, where the work done within the methods is as small as possible without

being meaningless, and in particular does not cause heap allocation or garbage collection to obscure the picture.

implementation	time (µs/call)	overhead
function	0.6	
standard-gf/fixnum	1.2	+100%
signum-gf/one-arg-cache	7.5	+1100%
signum-gf	23	+3800%
signum-gf/no-cache	240	+41000%

The relative picture is similar to the `cons-specializer` case; including a cache saves a factor of 10 in this case, and another factor of 3 for the one-argument cache special case. The cost of the genericity of the protocol here is starker; even the one-argument cache is a factor of 6 slower than the standard generic-function implementation, and a further factor of 2 away from the implementation of factorial as a function. We discuss ways in which we expect to be able to improve performance in section 5.1.

We could allow the metaprogrammer to improve on the one-argument performance by constructing a specialized cache: for `signum` arguments of `rational` arguments, the logical cache structure is to index a three-element vector with `(1+ signum)`. The current protocol does not provide a way of eliding the two generic function calls for the generic cache; we discuss possible approaches in section 5.

3.3 Full protocol

The protocol described in this paper is only part of a complete protocol for `specializer` and `generalizer` metaobjects. Our development of this protocol is as yet incomplete; the work described here augments that in [9], but is yet relatively untested – and additionally our recent experience of working with that earlier protocol suggests that there might be useful additions to the handling of `specializer` metaobjects, independent of the `generalizer` idea presented here.

4. RELATED WORK

The work presented here builds on specializer-oriented programming described in [9]. Approximately contemporaneously, filtered dispatch [3] was introduced to address some of the same use cases: filtered dispatch works by having a custom discriminating function which wraps the usual one, where the wrapping function augments the set of applicable methods with applicable methods from other (hidden) generic functions, one per filter group; this step is not memoized, and using `eql` methods to capture behaviours of equivalence classes means that it is hard to see how it could be. The methods are then combined using a custom method combination to mimic the standard one; in principle implementors of other method combinations could cater for filtered dispatch, but they would have to explicitly modify their method combinations. The Clojure programming language supports multimethods[6] with a variant of filtered dispatch as well as hierarchical and identity-based method selectors.

In context-oriented programming [6, 16], context dispatch occurs by maintaining the context state as an anonymous class with the superclasses representing all the currently active layers; this is then passed as a hidden argument to context-aware functions. The set of layers is known and under programmer control, as layers must be defined beforehand.

In some sense, all dispatch schemes are specializations of predicate dispatch [4]. The main problem with predicate dispatch is its expressiveness: with arbitrary predicates able to control dispatch, it is essentially impossible to perform any substantial precomputation, or even to automatically determine an ordering of methods given a set of arguments. Even Clojure's restricted dispatch scheme provides an explicit operator for stating a preference order among methods, where here we provide an operator to order specializers; in filtered dispatch the programmer implicitly gives the system an order of precedence, through the lexical ordering of filter specification in a filtered function definition.

The Slate programming environment combines prototype-oriented programming with multiple dispatch [13]; in that context, the analogue of an argument's class (in Common Lisp) as a representation of the equivalence class of objects with the same behaviour is the tuple of roles and delegations: objects with the same roles and delegations tuple behave the same, much as objects with the same generalizer have the same behaviour in the protocol described in this paper.

The idea of generalization is of course not new, and arises in other contexts. Perhaps of particular interest is generalization in the context of partial evaluation; for example, [12] considers generalization in online partial evaluation, where sets of possible values are represented by a type system construct representing an upper bound. Exploring the relationship between generalizer metaobjects and approximation in type systems might yield strategies for automatically computing suitable generalizers and cache functions for a variety of forms of generalized dispatch.

5. CONCLUSIONS

In this paper, we have presented a new generalizer metaobject protocol allowing the metaprogrammer to implement in a straightforward manner metaobjects to implement custom method selection, rather than the standard method selection as standardized in Common Lisp. This protocol seamlessly interoperates with the rest of CLOS and Common Lisp in general; the programmer (the user of the custom specializer metaobjects) may without constraints use arbitrary method combination, intercede in effective method combination, or write custom method function implementations. The protocol is expressive, in that it handles forms of dispatch not possible in more restricted dispatch systems, while not suffering from the indeterminism present in predicate dispatch through the use of explicit ordering predicates.

The protocol is also reasonably efficient; the metaprogrammer can indicate that a particular effective method computation can be memoized, and under those circumstances much of the overhead is amortized (though there remains a substantial overhead compared with standard generic-function or regular function calls). We discuss how the efficiency could be improved below.

[6] `http://clojure.org/multimethods`

5.1 Future work

Although the protocol described in this paper allows for a more efficient implementation, as described in section 3.1.2, than computing the applicable and effective methods at each generic function call, the efficiency is still some way away from a baseline of the standard generic-function, let alone a standard function. Most of the invocation protocol is memoized, but there are still two full standard generic-function calls – `generalizer-of-using-class` and `generalizer-equal-hash-key` – per argument per call to a generic function with extended specializers, not to mention a hash table lookup.

For many applications, the additional flexibility afforded by generalized specializers might be worth the cost in efficiency, but it would still be worth investigating how much the overhead from generalized specializers can be reduced; one possible avenue for investigation is giving greater control over the cacheing strategy to the metaprogrammer.

As an example, consider the `signum-specializer`. The natural cache structure for a single argument generic function specializing on `signum` is probably a four-element vector, where the first three elements hold the effective methods for `signum` values of -1, 0, and 1, and the fourth holds the cached effective methods for everything else. This would make the invocation of such functions very fast for the (presumed) common case where the argument is in fact a real number. We hope to develop and show the effectiveness of an appropriate protocol to allow the metaprogrammer to construct and exploit such cacheing strategies, and (more speculatively) to implement the lookup of an effective method function in other ways.

We also aim to demonstrate support within this protocol for some particular cases of generalized specializers which seem to have widespread demand (in as much as any language extension can be said to be in "demand"). In particular, we have preliminary work towards supporting efficient dispatch over pattern specializers such as implemented in the Optima library[7], and over a prototype object system similar to that in Slate [13]. Our current source code for the work described in this paper can be seen in the git source code repository at `http://christophe.rhodes.io/git/specializable.git`, which will be updated with future developments.

Finally, after further experimentation (and, ideally, non-trivial use in production) if this protocol stands up to use as we hope, we aim to produce a standards-quality document so that other implementors of Common Lisp can, if they choose, independently reimplement the protocol, and so that users can use the protocol with confidence that the semantics will not change in a backwards-incompatible fashion.

5.2 Acknowledgments

We thank Lee Salzman, Pascal Costanza and Mikel Evins for helpful and informative discussions, and all the respondents to the first author's request for imaginative uses for generalized specializers.

[7] `https://github.com/m2ym/optima`

6. REFERENCES

[1] Craig Chambers. Predicate Classes. In Oscar Nierstrasz, editor, *ECOOP 1993 – Object-Oriented Programming*, number 707 in LNCS, pages 268–296. Springer, 1993.

[2] Pascal Costanza and Charlotte Herzeel. make-method-lambda considered harmful. In *European Lisp Workshop*, 2008.

[3] Pascal Costanza, Charlotte Herzeel, Jorge Vallejos, and Theo D'Hondt. Filtered Dispatch. In *Dynamic Languages Symposium*. ACM, 2008.

[4] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In Eric Jul, editor, *ECOOP 1998 – Object-Oriented Programming*, number 1445 in LNCS, pages 186–211. Springer, Berlin, 1998.

[5] R. Fielding, J. Gettys, J. Movil, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999.

[6] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[7] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Mass., 1991.

[8] Gregor Kiczales and Luis Rodriguez. Efficient Method Dispatch in PCL. In *LISP and Functional Programming*, pages 99–105, Nice, 1990.

[9] J. Newton and C. Rhodes. Custom specializers in Object-oriented Lisp. *Journal of Universal Computer Science*, 14(20):3370–3388, 2008. Presented at *European Lisp Symposium*, Bordeaux, 2008.

[10] Kent Pitman and Kathy Chapman, editors. *Information Technology – Programming Language – Common Lisp*. Number 226–1994 in INCITS. ANSI, 1994.

[11] C. Rhodes. SBCL: A Sanely-Bootstrappable Common Lisp. In Robert Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems*, number 5146 in LNCS, pages 74–86. Springer-Verlag, Berlin, 2008. Presented at *Workshop on Self-Sustaining Systems*, Potsdam, 2008.

[12] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford, California, USA, 1993.

[13] Lee Salzman and Jonathan Aldrich. Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model. In Andrew P. Black, editor, *ECOOP 2005 – Object-Oriented Programming*, number 3586 in LNCS, pages 312–336. Springer, Berlin, 2005.

[14] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, Newton, Mass., 1984.

[15] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, Newton, Mass., second edition, 1990.

[16] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated Generic Functions: Enabling Context-Dependent Method Dispatch. In *International Conference on Software Composition*, pages 66–81, 2010.

Session II: Demonstrations

web-mode.el
Heterogeneous recursive code parsing with Emacs Lisp

François-Xavier Bois
Kernix Lab
15 rue Cels
75014 Paris - France
+33 (1) 53 98 73 43
fxbois@kernix.com

ABSTRACT
web-mode.el is an Emacs module for editing HTML templates. Unlike other "source codes", web templates can include various language components (*heterogeneous dimension*) that may embed themselves into each other (*recursive dimension*).

Indeed, an HTML document may contain a JavaScript that embeds a PHP block.

```
<div>
  <h1>title</h1>
  <script>var x = <?=$x?>;</script>
</div>
```

This recursive aspect invalidates standard ways of lexing (syntactic tokens), highlighting (colorizing) and indenting.

All the power of Emacs is necessary to enable contextual indentation and highlighting.

This paper describes web-mode.el features and some of its internal processes.

Categories and Subject Descriptors
Coding Tools and Techniques, Document and Text Editing, User Interfaces, Programming Environments, Reusable Software

General Terms
Algorithms, Languages.

Keywords
Emacs, Lisp, Web, HTML, templates, engines.

1. OVERVIEW
Emacs owes a big part of its success to two strengths that characterize it: its extensibility through modules (modes) and its "Lisp Machine" dimension.

Though compatible with the vast majority of programming languages, Emacs has suffered in the past years of its relative weakness in the field of web template editing. Those HTML documents which embed parts written in different languages (JavaScript, CSS, PHP, Java, Ruby, etc) are at the core of web development, a very dynamic domain of computer science.

1.1 Multi modes approach
HTML template editing under Emacs has long been the prerogative of "multi modes" like mumamo.el, mmm-mode.el or multi-web-mode.el. Those modes rely on the "available" dedicated modules to handle their corresponding "parts" in the template.

A template with HTML, JavaScript, CSS and PHP content may for example use

- nxml.el for HTML
- js2-mode.el for JavaScript code (located between `<script>` and `</script>` tags)
- css-mode.el for styles (between `<style>` and `</style>`)
- php-mode.el for PHP statements delimited by `<?php` and `?>`

In order to explain how this "multi modes" approach works, one should keep in mind that Emacs modes are divided in two families: the major and minor modes.

A major mode handles the responsibility of buffer highlighting and indentation; only one major mode may be running for a buffer at a given time.

To let many major modes "coexist", a "multi mode" (which is a minor mode) loads a specific major mode according to the cursor position `(point)`. If the cursor is between the delimiters `<?php` and `?>`, the "multi mode" activates the major mode php-mode.el. The *narrowing* mechanism helps the "multi-mode" to restrict the working space to the region between the delimiters (`<?php` and `?>`).

This "Unix-like" approach is very appealing:

- Each mode has one role that it tries to achieve the best it can.
- By combining those modes one may achieve a larger ambition.

Alas it can be very frustrating for the users:

- Going back and forth between modes triggers visual artifacts and slows down the workflow.
- Incompatibilities between modes result conflicts and errors.
- Customization is difficult and inconsistent (indeed, each major mode has its own parameters).
- The lack of common ground prevents advanced features (no context is shared between the major modes).

1.2 web-mode.el

Aware that no satisfactory results would ever happen with this approach, I started in 2011 the development of the major mode web-mode.el. The main features of this mode are

- Autonomous (no other major mode is required)
- Fast
- Simple (no configuration required)
- Effective (auto-closing, code folding, tag navigation)
- Aware of all the HTML format specificities
- Compatible with more than twenty engines (erb, jsp, php, asp, razor, django, mason, etc.)

2. IMPLEMENTATION NOTES

2.1 Terminology

As was said previously, a template may embed various code components. Two kinds will be considered here:

- A *part* is interpreted by the navigator (e.g. a JavaScript part or a CSS part).
- A *block* is processed (client-side or server-side) before being rendered by the navigator e.g. a PHP block, an Erb block, a dustjs block etc.

This terminology is not a standard but is useful for function/variable naming and for the documentation.

2.2 The main loop

Two tasks are executed as soon as the buffer is loaded and altered:

- Lexing: to detect the various entities (blocks, parts, nodes) and for each one, identify their tokens: block/part strings/comments, node attributes, block delimiters.
- Highlighting: to colorize the code.

2.3 Lexing phase

Three steps are necessary for parsing the code

1. Block identification and scanning.
2. HTML scanning (HTML tags / attributes / comments, doctype declaration)
3. Parts scanning (JavaScript, CSS).

When a file is opened, web-mode.el scans the entire buffer. Subsequent scans are performed on smaller zones "around" the altered zone: it is called the invalidation phase and is described below.

The order of the steps is very important: indeed, nodes/parts: identification must be done outside the blocks.

```
<div>
  <?php /* <script>var x = 1</script> */ ?>
</div>
```

Part scan is done during a specific step because web-mode.el can be used to edit files whose content-type is not HTML but JavaScript or CSS. For example *.js.erb is a Ruby on Rails JavaScript template.

Given the large variety of tokens (specific to languages) and the recursive dimension of code, web-mode.el can't rely on Emacs internal functions to tokenize and highlight.

Parameters are associated to each engine family and are used for tokenizing, indenting, highlighting, auto-pairing and auto-closing.

With each engine are associated

- Delimiters (e.g. `<?php ?>`)
- Control blocks (e.g. `<?php if(): ?>`)
- Syntactic tokens regular expressions
- Auto-pairs, snippets

In order to parse the code, web-mode.el must know which is the engine associated with the template. This association is automatic as soon as the file extension is obvious (e.g. *.erb). Association must be forced when the extension is too common.

```
(require 'web-mode)
(add-to-list 'auto-mode-alist
             '("\\.html\\'" . web-mode))

(setq web-mode-engines-alist
      '(("erb" . "/rails/.*\\.html\\'")
        ("php" . "/zend/.*\\.html\\'"))
)
```

2.4 Custom text properties

The scanning/lexing phase will help store information that will be used to implement interactive features like indentation, folding, or tag navigation.

This process is achieved with the "text-properties" which are plists attached to every single character in the buffer.

web-mode.el adds to the common properties (e.g. `'face`, `'visibility`) new ones that describe the following states

- `'block-side` is set to `t` throughout blocks characters
- `'block-(beg|end)` mark the blocks boundaries
- `'block-token`[1] is `'string`, `'comment` or `'delimiter` on block tokens
- `'block-controls` is the current block a control block? (e.g. `{% for %}` … `{% endfor %}`)
- `'tag-type` tells if the tag is a `'start`, `'end` or `'void` tag (`'tag-name` store the tag name).

Bitmask on `'*-beg` properties is an additional way to store information useful for the highlighting phase.

2.5 Indentation

As the main purpose of web-mode.el is to remain autonomous, a generic indentation engine was developed.

web-mode.el deals with three kinds of indentations

1/ HTML indentation relies on finding the first previous line beginning with a start tag. Deciding if the current line should be indented is done by counting start / end tags and see if a start tag remains unclosed.

[1] All the 'block-* properties are available as 'part-* equivalents.

2/ Bracket based indentation. It relies on counting the number of unclosed brackets. Inline calls are also handled by looking at the first unclosed bracket. This kind of indentation is used by languages like PHP, JavaScript, CSS, etc.

3/ With stack based indentation, each indentation line depends directly on the current and previous lines. For instance, if the previous line is a control statement (`if`) the current line is indented.

It is important to remember that templates are at the core of the MVC design pattern. The developer must separate the various logical components of its application: Model, View (templates) and Controller. The more code is placed in the Model or Controller components, the better.

More generally, a good habit in web development is to put foreign code in specific files (e.g. *.css for styles, *.js for javascripts, *.php for engine statements). Thus the indentation engine should not have to deal with complex and large parts or blocks.

2.6 Consistency

Being able to consider the whole buffer state is very useful to implement advanced features. The markup indentation engine can for example evaluate HTML elements and control blocks when calculating the indentation offset.

```
<div>
  <?php if ($x): ?>
    <span></span>
  <?php endif; ?>
<div>
```

2.7 Highlighting

Highlighting a buffer in Emacs involves the font-locking mechanism. The recursive dimension of templates and some complex parsing rules (e.g. for HTML attributes) prevents the use of standards font-lock keywords.

As for the scanning phase, the highlighting phase involves three steps:

1. node highlighting
2. part highlighting
3. block highlighting

Ending with block highlighting reflects a logical situation: a block can be included in a part or a node, block highlighting is thus priority.

Two techniques are used for highlighting

- Direct setting of the `'font-lock-face` text-property for HTML nodes (brackets, tags, attributes)
- Font-locking keywords for parts and blocks.

2.8 Decoration

After the highlighting phase, web-mode.el may “decorate” some of the tokens:

- String: variable interpolation (for erb and php, in double quoted string), css colorization (background refects the css color).
- Comment: keyword highlighting (ex. TODO, FIX).

2.9 Invalidation

Most major modes delegate “region invalidation” to Emacs. This process is the responsibility of font-locking; it automatically detects syntactic tokens and refreshes the colors of the altered zone.

As web-mode.el scans the buffer by itself, it has to trigger a new scan as soon as the user alters the buffer. The `'after-change-function` hook is used for this purpose. To ensure that parts and blocks are properly scanned, the following rule has been set: the region should begin and end with an HTML tag.

For the highlighting phase, the same region should be considered. web-mode.el can influence font-lock by associating a custom function to the `'font-lock-extend-region-functions`.

One should note that the lexical invalidation must be done before the highlighting; indeed highlighting uses some of the text-properties set by the lexical process (`'tag-attr`, `'block-token`, `'part-token`, etc.)

3. CONCLUSION

Thanks to the power of Lisp and to the advanced Emacs mechanisms, web-mode.el is able to provide a very robust and rich experience to its users.

Invalidation of a zone located in a part or a block is still a flaw that needs to be addressed. Indeed when such a part or block is huge, re scanning and re highlighting it entirely can be pricy. Identifying a narrower zone inside the block (or the part) is a very difficult task whenever this process must work with many languages/engines.

4. ACKNOWLEDGMENTS

A special thanks to Stefan Monnier a great Emacs maintainer and a wonderful guide to the Emacs internals.

5. REFERENCES

[1] François-Xavier Bois. *web-mode.el presentation and documentation*. http://web-mode.org.

[2] François-Xavier Bois. *web-mode.el code repository.* https://github.com/fxbois/web-mode.

[3] James Clark. *nXML mode, powerful mode for editing XML documents*. http://www.thaiopensource.com/nxml-mode.

[4] Multi Modes. *Introduction to multi modes*. http://www.emacswiki.org/emacs/MultipleModes.

Demonstration: The OMAS Multi-Agent Platform

Jean-Paul A. Barthès
UMR CNRS 7253 Heudiasyc
Université de Technologie de Compiègne
60205 Compiègne, France
barthes@utc.fr

ABSTRACT

OMAS is a platform developed for easy implementation of complex agent systems. It has a number of interesting features including four predefined types of agents: service agents, personal assistant agents, transfer agent and rule-based agents. Organization is peer to peer with no central functionalities. Personal assistants can interact in natural language typed or vocal. Multilingualism is supported. OMAS has been used in a number of projects during the last years. It was developed in the MCL and Allegro environments, and currently works in the Allegro Common Lisp environment. Persistency uses AllegroCache.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Data types and structures

General Terms

MAS, Programming structures

1. MULTI-AGENT SYSTEMS

Multi-agent systems (MAS) are systems involving agents, i.e. independent pieces of software that can run autonomously. MAS range from sets of reactive agents, similar to active objects, to cognitive agents capable of independent reasoning and free to take decisions. Reactive agents are used mainly for simulating systems with a large number of agents (e.g. ant colonies), cognitive agents are limited to a smaller set of agents mainly because they are much more complex and difficult to build. Agents communicate by exchanging messages. In the last 10 years the FIPA organization, now part of IEEE[1], issued a number of recommendations that led to the FIPA standards. Many platforms (toolkits), over 200, have been proposed to help people develop multi-agent systems, one of the most used worldwide being JADE written in Java[2]. There are not many platforms written in Lisp, the most famous one being SOAR[3], a complex environment designed to simulate cognitive behavior. We developed OMAS (Open Multi-Agent System) for our own needs, first for robotics, then with a broader scope, namely to be able to prototype complex systems involving cognitive agents interacting with humans easily[4].

[1]http://fipa.org/about/fipa_and_ieee.html
[2]http://jade.tilab.com/

2. PREVIOUS PROJECTS

OMAS [2] has been used in a number of projects among which the following ones.

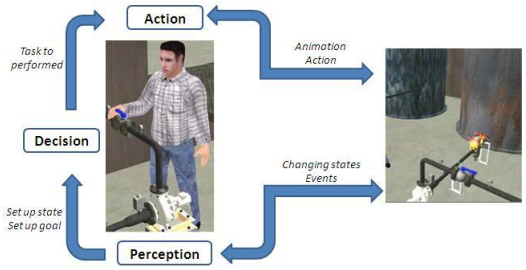


Figure 1: Avatar in the virtual environment controlled by an OMAS agent

V3S a project intended to train operators in the context of dangerous industrial plants (SEVESO plants) by simulating the work to do in a virtual environment populated by avatars controlled by agents (Fig.1). In this project OMAS was coupled with a virtual reality platform [4].

CODAVI a project testing the possibility of modeling a car as an intelligent agent with the driver interacting using voice and natural language (Fig.2). In the project OMAS was coupled with a fast real time platform, PACPUS, monitoring all the car systems [1].

TATIN-PIC a project using an interactive graphic surface and board for cooperative preliminary design (Fig.3). OMAS was interfaced with JADE. JADE was used to run the graphics and OMAS was used to provide personal assistants to the participants, allowing them to interact using voice and natural language [5].

[3]http://www.soartech.com/images/uploads/file/SoarTech_Autonomous_Platforms_Review.pdf
[4]OMAS can be downloaded (watch the tilde) from http://www.utc.fr/∼barthes/OMAS/

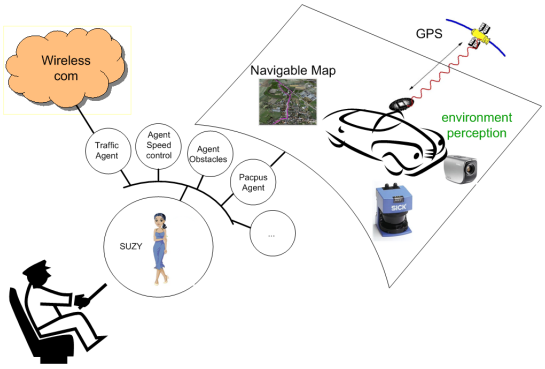


Figure 2: The personal assistant SUZY in the CO-DAVI project

HDSRI a project developed for managing the international relationship of our laboratory. It contains agents for handling contacts, international projects, missions, international announcements of research programs.

NEWS a prototype of international platform for exchanging multilingual information while letting the participants access the system using their own language.

Other applications are being developed in France, Japan and Brazil, e.g. [6].

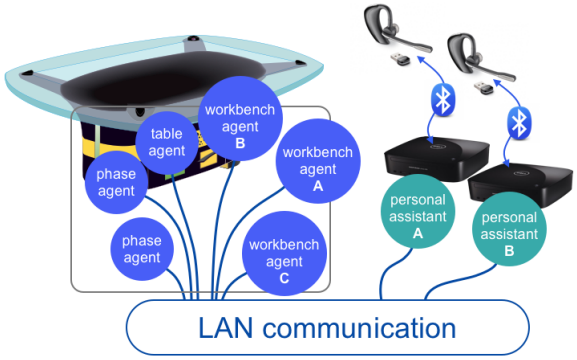


Figure 3: Simplified TATIN-PIC architecture showing the connection between the JADE Java agents (left) and the OMAS Lisp agents (right)

3. OMAS FEATURES

3.1 Agents

Our agents are rather complex. They are multithreaded and can answer several requests at the same time. They are built to last, meaning that, once they are created, they remain in the system waiting for something to do. Agents have skills (what they can do) and goals (what they plan to do). Agents are organized in groups called *coteries*, federating local subgroups. There are four types of agents: service agents, personal assistant agents, transfer agents (also called postmen) and rule-based agents. Agents have their own ontology and knowledge base. They reason using goals and queries on the knowledge base.

A particular care has been taken when designing personal assistant agents (PAs). One can implement natural language dialogs with vocal interaction [3, 5]. PAs can have helpers as staff agents in charge of more technical matters, implementing the concept of "digital butler."

Transfer agents or postmen are used for connecting local coteries, or for interacting with other systems (multi-agent or not).

3.2 Agent Communication Language (ACL)

Messages are structured using the OMAS Communication Language that can be translated by postmen agents to other ACLs, e.g. FIPA ACL.

Communication can be point-to-point, multicast, broadcast or conditional. Protocol is a subset of FIPA and supports Contract-Net, a special protocol allowing cooperation.

Messages are received by agents in their mailbox, and if not processed readily, are inserted into an agenda to be processed later. Each request gives birth to several processes (threads).

Because JADE is a widely used Java platform, we developed an interface at a fairly low level allowing OMAS to call any JADE agent directly and vice-versa.

4. DEMONSTRATION

The demonstration will show the NEWS application, a prototyped multilingual NEWS systems, on several machines (if possible). Time permitting, I also will show how to build an OMAS agent and add it to an existing application, how to access and edit objects belonging to agents using a web browser, and how to use the IDE.

5. REFERENCES

[1] J.-P. Barthès and P. Bonnifait. Multi-Agent Active Interaction with Driving Assistance Systems. In I. ITSC, editor, *Multi-Agent Active Interaction with Driving Assistance Systems*, pages 1–7, Funchal, Portugal, Sept. 2010. 7 pages.

[2] J.-P. A. Barthès. Omas - a flexible multi-agent environment for cscwd. *Future Generation Computer Systems*, 27:78–87, 2011.

[3] J.-P. A. Barthès. Improving human-agent communication using linguistic and ontological cues. *Int. J. Electronic Business*, 10(3):207–231, 2013.

[4] L. Edward, D. Lourdeaux, and J.-P. A. Barthès. Virtual autonomous agents in an informed environment for risk prevention. In *IVA*, pages 496–497, 2009.

[5] A. Jones, A. Kendira, C. Moulin, J.-P. A. Barthès, D. Lenne, and T. Gidel. Vocal Interaction in Collocated Cooperative Design. In *Proc. ICCI*CC 2012*, pages 246–252, 2012.

[6] K. Sugawara and J.-P. A. Barthès. An Approach to Developing an Agent Space to Support Users' Activities. In *Proc. The Fifth International Conference on Advances in Human-oriented and Personalized Mechanisms*, pages 84–90, Lisbon, Portugal, 2012.

Yet Another Wiki!

Alain Marty Engineer Architect
66180, Villeneuve de la Raho, France
marty.alain@free.fr

Abstract

The present contribution introduces a small environment working on top of any modern browser, allowing to write, style and script dynamic WEB pages using a simple and unique LISP-like syntax.

Keywords

Wiki, CMS, interpreter, language, Lisp

1. INTRODUCTION

Web browsers can parse data (HTML code, CSS rules, JS code, ...) stored on the server side and display rich multimedia dynamic pages on the client side. Some HTML functions, (texarea, input, form, ...) associated with script languages (PHP,...) allow interactions with these data leading to web apps like **blogs, wikis and CMS**. Hundreds of engines have been built, managing files on the server side and interfaces on the client side, such as Wordpress, Wikipedia, Joomla,.... Syntaxes are proposed to simplify text enrichment, pages composing, multimedia handling. The **Markdown syntax** is the *de facto* standard to help writing styled and structured texts, but stays far from the wish of the father of LISP, **John McCarthy**: *« An environment where the markup, styling and scripting is all s-expression based would be nice. »* Works have been done in this direction, for instance:

- **Skribe** [1] a text-processor based on the SCHEME programming language dedicated to writing web pages,
- **HOP** [2] a Lisp-like progamming language for the Web 2.0, based on SCHEME,
- **BRL** [3] based on SCHEME and designed for server-side WWW-based applications.

All of these projects are great and powerful. With the plain benefit of existing SCHEME implementations they make a strong junction between the mark-up (HTML/CSS) and programming (JS, PHP,...) syntaxes. But these tools are devoted to developers, not to users or web-designers.

The **α-wiki project** [4] is intended to **link** the user, the web-designer and the developer in a single collaborative work: 1) **α-wiki** is **a small wiki** intended to be easy to install, its archive is about 100kb (about 1000 JS lines), with nothing but PHP on the server side and no external library. 2) **α-wiki** is a small and easy to use environment on top of the browser allowing to write, style and script WEB pages with the **same** LISP-like syntax: **λ-talk**. In this paper I will present a few elements of this syntax and its evaluator.

2. λ-talk SYNTAX

The code is keyed into the frame editor as a mix of plain text and s-expressions. Valid s-expressions are evaluated by **λ-talk** and displayed by **α-wiki** in the wiki page; others are ignored. At least, the code is displayed without any enrichment and without any structure, as a sequence of words.

2.1 Words

First of all, **α-wiki** is a **text editor**. As in any text editor, enriching a sequence of words proceeds into two steps: **select & apply**. In **α-wiki**, selection uses curly braces **{ }** and application uses a dictionary of HTML tags, to build s-expressions : **{tag any text}**. **λ-talk** translates them into HTML expressions to be evaluated and displayed by the browser. For instance, writing in the editor frame:

```
{div {@ id="myId"
        style="text-align:center;
           border:1px solid;"}
 I am {b fat},
 I am {b {i fat italicized}},
 I am {b {i {u
  fat italicized underlined}}}.
}
```

displays in the wiki page :

I am **fat**, I am ***fat italicized***, I am ***fat italicized underlined***.

Note that the function **@** contains HTML attributes and CSS rules expressed in the standard HTML/CSS syntax, not in an s-expression syntax. This is a matter of choice : not to use a pure s-expression such as **{@ {id myId} {style {text-align center} {border 1px solid}}}** avoids dictionary pollution, support HTML/CSS future evolution and is well known by a web-designer.

2.2. Numbers

α-wiki offers the usual numeric computation capabilities that a pocket calculator would have. Following the same syntax **{first rest}** where **first** is a math function (+, -, *, /, %, sqrt, ...) and **rest** a sequence of numbers and/or valid s-expressions, any complex math expressions can be evaluated by **λ-talk** and inserted anywhere in the page, for instance writing in the editor frame:

```
1: {* 1 2 3 4 5 6}
2: {sqrt {+ {* 3 3} {* 4 4}}}
3: {sin {/ {PI} 2}}
4: {map {lambda {:x} {* :x :x}} {serie 1 10}}
5: {reduce + {serie 1 100}}
```

displays in the wiki page :

1: 720
2: 5
3: 1
4: 1 4 9 16 25 36 49 64 81 100
5: 5050

2.3. Code

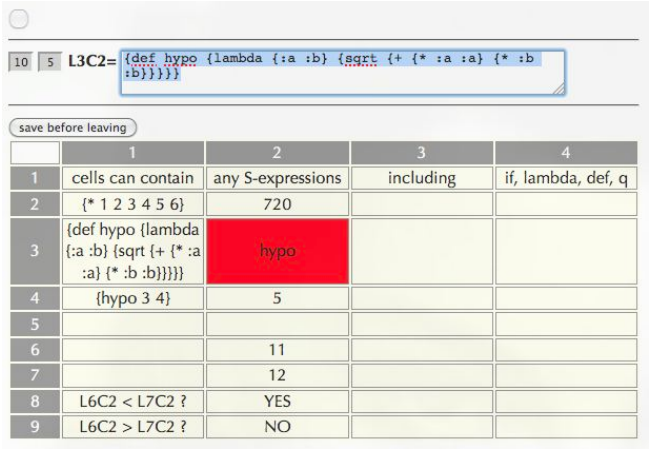
λ-talk is a programmable programming language. It keeps from LISP nothing but three special forms (**lambda, def, if**) opening the door to recursion (and thus iteration), local variables (via lambdas), partial application (currying). The **if, lambda, def** forms can be nested and the **λ-talk**'s dictionary can be extended via the **def** form. For instance, writing in the editor frame :

```
{b 1) a basic function:}
{def hypo
 {lambda {:a :b}
  {sqrt {+ {* :a :a} {* :b :b}}}}}
hypo(3,4) = {hypo 3 4}
{b 2) a recursive function:}
{def fac
 {lambda {:n}
  {if {< :n 1}
   then 1
   else {* :n {fac {- :n 1}}}}}}
fac(6) = {fac 6}
{b 3) the first derivees of y=x{sup 3}
                using partial function calls:}
{def D
 {lambda {:f :x}
  {/ {- {:f {+ :x 0.01}}
        {:f {- :x 0.01}}} 0.02}}}
{def cubic
 {lambda {:x} {* :x :x :x}}}
cubic(1)={cubic 1}
cubic'(1)={{D cubic} 1}
cubic''(1)={{D {D cubic}} 1}
cubic'''(1)={{D {D {D cubic}}} 1}
cubic''''(1)={{D {D {D {D cubic}}}} 1}
```

displays in the wiki page:

1) a basic function:
hypo
hypo(3,4) = 5
2) a recursive function:
fac(6) = 720
3) the first derivees of $y=x^3$ using partial function calls:
cubic cubic(1) = 1
cubic'(1) = 3.0000999999999998 ≠3
cubic''(1) = 5.999999999999628 ≠6
cubic'''(1) = 6.000000000007111 ≠6
cubic''''(1) = 4.107825191113079e-9 ≠0

And the underground **JS** language can always be called via the **input** function and external plugins to give access to user interaction (buttons) and more complex tools like graphics, raytracing, fractals, and spreadsheets. Spreadsheets are known to be a good illustration of the functional approach, for instance:



	1	2	3	4
1	cells can contain	any S-expressions	including	if, lambda, def, q
2	{* 1 2 3 4 5 6}	720		
3	{def hypo {lambda {:a :b} {sqrt {+ {* :a :a} {* :b :b}}}}}	hypo		
4	{hypo 3 4}	5		
5				
6		11		
7		12		
8	L6C2 < L7C2 ?	YES		
9	L6C2 > L7C2 ?	NO		

3. λ-talk EVALUATOR

The **λ-talk**'s code is a function defined and executed on page loading. This function creates a dictionary containing a set of pairs [**function_name : function_value**], defines the function **evaluate()** and a few associated ones. The function evaluate() is called at every keyUp and the page's display follows the edition in real-time:

```
function evaluate(str) {
  str = preprocessing( str );
  str = eval_ifs( str );
  str = eval_lambdas( str );
  str = eval_defs( str, true );
  str = eval_sexprs( str );
  str = postprocessing( str );
  return str;
};
```

The **eval_sexprs()** function starts a loop based on a single pattern (a Regular Expression) used in only one **JS** line to replace s-expressions by HTML expressions or evaluated math expressions:

```
function eval_sexprs(str) {
 var rex=/\{([^\s{}]*)(?:[\s]*)([^{}]*)\}/g;
 while (str != (str =
                  str.replace(rex,do_apply)));
 return str;
}
function do_apply() {
 var f = arguments[1], r = arguments[2];
 if (dico.hasOwnProperty(f))
  return dico[f].apply(null,[r]);
 else
  return '('+f+' '++')';
};
```

The three special forms "**if, lambda, def**" are pre-processed before the s-expressions evaluation. For instance, this is the simplified pseudo-code of the **eval_lambda()** function:

```
function eval_lambda(s) {
 s = eval_lambdas(s);
 var name = random_name()
 var args = get_arguments(s)
 var body = get_body(s)
 dico[name] = function(vals) {
  return function(bod){
   for every i in vals
    replace in bod args[i] by vals[i]
   return bod
  }(body)
 }
 return name;
}
```

The λ-talk's dictionary contains about 110 primitives handling HTML markup, math functions, ... For instance this the code of a simplified "*" function:

```
dico['*'] = function() {
 return arguments[0]*arguments[1]
};
```

4. CONCLUSION

With **α-wiki** and **λ-talk**, the beginner, the web-designer and the developer benefit from a **simple text editor and a coherent syntax** allowing them, in a gentle learning slope and a collaborative work, to build sets of complex and dynamic pages.

5. REFERENCES

- [1] : Manuel Serrano, http://www-sop.inria.fr/,
- [2] : Manuel Serrano, http://en.wikipedia.org/wiki/Hop,
- [3] : Bruce R.Lewis, http://brl.sourceforge.net/,
- [4] : Alain Marty, http://epsilonwiki.free.fr/alphawiki_2/

Session III: Application and Deployment Issues

High performance concurrency in Common Lisp - hybrid transactional memory with STMX

Massimiliano Ghilardi
TBS Group
AREA Science Park 99, Padriciano
Trieste, Italy
massimiliano.ghilardi@gmail.com

ABSTRACT

In this paper we present STMX, a high-performance Common Lisp implementation of transactional memory.

Transactional memory (TM) is a concurrency control mechanism aimed at making concurrent programming easier to write and understand. Instead of traditional lock-based code, a programmer can use atomic memory transactions, which can be composed together to make larger atomic memory transactions. A memory transaction gets committed if it returns normally, while it gets rolled back if it signals an error (and the error is propagated to the caller).

Additionally, memory transactions can safely run in parallel in different threads, are re-executed from the beginning in case of conflicts or if consistent reads cannot be guaranteed, and their effects are not visible from other threads until they commit.

Transactional memory gives freedom from deadlocks and race conditions, automatic roll-back on failure, and aims at resolving the tension between granularity and concurrency.

STMX is notable for the three aspects:

- It brings an actively maintained, highly optimized transactional memory library to Common Lisp, closing a gap open since 2006.
- It was developed, tested and optimized in very limited time - approximately 3 person months - confirming Lisp productivity for research and advanced programming.
- It is one of the first published implementations of hybrid transactional memory, supporting it since August 2013 - only two months after the first consumer CPU with hardware transactions hit the market.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures; F.1.2 [**Modes of Computation**]: Parallelism and concurency; D.2.13 [**Reusable Software**]: Reusable Libraries; D.2.11 [**Software Architectures**]: Patterns

General Terms

Algorithms, Theory

Keywords

Common Lisp, parallelism, concurrency, high-performance, transactions, memory

1. INTRODUCTION

There are two main reasons behind transactional memory.

The first is that in recent years all processors, from high-end servers, through consumer desktops and laptops, to tablets and smartphones, are increasingly becoming multi-core. After the Pentium D (2005), one of the first dual-core consumer CPU, only six years passed to see the 16-core AMD Opteron Interlagos (2011). Supercomputers and high-end servers are much more parallel than that, and even tablets and smartphones are often dual-core or quad-core. Concurrent programming has become mandatory to exploit the full power of multi-core CPUs.

The second reason is that concurrent programming, in its most general form, is a notoriously difficult problem [5, 6, 7, 11, 12]. Over the years, different paradigms have been proposed to simplify it, with various degrees of success: functional programming, message passing, futures, π-calculus, just to name a few.

Nowadays, the most commonly used is multi-threading with shared memory and locks (mutexes, semaphores, conditions ...). It is very efficient when used correctly and with fine-grained locks, as it is extremely low level and maps quite accurately the architecture and primitives found in modern multi-core processors. On the other hand, it is inherently fraught with perils: deadlocks, livelocks, starvation, priority inversion, non-composability, nondeterminism, and race conditions. The last two can be very difficult to diagnose, to reproduce, and to solve as they introduce non-deterministic behavior. To show a lock-based algorithm's correctness, for

example, one has to consider all the possible execution interleavings of different threads, which increases exponentially with the algorithm's length.

Transactional memory is an alternative synchronisation mechanism that solves all these issues (with one exception, as we will see). Advocates say it has clean, intuitive semantics and strong correctness guarantees, freeing programmers from worrying about low-level synchronization details. Skeptics highlight its disadvantages, most notably an historically poor performance - although greatly improved by recent hardware support (`Intel TSX` and `IBM Power ISA v.2.0.7`) - and that it does not solve livelocks, as it is prone to almost-livelocks in case of high contention.

STMX is a high-performance Common Lisp implementation of transactional memory. It is one of the first implementations supporting hybrid transactions, taking advantage of hardware transactions (`Intel TSX`) if available and using software-only transactions as a fallback.

2. HISTORY

Transactional memory is not a new idea: proposed as early as 1986 for Lisp [8], it borrows the concurrency approach successfully employed by databases and tries to bring it to general purpose programming. For almost ten years, it was hypothesized as a hardware-assisted mechanism. Since at that time no CPU supported the required instructions, it was mainly confined as a research topic.

The idea of software-only transactional memory, introduced by Nir Shavit and Dan Touitou in 1995 [11], fostered more research and opened the possibility of an actual implementation. Many researchers explored the idea further, and the first public implementation in Haskell dates back to 2005 [6].

Implementations in other languages followed soon: C/C++ (LibLTX, LibCMT, SwissTM, TinySTM), Java (JVSTM, Deuce), C# (NSTM, MikroKosmos), OCaml (coThreads), Python (Durus) and many others. Transactional memory is even finding its way in C/C++ compilers as GNU gcc and Intel icc.

Common Lisp had CL-STM, written in 2006 Google Summer of Code[1]. Unfortunately it immediately went unmaintained as its author moved to other topics. The same year Dave Dice, Ori Shalev and Nir Shavit [4] solved a fundamental problem: guaranteeing memory read consistency.

Despite its many advantages, software transactional memory still had a major disadvantage: poor performance. In 2012, both Intel[2] and IBM[3] announced support for hardware transactional memory in their upcoming lines of products. The IBM products are enterprise commercial servers implementing "Power ISA v.2.0.7": Blue Gene/Q[4] and zEnterprise EC12, both dated 2012, and Power8[5] released in May 2013. Intel products are the "Haswell" generation of Core i5 and Core i7, released in June 2013 - the first consumer CPUs offering hardware transactional memory under the name "Intel TSX".

Hardware support greatly improves transactional memory performance, but it is never guaranteed to succeed and needs a fallback path in case of failure.

Hybrid transactional memory is the most recent reinvention. Hypothesized and researched several times in the past, it was until now speculative due to lack of hardware support. In March 2013, Alexander Matveev and Nir Shavit [10] showed how to actually implement a hybrid solution that successfully combined the performance of Intel TSX hardware transactions with the guarantees of a software transaction fallback, removing the last technical barrier to adoption.

STMX started in March 2013 as a rewrite of CL-STM, and a first software-only version was released in May 2013. It was extended to support hardware transactions in July 2013, then hybrid transactions in August 2013, making it one of the first published implementations of hybrid transactional memory.

3. MAIN FEATURES

STMX offers the following functionalities, common to most software transactional memory implementations:

- atomic blocks: each `(atomic ...)` block runs code in a memory transaction. It gets committed if returns normally, while it gets rolled back if it signals an error (and the error is propagated to the caller). For people familiar with ContextL[6], transactions could be defined as layers, an atomic block could be a scoped layer activation, and transactional memory is analogous to a layered class: its behavior differs inside and outside atomic blocks.
- atomicity: the effects of a transaction are either fully visible or fully invisible to other threads. Partial effects are never visible, and rollback removes any trace of the executed operations.
- consistency: inside a transaction data being read is guaranteed to be in consistent state, i.e. all the invariants that an application guarantees at commit time are preserved, and they can be temporarily invalidated only by a thread's own writes. Other simultaneous transactions cannot alter them.
- isolation: inside a transaction, effects of transactions committed by other threads are not visible. They become visible only after the current transaction commits or rolls back. In database terms this is the highest possible isolation level, named "serializable".
- automatic re-execution upon conflict: if STMX detects a conflict between two transactions, it aborts and restarts at least one of them.

[1] http://common-lisp.net/project/cl-stm/
[2] http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell
[3] https://www.power.org/documentation/power-isa-transactional-memory/
[4] http://www.kurzweilai.net/ibm-announces-20-petaflops-supercomputer
[5] https://www.power.org/documentation/power-isa-version-2-07/
[6] http://common-lisp.net/project/closer/contextl.html

- read consistency: if STMX cannot guarantee that a transaction sees a consistent view of the transactional data, the whole atomic block is aborted and restarted from scratch **before** it can see the inconsistency.
- composability: multiple atomic blocks can be composed in a single, larger transaction simply by executing them from inside another atomic block.

STMX also implements the following advanced features:

- waiting for changes: if the code inside an atomic block wants to wait for changes on transactional data, it just needs to invoke `(retry)`. This will abort the transaction, sleep until another thread changes some of the transactional data read since the beginning of the atomic block, and finally re-execute it from scratch.
- nested, alternative transactions: an atomic block can execute two or more Lisp forms as alternatives in separate, nested transactions with `(atomic (orelse form1 form2 ...))`. If the first one calls `(retry)` or aborts due to a conflict or an inconsistent read, the second one will be executed and so on, until one nested transaction either commits (returns normally) or rollbacks (signals an error or condition).
- deferred execution: an atomic block can register arbitrary forms to be executed later, either immediately before or immediately after it commits.
- hybrid transactional memory: when running on 64-bit Steel Bank Common Lisp (SBCL) on a CPU with Intel TSX instructions, STMX automatically takes advantage of hardware memory transactions, while falling back on software ones in case of excessive failures. The implementation is carefully tuned and allows software and hardware transactions to run simultaneously in different threads with a constant (and very low) overhead on both transaction types. STMX currently does **not** support IBM Power ISA hardware transactions.

4. DESIGN AND IMPLEMENTATION

STMX brings efficient transactional memory to Common Lisp thanks to several design choices and extensive optimization. Design and implementation follows three research papers [6] [4] [10]. All of them contain pseudo-code for the proposed algorithms, and also include several correctness demonstrations.

Keeping the dynamically-typed spirit of Lisp, STMX is value-based: the smallest unit of transactional memory is a single cell, named `TVAR`. It behaves similarly to a variable, as it can hold a single value of any type supported by the hosting Lisp: numbers, characters, symbols, arrays, lists, functions, closures, structures, objects... A quick example:

```
(quicklisp:quickload :stmx)
(use-package :stmx)
(defvar *v* (tvar 42))
(print ($ *v*))      ;; prints 42
(atomic
  (if (oddp ($ *v*))
    (incf ($ *v*))
    (decf ($ *v*)))) ;; *v* now contains 41
```

While `TVAR`s can be used directly, it is usually more convenient to take advantage of STMX integration with `closer-mop`, a Metaobject Protocol library. This lets programmers use CLOS objects normally, while internally wrapping each slot value inside a `TVAR` to make it transactional. Thus it can also be stated that STMX is slot-based, i.e. it implements transactional memory at the granularity of a single slot inside a CLOS object. This approach introduces some space overhead, as each `TVAR` contains several other informations in addition to the value. On the other hand, it has the advantage that conflicts are detected at the granularity of a single slot: two transactions accessing different slots of the same object do not interfere with each other and can proceed in parallel. A quick CLOS-based example:

```
(transactional
  (defclass bank-account ()
    ((balance :type rational :initform 0
              :accessor account-balance))))
(defun bank-transfer (from-acct to-acct amount)
  (atomic
    (when (< (account-balance from-acct) amount)
          (error "not enough funds for transfer"))
    (decf (account-balance from-acct) amount)
    (incf (account-balance to-acct) amount)))
```

Object-based and stripe-based implementations exist too. In the former, the smallest unit of transactional memory is a single object. In the latter, the smallest unit is instead a "stripe": a (possibly non-contiguous) region of the memory address space - suitable for languages as C and C++ where pointers are first-class constructs. Both have lower overhead than slot-based transactional memory, at the price of spurious conflicts if two transactions access different slots in the same object or different addresses in the same stripe.

4.1 Read and write implementation

The fundamental operations on a `TVAR` are reading and writing its value. During a transaction, `TVAR` contents are never modified: that's performed at the end of the transaction by the commit phase. This provides the base for the atomicity and isolation guarantees. So writing into a `TVAR` must store the value somewhere else. The classic solution is to have a transaction write log: a thread-local hash table recording all writes. The hash table keys are the `TVAR`s, and the hash table values are the values to write into them.

Reading a `TVAR` is slightly more complex. Dave Dice, Ori Shalev and Nir Shavit showed in [4] how to guarantee that a transaction always sees a consistent snapshot of the `TVAR`s contents. Their solution requires versioning each `TVAR`, and also adding a "read version" to each transaction. Such version numbers are produced from a global clock. One bit of the `TVAR` version is reserved as a lock.

To actually read a `TVAR`, it is first searched in the transaction write log and, if found, the corresponding value is returned. This provides read-after-write consistency. Otherwise, the `TVAR` contents is read without acquiring any lock - first retrieving its full version (including the lock bit), then issuing a memory read barrier, retrieving its value, issuing another memory read barrier, and finally retrieving again its full version. The order is intentional, and the memory read barriers are fundamental to ensure read consistency, as they couple

with the memory write barriers used by the commit phase when actually writing TVAR contents. Then, the two TVAR versions read, including the lock bits, are compared with each other: if they differ, or if one or both lock bits are set, the transaction aborts and restarts from scratch in order to guarantee read consistency and isolation. Then, the TVAR version just read is compared with the transaction read version: if the former is larger, it means the TVAR was modified after the transaction started. In such case, the transaction aborts and restarts too. Finally, if the TVAR version is smaller than or equal to the transaction read version, the TVAR and the retrieved value are stored in the transaction read log: a thread-local hash table recording all the reads, needed by the commit phase.

4.2 Commit and abort implementation

Aborting a transaction is trivial: just discard some thread-local data - the write log, the read log and the read version.

Committing a STMX transaction works as described in [4]:

First, it acquires locks for all TVARs in the write log. Using non-blocking locks is essential to avoid deadlocks, and if some locks cannot be acquired, the whole transaction aborts and restarts from scratch. STMX uses compare-and-swap CPU instructions on the TVAR version to implement this operation (the version includes the lock bit).

Second, it checks that all TVARs in the read log are not locked by some other transaction trying to commit simultaneously, and that their version is still less or equal to the transaction read version. This guarantees the complete isolation between transactions - in database terms, transactions are "serializable". If this check fails, the whole transaction aborts and restarts from scratch.

Now the commit is guaranteed to succeed. It increases the global clock by one with an atomic-add CPU instruction, and uses the new value as the transaction write version. It then loops on all TVARs in the write log, setting their value to match what is stored in the write log, then issuing a memory write barrier, finally setting their version to the transaction write version. This last write also sets the lock bit to zero, and is used to release the previously-acquired lock.

Finally, the commit phase loops one last time on the TVARs that have been just updated. The semaphore and condition inside each TVAR will be used to notify any transaction that invoked `(retry)` and is waiting for TVARs contents to change.

4.3 Novel optimizations

In addition to the algorithm described above, STMX uses two novel optimizations to increase concurrency, and a third to reduce the overhead:

If a transaction tries to write back in a TVAR the same value read from it, the commit phase will recognize it before locking the TVAR by observing that the TVAR is associated to the same value both in the write log and in the read log. In such case, the TVAR write is degraded to a TVAR read and no lock is acquired, improving concurrency.

When actually writing value and version to a locked TVAR, the commit phase checks if it's trying to write the same value already present in the TVAR. In such case, the value and version are not updated. Keeping the old TVAR version means other transaction will not abort due to a too-large version number, improving concurrency again.

To minimize the probability of near-livelock situations, where one or more transactions repeatedly abort due to conflicts with other ones, the commit phase should acquire TVAR locks in a stable order, i.e. different transactions trying to lock the same TVARs A and B should agree whether to first lock A or B. The most general solution is to sort the TVARs before locking them, for example ordering by their address or by some serial number stored inside them. Unluckily, sorting is relatively expensive - its complexity is $O(N \log N)$ - while all other operations performed by STMX during commit are at most linear, i.e. $O(N)$ in the number of TVARs. To avoid this overhead, STMX omits the sort and replaces it with a faster alternative, at the price of increasing vulnerability to near-livelocks (crude tests performed by the author seem to show that near-livelocks remain a problem only under extreme contention). The employed solution is to store a serial number inside each TVAR and use it for the hashing algorithm used by the read log and write log hash tables. In this way, iterating on different write logs produces relatively stable answers to the question "which TVAR should be locked first, A or B ?" - especially if the hash tables have the same capacity - maintaining a low probability for near-livelock situations, without any overhead.

4.4 Automatic feature detection

ANSI Common Lisp does not offer direct access to low-level CPU instructions used by STMX, as memory barriers, compare-and-swap, and atomic-add. Among the free Lisp compilers, only Steel Bank Common Lisp (SBCL) exposes them to user programs. STMX detects the available CPU instructions at compile time, while falling back on slower, more standard features to replace any relevant CPU instruction not exposed by the host Lisp.

If memory barriers or compare-and-swap are not available, STMX inserts a `bordeaux-threads:lock` in each TVAR and uses it to lock the TVAR. The operation "check that all TVARs in the read log are not locked by some other transaction" in the commit phase requires getting the owner of a lock, or at least retrieving whether a lock is locked or not and, in case, whether the owner is the current thread. `Bordeaux-threads` does not expose such operation, but the underlying implementation often does: Clozure Common Lisp has `(ccl::%%lock-owner)`, CMUCL has `(mp::lock-process)` and Armed Bear Common Lisp allows to directly call the Java methods `ReentrantLock.isLocked()` and `ReentrantLock.isHeldByCurrentThread()` to obtain the same information. STMX detects and uses the appropriate mechanism automatically.

Similarly, the global counter uses atomic-add CPU instructions if available, otherwise it falls back on a normal add protected by a `bordeaux-threads:lock`.

4.5 Hybrid transactions

In June and July 2013 we extended STMX to support the Intel TSX CPU instructions[7], that provide hardware memory transactions.

Intel TSX actually comprise two sets of CPU instructions: HLE and RTM. Hardware Lock Elision (HLE) is designed as a compatible extension for existing code that already uses atomic compare-and-swap as locking primitive. Restricted Transactional Memory (RTM) is a new set of CPU instructions that implement hardware memory transactions directly at the CPU level:

- `XBEGIN` starts a hardware memory transaction. After this instruction and until the transaction either commits or aborts, all memory accesses are guaranteed to be transactional. The programmer must supply to `XBEGIN` the address of a fallback routine, that will be executed if the transaction aborts for any reason.
- `XEND` commits a transaction.
- `XABORT` immediately aborts a transaction and jumps to the fallback routine passed to `XBEGIN`. Note that hardware transactions can also abort spontaneosly for many different reasons: they are executed with a "best effort" policy, and while following Intel guidelines and recommendations usually results in very high success rates ($> 99.99\%$), they are **never** guaranteed to succeed and they have limits on the amount of memory that can be read and written within a transaction. Also, many operations usually cause them to abort immediately, including: conflicting memory accesses from other CPU cores, system calls, context switches, `CPUID` and `HLT` CPU instructions, etc.
- `XTEST` checks whether a transaction is in progress.

Exposing the `XBEGIN`, `XEND`, `XABORT`, and `XTEST` CPU instructions as Lisp functions and macros is non-portable but usually fairly straightforward, and we added them on 64-bit SBCL.

The real difficulty is making them compatible with software transactions: the software-based commit uses locks to prevent other threads from accessing the `TVAR`s it wants to modify, so if a hardware transaction reads those `TVAR`s at the wrong time, it would see a half-performed commit: isolation and consistency would be violated. A naive solution is to instrument hardware transactions to check whether `TVAR`s are locked or not when reading or writing them. It imposes such a large overhead that cancels the performance advantage. Another attempt is to use hardware transactions **only** to implement the commit phase of software transactions. Tests on STMX show that the performance gain is limited - about 5%.

The key was discovered by Alexander Matveev and Nir Shavit [10] in 2013: use a hardware transaction to implement the commit phase of software transactions, not to improve performance, but to make them really atomic at the CPU level. Then the software commit phase does not need anymore to lock the `TVAR`s: atomicity is now guaranteed by the hardware transaction. With such guarantees, hardware transactions can directly read and write `TVAR`s without any instrumentation - no risk of seeing a partial commit - and their overhead is now almost zero. The only remaining overhead is the need to write both `TVAR`s value and version, not just the value.

There were two problems left.

The first is: as stated above, hardware transaction are never guaranteed to succeed. They may abort if hardware limits are exceeded or if the thread attempts to execute a CPU instruction not supported inside a hardware transaction. For example, memory allocation in SBCL almost always causes hardware transactions to abort - this is an area that could be significantly improved by creating thread-local memory pools in the host Lisp.

Alexander Matveev and Nir Shavit [10] provided a sophisticated solution to this problem, with multiple levels of fallbacks: software transactions using a smaller hardware transaction during commit, software-only transactions, and intrumented hardware transactions.

We added hybrid transactions to STMX using a simplified mechanism: if the commit phase of software transactions fails (remember, it is now implemented by a hardware transaction), it increments a global counter that prevents **all** hardware transactions from running, then performs an old-style software-only commit, finally decrements the global counter to re-enable hardware transactions.

The second problem is: the commit phase of a transaction - either hardware or software - must atomically increment the global clock. For hardware transactions, this means modifying a highly contended location, causing a conflict (and an abort) as soon as two or more threads modify it from overlapping transactions.

A partial solution is described in [4, 10]: use a different global clock algorithm, named GV5, that increases the global clock **only** after an abort. It works by writing the global clock +1 into `TVAR`s during commit without increasing it, and has the side effect of causing approximately 50% of software transactions to abort.

The full solution, as described in [10, 2] is to use an adaptive global clock, named GV6, that can switch between the normal and the GV5 algorithm depending on the success and abort rates of software and hardware transactions. STMX stores these rates in thread-local variables and combines them only sporadically (every some hundred transactions) to avoid creating other highly contended global data.

We released STMX version 1.9.0 in August 2013 - the first implementation to support hybrid transactional memory in Common Lisp, and one of the first implementations to do so in any language.

4.6 Data structures

STMX includes transactional versions of basic data structures: `TCONS` and `TLIST` for cons cells and lists, `TVECTOR` for vectors, `THASH-TABLE` for hash tables, and `TMAP` for sorted maps (it is backed by a red-black tree).

[7] http://www.intel.com/software/tsx

`THASH-TABLE` and `TMAP` also have non-transactional counterparts: `GHASH-TABLE` and `GMAP`. They are provided both for completeness and as base classes for the corresponding transactional version. This makes them practical examples showing how to convert a normal data structure into a transactional one.

In many cases the conversion is trivial: change `(defclass foo ...)` definition to `(transactional (defclass foo ...))`[8]. When needed, it is also possible to decide on a slot-by-slot basis whether they should become transactional or not. This can significantly reduce the overhead in certain cases, as shown in [3]. For slots that contain non-immutable values (i.e. objects, arrays, etc.), such inner objects must also be replaced by their transactional counterparts if their contents can be modified concurrently. STMX also includes some transactional-only data structures: a first-in last-out buffer `TSTACK`, a first-in first-out buffer `TFIFO`, a reliable multicast channel `TCHANNEL`, and its reader side `TPORT`.

5. BENEFITS

The conceptual simplicity, intuitivity and correctness guarantees of transactional memory are not its only advantages.

A more subtle, important advantage is the fact that converting a data structure into its transactional version is almost completely mechanical: with STMX, it is sufficient to replace a CLOS `(defclass foo ...)` with `(transactional (defclass foo ...))`, with object-valued slots needing the same replacement.

This means that arbitrarily complex algorithms and data structures can be easily converted, without the need to analyze them in deep detail, as it's usually the case for the conversion to fine-grained lock-based concurrency. Such ability makes transactional memory best suited for exactly those algorithms and data structures that are difficult to parallelize with other paradigms: large, complex, heterogeneous data structures that can be modified concurrently by complex algorithms and do not offer easy divisions in subsets.

Clearly, analyzing the algorithms and data structures can provide benefits, in the form of insights about the subset of the data that really needs to become transactional, and which parts of the algorithms should be executed inside transactions.

A practical example is Lee's circuit routing algorithm, also used as transactional memory benchmark [1]: the algorithm takes as input a large, discrete grid and pairs of points to connect (e.g. an integrated circuit) and produces non-intersecting routes between them. Designing a lock-based concurrent version of Lee's algorithm requires decisions and trade-offs, as one has to choose at least the locking approach and the locks granularity. The transactional version is straightforward: the circuit grid becomes transactional. A deeper analysis also reveals that only a small part of the algorithm, namely backtracking, needs to be executed inside a transaction.

[8] an analogous macro for structure-objects defined with `(defstruct foo ...)` is currently under development.

6. DISADVANTAGES

Transactional memory in general has some drawbacks, and STMX inherits them.

One is easy to guess: since transactions can abort and restart at any time, they can be executed more times than expected, or they can be executed when not expected, so performing any irreversible operation inside a transaction is problematic. A typical example is input/output: a transaction should not perform it, rather it should queue the I/O operations in a transactional buffer and execute them later, from outside any transaction. Hardware transactions - at least Intel TSX - do not support any irreversible operation and will abort immediately if you try to perform input/output from them.

Another drawback is support for legacy code: to take advantage of transactions, code must use transactional cells, i.e. `TVAR`s. This requires modifications to the source code, which can be performed automatically only by transaction-aware compilers or by instrumentation libraries as Java Deuce [9]. STMX is implemented as a normal library, not as a compiler plugin, so it requires programmers to adapt their code. The modifications are quite simple and mechanic, and STMX includes transactional versions of some popular data structures, both as ready-to-use solutions and as examples and tutorials showing how to modify a data structure to make it transactional.

The last disadvantage is proneness to almost-livelocks under high contention. This is common to all implementations that use non-blocking mutexes (STMX uses compare-and-swap ones) as synchronization primitives, as they either succeed or fail immediately, and they are not able nor supposed to sleep until the mutex can be acquired: doing so would cause deadlocks.

7. TRANSACTIONAL I/O

We present a novel result, showing that in a very specific case it is possible to perform I/O from a hardware transaction implemented by Intel TSX, working around the current Intel hardware limitations. The result is transactional output, i.e. the output is performed if and only if the hardware transaction commits.

Intel reference documentation[9] states that attempting to execute I/O from an Intel TSX transactions **may** cause it to abort immediately, and that the exact behavior is implementation-dependent. On the hardware tested by the author (Intel Core i7 4770) this is indeed the case: syscalls, context switches, I/O to hardware ports, and the other operations that "**may** abort transactions", actually abort them. The technique described below works around this limitation.

Hardware transactions are guaranteed to support only manipulation of CPU registers and memory. Anyway, the content and meaning of the memory is irrelevant for Intel TSX. It is thus possible to write to memory-mapped files or shared memory, as long as doing so does not immediately trigger a context switch or a page fault.

[9] http://download-software.intel.com/sites/default/files/319433-014.pdf - section 8.3.8.1, pages 391-392

Thus, if some pages of memory mapped file are already dirty - for example because we write into them from outside any transaction - it is possible to continue writing into them from hardware transactions. After some time, the kernel will spontaneously perform a context switch and write back the pages to disk. Since hardware transactions are atomic at the CPU level and they currently abort upon a context switch, the kernel will observe that some of them have committed and altered the pages, while some others have aborted and their effects are completely rolled back. The memory pages, altered only by the committed transactions, will be written to disk by the kernel, thus implementing transactional I/O.

Author's initial tests show that it is possible to reach very high percentages of successful hardware transactions - more than 99% - writing to memory mapped files, provided the transactions are short and there is code to dirty again the pages if the hardware transactions fail.

This is a workaround - maybe even a hack - yet it is extremely useful to implement database-like workloads, where transactions must also be persistent, and shared memory inter-process communication. The author is currently using this technique to implement Hyperluminal-DB[10], a transactional and persistent object store, on top of STMX.

8. PERFORMANCE

This paragraph contains benchmark results obtained on an Intel Core i7 4770, running 64-bit versions of Linux/Debian jessie, SBCL 1.1.15 and the latest STMX. **Disclaimer**: results on different systems **will** vary. Speed differences up to **100 times and more** have been observed, depending on the Lisp compiler and the support for features used by STMX. System setup: execute the forms

```
(declaim (optimize (compilation-speed 0) (space 0)
                   (debug 0) (safety 0) (speed 3)))
(ql:quickload "stmx")
(ql:quickload "stmx.test")
(fiveam:run! 'stmx.test:suite)
```

before loading any other Lisp library, to set optimization strategy, load STMX and its dependencies, and run the test suite once to warm up the system.

8.1 Micro-benchmarks

We then created some transactional objects: a `TVAR` v, a `TMAP` tm, a `THASH-TABLE` th and fill them - full details are described in STMX source code[11]. Note that `TMAP` and `THASH-TABLE` are CLOS objects, making the implementation short and (usually) clear but not heavily optimized for speed. Rewriting them as structure-objects would definitely improve their performance. Finally, `$` is the function to read and write `TVAR` contents.

To record the execution time, we repeated each benchmark one million times in a loop and divided the resulting time by the number of iterations.

In Table 1, we report three times for each micro-benchmark: the first for software-only transactions, the second for hybrid transactions, the third for non-transactional execution with non-transactional data structures.

Table 1: micro-benchmarks time, in nanoseconds

Name	Code	SW tx	hybrid	no tx
read	`($ v)`	87	22	< 1
write	`(setf ($ v) 1)`	113	27	< 1
incf	`(incf ($ v))`	148	27	3
10 incf	`(dotimes (j 10) (incf (the fixnum ($ v))))`	272	59	19
100 incf	`(dotimes (j 100) (incf (the fixnum ($ v))))`	1399	409	193
1000 incf	`(dotimes (j 1000) (incf (the fixnum ($ v))))`	12676	3852	1939
map read	`(get-gmap tm 1)`	274	175	51
map incf	`(incf (get-gmap tm 1))`	556	419	117
hash read	`(get-ghash th 1)`	303	215	74
hash incf	`(incf (get-ghash th 1))`	674	525	168

Some remarks and deductions on the micro-benchmarks results: STMX software-only transactions have an initial overhead of ~ 130 nanoseconds, and hybrid transactions reduce the overhead to ~ 25 nanoseconds.

In software-only transactions, reading and writing `TVAR`s, i.e. transactional memory, is 6–7 times slower than reading and writing normal memory. Hardware transactions improve the situation: inside them, transactional memory is twice as slow as normal memory. In this respect, it is worth noting that STMX can be further optimized, since in pure hardware transactions (which do not use `TVAR`s nor the function `$`) reading and writing memory has practically the same speed as normal memory access outside transactions.

The results on CLOS sorted maps and hash tables show that they are relatively slow, and the transactional version even more so. To have a more detailed picture, non-CLOS implementations of sorted maps and hash tables would be needed for comparison.

8.2 Lee-TM

Finding or designing a good synthetic benchmark for transactional memory is not easy. Lee's circuit routing algorithm, in the proposers' opinion [1], is a more realistic benchmark than classic ones (red-black trees and other micro-benchmarks, STMBench7 ...). It takes as input a large, discrete grid and pairs of points to connect (e.g. an integrated circuit) and produces non-intersecting routes between them. Proposed and used as benchmark for many transactional memory implementations (TL2, TinySTM, RSTM, SwissTM ...), it features longer transactions and non-trivial data contention.

After porting Lee-TM to STMX[12], we realized that it spends about 99.5% of the CPU time `outside` transactions due

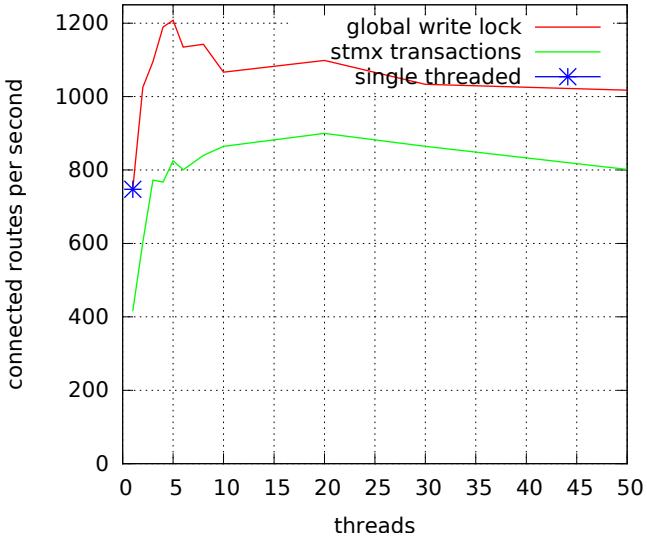
[10] https://github.com/cosmos72/hyperluminal-db
[11] http://github.com/cosmos72/stmx
[12] https://github.com/cosmos72/lee-stmx

to the (intentionally naive) grid exploration algorithm, and 0.5% in the backtracking algorithm (executed inside a transaction). It is thus not really representative of the strength and weaknesses of transactional memory. Lacking a better alternative we present it nevertheless, after some optimizations (we replaced Lee's algorithm with faster Hadlock's one) that reduce the CPU time spent outside transactions to 92–94%. The effect is that Lee-TM accurately shows the overhead of reading transactional memory from **outside** transactions, but is not very sensitive to transactional behavior.

In Table 2, we compare the transactional implementation of Lee-TM with a single-thread version and with a simple lock-based version that uses one global write lock. The results show that transactional memory slows down Lee's algorithm (actually, Hadlock's algorithm) by approximately 20% without altering its scalability.

The global write lock is a particularly good choice for this benchmark due to the very low time spent holding it (6–8%), and because the algorithm can tolerate lock-free reads from the shared grid. Yet, the overhead of the much more general transactional memory approach is contained. More balanced or more complex algorithms would highlight the poor scalability of trying to parallelize using a simple global write lock.

Table 2: Lee-TM, mainboard circuit



9. CONCLUSIONS

Transactional memory has a long history. Mostly confined to a research topic for the first two decades, it is now finding its way into high-quality implementations at an accelerating pace. The long-sought arrival of hardware support may well be the last piece needed for wide diffusion as a concurrent programming paradigm.

STMX brings state of the art, high-performance transactional memory to Common Lisp. It is one of the first publicly available implementations to support hybrid transactions, integrating "Intel TSX" CPU instructions and software transactions with minimal overhead on both.

STMX is freely available: licensed under the "Lisp Lesser General Public Licence" (LLGPL), it can be installed with Quicklisp `(ql:quickload "stmx")` or downloaded from `http://stmx.org/`

10. REFERENCES

[1] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-TM: A non-trivial benchmark suite for transactional memory. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '08, pages 196–207, 2008.

[2] H. Avni. A transactional consistency clock defined and optimized. Master's thesis, Tel-Aviv University, 2009. http://mcg.cs.tau.ac.il/papers/hillel-avni-msc.pdf.

[3] F. M. Carvalho and J. Cachopo. STM with transparent API considered harmful. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'11, pages 326–337, Berlin, Heidelberg, 2011.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC'06 Proceedings of the 20th international conference on Distributed Computing*, pages 194–208. Sun Microsystems Laboratories, Burlington, MA, 2006.

[5] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Publishing Company, Boston, Massachusetts, 2006.

[6] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05 Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. Microsoft Research, Cambridge, UK, 2005.

[7] G. Karam and R. Buhr. Starvation and critical race analyzers for Ada. *IEEE Transactions on Software Engineering*, 16(8):829–843, August 1990.

[8] T. Knight. An architecture for mostly functional languages. In *LFP '86 Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112. Symbolics, Inc., and The M.I.T. Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1986.

[9] G. Korland, N. Shavit, and P. Felber. Noninvasive Java concurrency with Deuce STM. In *Proceedings of the Israeli Experimental Systems Conference*, SYSTOR'09. Tel-Aviv University, Israel and University of Neuchâtel, Switzerland, 2009.

[10] A. Matveev and N. Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *SPAA '13 Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 11–22. MIT, Boston, MA and Tel-Aviv University, Israel, 2013.

[11] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95 Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. MIT and Tel-Aviv University, 1995.

[12] D. A. Wheeler. Secure programming for Linux and Unix HOWTO. http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/avoid-race.html, 1991.

A functional approach for disruptive event discovery and policy monitoring in mobility scenarios

Ignasi Gómez-Sebastià
Universitat Politécnica de Catalunya - Barcelona Tech
igomez@lsi.upc.edu

Luis Oliva-Felipe
Universitat Politécnica de Catalunya - Barcelona Tech
loliva@lsi.upc.edu

Sergio Alvarez-Napagao
Universitat Politécnica de Catalunya - Barcelona Tech
salvarez@lsi.upc.edu

Dario Garcia-Gasulla
Universitat Politécnica de Catalunya - Barcelona Tech
dariog@lsi.upc.edu

Arturo Tejeda-Gómez
Universitat Politécnica de Catalunya - Barcelona Tech
jatejeda@lsi.upc.edu

Javier Vázquez-Salceda
Universitat Politécnica de Catalunya - Barcelona Tech
jvazquez@lsi.upc.edu

ABSTRACT

This paper presents the results obtained of using LISP for real-time event detection and how these results are interpreted and used within the context of SUPERHUB, a European-Funded projected aimed to achieve a more sustainable mobility behaviour in cities. Real-time detection allows faster reaction for decision making processes as well as a valuable asset for policy makers to know what should be done when an unexpected event occurs. The use of LISP has facilitated most of this process and, specially, supported to parallelize the capture, aggregation and interpretation of data.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

Smart cities, Clojure, Event Detection, Policy evaluation

1. INTRODUCTION

Mobility is one of the main challenges for urban planners in the cities. Even with the constant technological progress, it is still difficult for policy makers and transport operators to 1) know the state of the city in (near) real-time, and 2) achieve proximity with the end-user of such city services, especially with regards to communicating with the citizen and receiving proper feedback.

There is a relatively recent technological advance that enables an opportunity to partially tackle these issues: ubiquitous computational resources. For instance, thanks to smartphones, users that move in a city can potentially generate automatic data that may be hard to obtain otherwise: location, movement flow, average trip times, and so on[1]. Moreover, transport network problems and incidents that affect mobility services are often documented by someone somewhere in the Internet at the same time or even before, in many cases, than they will appear in the official sources or in the news media.

This phenomenon has been referred to as *humans as sensors* [11]. Sensing through mobile humans potentially provides sensor coverage where events are taking place. An additional benefit is that human expertise can be used to operate such sensors to raise the quality of measurements, through e.g. a more intelligent decision making, such as setting up a camera in an optimal way in poor lighting conditions; or providing exploitable additional metadata, as in collaborative tagging processes such as hashtagging.

In this paper, we show a system that is able to mine such data in order to:

1. improve knowledge obtained from other data generation approaches, such as GPS pattern analysis, and
2. detect unexpected situations in the city that may affect large groups of people at a certain location, e.g. public demonstrations or celebrations or sudden traffic jams caused by accidents.
3. enable services to users that exploit such generated knowledge, providing novel kinds of real-time information and recommendation.

The paper presents, due to space constraints, just a general overview of the problems we tackle, the preliminary results of the parts already implemented, and the future work. For deeper reports on the technical details, please refer to the related deliverables[2] and to [6].

This paper is structured as follows: in §2 we introduce SUPERHUB, an urban mobility-related EU project[3] and §3 motivates our use of a LISP dialect ; §4 contains an explanation of the extent of the contextual detection, focusing on social network data; §5 explains the policy monitoring and optimization procedures; and finally §6 presents related work and wraps up the paper with conclusions.

[1] Please, notice all data is anonymized and sensitive personal data is not stored neither gathered under any circumstances.

[2] `http://www.superhub-project.eu/downloads/viewcategory/6-approved-deliverables.html`

[3] This work has been supported by the EU project ICT-FP7-289067 SUPERHUB.

2. THE SUPERHUB PROJECT

SUPERHUB [3] is a project co-funded by the European Commission. Its main goal is to provide an open platform capable of considering in real time various mobility offers, in order to provide a set of mobility services able to address users' needs. At the same time the project intends to promote user participation and environmental friendly and energy-efficient behaviours.

The project builds on the notion that citizens are not just mere users of mobility services, but represent an active component and a resource for policy-makers willing to improve sustainable mobility in smart cities. Existing journey planners only provide a few options to let users customize, to some extent, how the journey should look like. The reality, however, is more nuanced – different users might prefer different routes which, in addition, depend on the user's context (*e.g.* , a shopping trip, travelling with small children or going back home) as well as on the environmental context: weather, traffic, crowdedness, events, *etc.* .

SUPERHUB will provide an open platform, through which users shall inquire for possible mobility options to reach a given destination at any given time. The back-end system replies providing a rich set of possible options and recommendations taking into account a number of mobility solutions. The possible options are ranked based on the preferences elaborated within the user's profile, which includes information such as the perceived importance of the environmental impact, the willingness to walk/cycle in rainy weather etc. After the choice is made by the user, the system will track and guide the user throughout her/his journey and will constantly offer, at run time, new options/suggestions to improve the service experience, for example assisting her/him in the search of the nearest parking lot or providing her/him additional and customised information services such as pollutions maps.

To achieve these objectives SUPERHUB is developing, but not limited to:

1. Novel methods and tools for event detection via real-time reasoning on large data streams coming from heterogeneous sources.
2. New algorithms and protocols for inferring traffic conditions from mobile users, by coupling data from mobile phone networks with information coming from both GPS data and social network streams.
3. A policy monitor component to put in contrast the political reality designed by policy-makers with the social reality represent by the actual state of the city *w.r.t.* mobility.
4. A policy optimizer for analysing and suggesting improvements to the policies in places.

In this paper, we focus on two specific components of the SUPERHUB project in which the authors have been involved: the event detection mechanism and the policy framework.

The Disruptive Event Detector is a component that provides knowledge in the form of RDF triples inferred from sensor data (esp. social network data) that is of a higher level of abstraction than what is usually obtained with other techniques, acting as a central point for data homogenization. Via this component, raw data is filtered, normalised and interpreted into high-level concepts. Such concepts can be merged and analysed to generate derivative concepts that are not explicit in the sensor data but implicit in the aggregation of large instances of it.

The Policy Framework (including the policy monitor and optimizer among other sub-components) combines real-time information analysis with state of the art Urban Mobility Decision Support Systems, accomplishing two main objectives. First, helping policy makers design the actions that will improve urban mobility via the policy optimizer component. Second, real-time assessment of the state of the city, via the policy monitor, in order to find out when to apply such actions.

3. CHOOSING A LANGUAGE: CLOJURE

The following are the requirements and constraints that affected the decision on which language to use to implement the systems described in §2:

1. Due to project-wide constraints, they have to be run on top of a Java Virtual Machine (JVM), and use Java code for easy integration with the rest of the components.
2. Facilitate working with maps and vectors, and especially with JSON data.
3. Shared data models have been in continuous change, so it was important to be able to change internal data models as simply as possible and to be able to use legacy versions at the same time as new ones.
4. Transparent support for immutable structures, for working concurrently with several city sensor inputs and different policies to be evaluated in parallel.
5. Native, and easy to use, support for multi-core architectures where several threads run in parallel.
6. First class functions. Policies are self-contained, typically expressing computation and data aggregation methods inside the fields. Even when these methods are expressed as part of the policy (data) they should be transparently used for evaluating it (*i.e.* used as code).
7. A runtime as dynamic as possible, as any kind of downtime, esp. change in code, can reflect on gaps on the data collected from the sensors.

Considering these requirements, Clojure has been a natural choice. Due to its rigid type system, the difficulty of working with reflection, and the lack of a high-level concurrency framework, Java was discarded.

On the other hand, object orientation has never been a needed feature, which added to the fact that continuously evolving data models were not encouraging us to use type systems, Clojure was finally chosen over Scala and Groovy. The use of Clojure protocols has been enough to cover types on the few cases in which this was needed.

Additionally, Clojure is the only JVM language that allows us to handle self-contained policies as *code as data* (or *data as code*), effectively enabling us to remove one tier of processing between definition and execution.

4. DISRUPTIVE EVENT DETECTION

Event detection based on social networks is a topic of recent interest. Since the data provided by social networks is large in volume and can be accessed in real time through available APIs, predicting either expected or unexpected events in which a significant amount of people is involved becomes feasible. For that reason, all approaches within the state-of-the-art follow the *human as a sensor* approach for data collection.

In [9], authors identify earthquakes, typhoons and traffic jams based on tweets from Twitter. They estimate the location and trajectory of those target events and model three main characteristics:

1. Scale of event - many users experience the event
2. Impact of event - they affect people's life and, therefore, their behaviour
3. Location of event - they take place in a spatial and temporal regions

Authors define an *event* as an arbitrary classification of a space-time region. Their temporal model assumes that users messages represent an exponential distribution since users post the most after a given critical time. Their spatial model assumes messages are geo-located, and uses Bayesian filters to identify location and trajectory of the event.

In [14], authors classify event detection algorithms into two categories: document-pivot methods and feature-pivot methods. The former is based on clustering documents, according to a semantic distance. The latter on clustering words together, instead of documents. Authors focus on feature-pivot methods to propose an event detection algorithm based on clustering Wavelet-based signals. Wavelet analysis shows when and how the frequency of a signal changes over time. Their event detection algorithm builds signals for single words and captures only the bursts ones to measure the cross-correlation between signals. The detection arises when they cluster signals together by modularity-base graph partitioning.

A different approach to event detection is that of summarizing long-running structure-rich events [4]. Authors assume that, when a new event is detected, the immediate goal is to extract the information that best describes the chain of interesting occurrences explaining the event. In order to carry out this process, previous experience is needed (*i.e.* repeated events). A modified Hidden Markov Model is integrated with the event time-line, based on the stream of tweets and their word distribution. By splitting the time-line, a set of sub-events is found each of which describing a portion of the full event.

Finally, in [8] the TEDAS system is proposed for online and offline analysis. The online solves analytical queries and generates visual results to rank tweets and extract patterns for the query based on a clustering model; the offline one retrieves tweets, classifies them and stores statements (events). Another proposed application, Tweevent, was presented in [7]. It retrieves tweets from the Twitter stream, and segments each tweet in a sequence of consecutive phrases, every segment being modelled as a Gaussian distribution based on its frequency. Tweevent applies a clustering algorithm to group segments; each segment representing a detected event.

All these proposals have in common that they consider internal features of the social network activity (in most cases, the text of tweets) in order to build their model. As a result, the main challenges they must face are frequently related with Natural Language Processing (*e.g.* solving ambiguities, identifying stop-words, removing emoticons, handling hashtags, etc.). Our proposal is quite different in that it does not consider features of social network activities, only the existence of the activity itself. As we will see next, this allows us to to model the city and predict unexpected events in a simple manner, although we cannot determine their specific nature.

4.1 Data Sources

Social networks are among the most frequently used applications in smart mobile devices. One of the main benefits of social network applications (from the data scientist point of view) is that people use them at all times, everywhere. Some of these social networks offer APIs which can be queried for user information (information which users have previously accepted to provide). The Twitter API allows to query Tweets generated in a given area. Foursquare provides information about where users have *checked in* (where they have marked as currently being at) at a given time. Finally, the third large social network with available API is Instagram, providing access to the uploaded pictures, the text added by users to those pictures, and the location where the upload took place. These three social networks will be our main sources of data.

The information provided by these social networks has a very small granularity and size, which complicates its atomic interpretation. The semantics of each tweet are hard to understand through Natural Language Processing due to their short nature, the use of slang, hashtags, *etc.* Foursquare is smaller (*i.e.* it has less activity) and its data is biased towards points of interest. Data from Instagram is also biased towards sight-seeing locations. Unfortunately, the computational cost of performing image recognition on each photo is prohibitive. With these restrictions, an attempt to analyse their combined data in detail will result in multiple problems of high complexity. However, if one focuses on their most basic shared properties (*e.g.* a happening, and its time and location) and discards the most context dependent properties (*e.g.* text, images, categories) the problem becomes much simpler, while its potential benefits remain remarkable. And most important, by simplifying the problem we are increasing the amount of the dataset (Twitter + Foursquare + Instagram). We focus on atomic data points representing single activities in a place and time, normalizing tweets, check-ins and image uploads to consider only when and where they happen. We aggregate them into a single model and obtain a combined data source with millions of real-time sensors of a wider spectrum.

One of the important theorems for Big Data is the law of large numbers, which states that a large sample will converge to a normalized average as the sample grows. In that regard we produce a larger and broader sample than the related work by combining three social networks. This makes our model resistant to variations and outliers. In a domain as volatile as the behaviour of a large city (*i.e.* the semi-coordinated and combined behaviour of millions of people), these features will be a valuable asset.

4.2 Data Properties

The quest for simple data made us focus on the most basic features available from the data sources. Concretely we work only with *when* and *where* a social network action takes place.

These two properties are shared by the three data sources, which allows us to easily aggregate their information. Further information would also enrich the model and enhance the capabilities of the system (for example by trying to detect the semantics of events based on the text), which is why we consider using it in the future as a second step process. However, from the perspective of this paper we work only in terms of time and space, to demonstrate what can be done with such a simple, but vast, input.

Due to the fact that all the data sources that we currently use provide data in JSON format, Clojure gives us a noticeable advantage over any other language thanks to the immediate mapping from JSON to Clojure nested associative maps by the use of the *data.json* library. Handling data directly as associative maps allows for straightforward handling and mapping to and from data structures, as in the following example that is used to convert tweets into our normalized data structure:

```
(defn map-from-twitter
  "Maps␣a␣Tweet␣structure␣into␣an␣ad␣hoc␣one."
  [tweet]
  (hash-map
    :_id (str "twitter-" (:id_str tweet))
    :lat (second (:coordinates (:coordinates tweet)))
    :lng (first (:coordinates (:coordinates tweet)))
    :geometry {:lat (second (:coordinates (:coordinates tweet)))
               :lng (first (:coordinates (:coordinates tweet)))}
    :ts (int (/ (.getTime (.parse
                           th/date-formatter-twitter
                           (:created_at tweet))) 1000))
    :tags (:hashtags (:entities tweet))
    :urls (:urls (:entities tweet))
    :mentions (:user_mentions (:entities tweet))
    :venue-id (:id (:place tweet))
    :text (:text tweet)
    :count 1
    :lang (:lang tweet)
    :user (:id_str (:user tweet))
    :user-twitter (:user tweet)
    :favorite-count (:favorite_count tweet)
    :retweet-count (:retweet_count tweet)
    :in-reply-to-status-id (:in_reply_to_status_id_str tweet)
    :in-reply-to-user-id (:in_reply_to_user_id_str tweet)
    :app "twitter"))
```

An advantage of defining such simple mappings between data models allows us to modify the mappings whenever it is needed, adding or removing particular data from the sources in execution time. In our case, this has been really important due to the fact that we have been able to use, for our training sets, data collected since the beginning of our deployment even when our mappings have been evolving over time.

The small granularity of the information available forces us to characterize events in a simple manner. Of the two main features we handle, time and space, we will only use time dynamically, to represent the persistence of events through time. This allows us to study how events are created and how they fade away over time. Space could also be used dynamically, and it would allow us to introduce the notion of *travelling* event (*e.g.* a demonstration which moves), a topic which we intend to tackle in the future. However for the scope of this paper we will only model events dynamically through time. Space will be used statically.

4.2.1 Handling of time

To represent time we split it in 15 minutes portions (what we call *time-windows*). This time length is a trade-off between the event granularity we wish to detect (any event shorter than 15 minutes is not worth detecting for us) and a minimum temporal gap which would guarantee an stable data input (shorter portions would result in more variable data). We starting collecting data in July 2013, and at the time of writing this paper we are still collecting it. That roughly amounts to 22,000 non-overlapping time-windows.

There is a set of predicates that can be applied to each timestamp of T in order to retrieve its calendar information: *month*, *day* (of the month), *weekday*, *hour* (of the day), and *minute* (of the hour):

```
(defn get-date-fields [ts]
  (let [millis (* 1000 ts)
        day (.format cet-date-format millis)
        t (.format cet-time-format millis)
        day-tokens (clojure.string/split day #"[\s|,]")
        year (java.lang.Integer/parseInt (last day-tokens))
        weekday (first day-tokens)
        day (java.lang.Integer/parseInt (nth day-tokens 2))
        month (nth day-tokens 3)
        time-tokens (clojure.string/split t #":")
        hour (java.lang.Integer/parseInt (first time-tokens))
        minute (java.lang.Integer/parseInt (second time-tokens))]
    {:weekday weekday
     :month month
     :day day
     :hour hour
     :minute minute
     :year year}))
```

Analogously, each of these predicates can be applied to a *time-window*, retrieving the information corresponding to its initial timestamp. To compare the measurements occurring in a specific *time-window* with respect to historical data, we use the history of measurements having occurred in the correspondent time-window for all the other weeks. Therefore, two *time-windows* will be comparable if they share the same weekday, hour and minute, which is true whenever the result of *abstract-interval* below is the same for two given timestamps:

```
(defn abstract-interval [ts]
  (let [jt (timec/from-long (* 1000 ts))
        hour (time/hour jt)
        minute (time/minute jt)
        weekday (["" "Monday" "Tuesday" "Wednesday"
                  "Thursday" "Friday" "Saturday" "Sunday"]
                  (time/day-of-week jt))]
    {:hour hour :minute minute :weekday weekday :interval ts}))
```

4.2.2 Handling of space

Regarding our representation of space, we focus on the Barcelona metropolitan area, with an area of $633km^2$ and a population of approximately 3.2 million people. We split the city into sectors based on geohashes, a hierarchical representation of space in the form of a recursive grid [5]. Sectors are defined by geohashes of n characters, where $1 \leq n < 12$ (a geohash of 12 characters represents a one-dimensional coordinate). This allows to split geographical locations in non-overlapping sectors of equal area. We decided to work with geohashes six characters long, roughly representing $0.55km^2$ in the area of Barcelona. As a result we have over 2,000 land sectors with relevant data (we consider an slightly bigger area than that of the Barcelona metropolitan area).

By splitting Barcelona and its surrounding cities in equally seized sectors, obtaining data for each of those sectors every 15 minutes for over seven months, and combining all that data, we build an approximate model for the whole city. Aggregations are made by mapping coordinates to sectors, each coordinate c defined by a pair of floating point numbers and representing Earth latitude and longitude. As a result we produce a separate behavioural model for each sector and weekday using the social network activity taking place in it.

4.3 Data Aggregation and Deviation Detection

The main data aggregation process of our system computes the crowd density in a specific pair of $< sector, interval >$ by gathering all data taking place in that time and location. For

a given *time-window* corresponding to a specific period of 15 minutes or 900 seconds and a given sector, we generate several values that correspond to: 1) the set of all the raw data obtained during the time-window geolocated inside the sector, 2) the set of sources that pushed data during such time-window, regardless of the sector, 3) the sum of all the unique *users* for each *source* obtained by push for the given sector, and 4)t he aggregation of all the values of the property *count* (always 1 in the case of Twitter and Instagram, ≥ 0 in the case of Foursquare).

In order to make predictions we need to train the system with already collected data. We assume that not all *sources* have been actively polled during the full expanse of the crawling, so we need a mechanism to consider which aggregations are statistically valid. We will only consider valid those aggregations done for a time-window in which there is a minimum value of 1 for each of the sums of the full historical set of *sources*.

In order to define *normality* as a metric of what can be expected of sensor data in a certain time-window, we split the week into all of its possible distinct *time-windows* (w, h, m denote *weekday*, *hour* and *minute* respectively).

Therefore, for each combination of *weekday*, *hour* and *minute*, for a certain training set for a specific sector, a function *statistics-area-interval* returns all the aggregation sums. Taking this set as basis, we can thus infer normality by using the *median* and the *inter-quartile range* [$iqr(set) = q_3(set) - q_1(set)$] [13]:

```
all-values (map :sum new-val)
qs (quantile all-values)
q1 (second qs)
q3 (nth qs 3)
iq (- q3 q1)
upper-distance (* 3 iq)
upper-inner-fence (+ q3 lower-distance)
upper-outer-fence (+ q3 upper-distance)
```

Given this set of statistical measures, we can now define three predicates over an aggregation: *normal*, *deviated* and *abnormal*, each of them denoting a different degree of *normality* with respect to a training set:

```
(defn statistics-area-interval [area ts]
  (let [agg (aggregate-by-weekday-area area)
        interval (th/abstract-interval ts)
        agg-by-hour-minutes (group-by :interval agg)]
    (let [values (get agg-by-hour-minutes interval)
          hour-minute (* 3600000
                         (+ (:hour interval)
                         (/ (:minute interval) 60)))
          correct-value (first
                          (filter
                            (fn [a]
                              (= (:weekday a)
                                 (:weekday interval)))
                            (get agg-by-hour-minutes hour-minute)))]
      (if (nil? correct-value)
        nil
        {:interval (* 1000 (:interval interval))
         :normal (:median correct-value)
         :deviated (:upper-inner-fence correct-value)
         :abnormal (:upper-outer-fence correct-value)}))))
```

We detect an event in a certain sector during a set of time-windows, when all of the time-windows correspond to deviated sensor data, and at least one of those corresponds to abnormal sensor data.

4.4 Event Representation

Once we have defined our model of city behaviour and how we detect events in it, now we introduce our model of events. We represent events as pseudo-trapezoids where the horizontal axis represents time and the vertical axis certainty. In this representation one can imagine an event which is slowly forming as a trapezoid beginning with a gentle ascending slope until reaching certainty 1, and an event which abruptly ends as a trapezoid ending with a strong descending slope (see Figure 1). The segment in the horizontal axis in which the trapezoid bottom and top lines are parallel would represent the temporal gap in which the event is happening with full certainty (*i.e.* abnormal status). In this representation the low base must be at least as large as the top base, and must include it *w.r.t.* the horizontal axis. In the example of Figure 1 the lower base goes from t1 to t11, and the top base from t5 to t10.

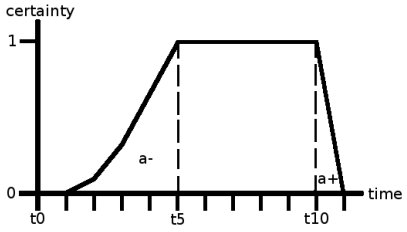


Figure 1: Trapezoid representation of an event. Its beginning goes from t1 to t5. Its ending from t10 to t11.

In the trapezoid representation we can identify three main components in each event: Its beginning, its body and its ending. The beginning of an event a is called a^-, while the ending is called a^+ (see Figure 1). Both these components represent a deviated status as previously defined. The body section represents an abnormal status. This decomposition of events into beginning, body and ending is frequent in the bibliography and allows temporal reasoning based on their properties [1, 10]. In order to implement this algorithm, we make intensive use of functions as first-class citizens of the language. Each of the thirteen temporal reasoning predicates is implemented as a boolean function such as:

```
(defn A-before-B
  [A B]
  (subint<subint
    (get-ending-of-interval A)
    (get-beginning-of-interval B)))

(defn A-after-B
  [A B]
  (A-before-B B A))
```

To obtain all of the values for a pair of events, the thirteen functions are then statically defined in a vector and iterated with a *map* invocation:

```
(defn get-all-AB-temporal-relations
  [A B]
  (let [fns [A-before-B A-after-B A-overlaps-B
             A-overlapped_by-B A-during-B
             A-contains-B A-meets-B A-met_by-B
             A-starts-B A-started_by-B A-equals-B
             A-finishes-B A-finished_by-B]
        all-rels (zipmap
                   [:before :after :overlaps :overlapped_by
                    :during :contains :meets :met_by :starts
                    :started_by :finishes :finished_by :equals]
                   (map #(% A B) fns))]
    all-rels))
```

4.5 Experimentation

There are multiple ways of exploiting the huge amount of available data. In here we will focus on how event detection performs, and the profile of the detected events. We begin by showing an example of detected event and its representation in our model, to help illustrate the approach. Afterwards we

will show the potential of the methodology by showing the most relevant events detected in the period of data captured and some other interesting events we have identified. That will motivate a later study on the potential applications of the approach.

4.5.1 Event Model

To illustrate the methodology, this section presents an event detected the 20th of November, 2013 in the *sp3e37* geohash sector (see Figure 2). The event was later manually identified as a concert of the popular band Mishima in the city of Barcelona. We also learned that a warm-up band started playing at 20:00 while the main concert started at 21:00.

As shown in Figure 2, the data retrieved for this event started being relevant at 19:45, in coherency with the starting of the warm-up concert. As people arrive at the venue, the social network activity becomes more dense than normal which triggers the event detection (deviated status). The event reached full certainty (abnormal status) at 20:15, when the warm-up concert was already going on, and shortly before the main concert began. At that point the event would remain certain (with a minor fall at 20:30) long after the end of the main concert, until it finally disappeared at 00:45.

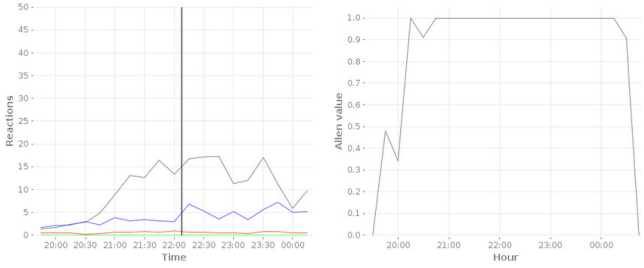


Figure 2: Model of a concert event. Left image represents the social network activity through time (Top line is the event activity, middle line is the upperfence for event detection). Right image shows its trapezoid representation, with time on the x axis and certainty on the y axis.

The beginning of this event, between 19:15 and 20:15, represents the gradual arrival of people to the venue, and is coherent with the idea that people arrive to a concert between 2 hours and half an hour before it begins. The duration of the event itself, from 20:15 to 00:15, includes the whole duration of the concert with an additional margin on both sides. This most likely represents the people staying around the concert area shortly after the concert has ended. Finally, the end of the event is more sudden than its beginning (it goes from 00:15 to 00:45). This strong descending slope can be understood as the quick dispersion of the people who went to the concert. This example allows us to motivate an interesting application of the methodology, as the model of events can be used to profile them and help understand their nature. In this case we could argue that concerts tend to attract people gradually, while their dispersion afterwards is much faster.

4.5.2 Massive Event Discovery

To further analyse the capabilities of the event discovery approach, we now focus on the full potentiality of the system. At the time of writing this paper our system has detected 712 events. These, in the 229 days of captured data correspond to 3.1 events per day. We decided to study the most relevant events detected in the time gap captured, understanding relevance as the certainty and length of the event's occurrence. From the top 40 events captured, 15 are Football Club Barcelona (FCB) games (the most popular sport team in the city), one being of its basketball section. To evaluate the social impact of one of those games consider that the FCB stadium has a capacity for over 98,000 people. The second most popular sport team in the city, Real Club Deportiu Espanyol (RCDE), caused 2 events in the top 40. This club has significantly fewer supporters and its stadium has capacity for 40,500 people. Within the top 40 there are also 10 concerts, 5 events related with New Year's Day and Christmas, 5 events in the Barcelona airport, 2 events in popular locations and 1 event associated with an important and yearly fashion event in the city of Barcelona.

The ranking of football game events found within the top 40 seem to correlate with the idea that the popularity of the game is associated with its impact on the model. The top ranked game is against Real Madrid, the arch-rivals of FCB. Champions League games (the most important international competition FCB plays) are also highly ranked (*vs.* Milan, *vs.* Celtic) on average. The correlation between popularity and impact is similarly found in concerts, which events rank based on the popularity of the performer. The first concert however corresponds to a day in which three concerts took place in the same area at the same time. Next come concerts which had an isolated location but which had very popular performers (Michael Buble, Arctic Monkeys, Bruno Mars, Depeche Mode, *etc.*). It is also relevant to see how New Year's day is a source of huge events, something which is coherent with the specially active behaviour of the city during that day.

Beyond the top 40 there are other interesting events which we have identified within the total of 712 detected. Next we list some of them to illustrate the wide capabilities of the system. The variety in the nature, target audience and impact of the detected events suggests that our approach can be used in a wide variety of contexts:

1) The iPhone 5 release date caused an event in the area of the Apple store. 2) A 3 day special sale for movie theatre tickets (half price) city wide, caused events in several movie theatres. 3) A strike in the train service caused events in the main stations. 4) The re-opening day of an old market caused events in the market area. 5) Congresses such as ERS congress, Smart Cities congress and others, caused events in the venue areas. 6) Barcelona shopping night, an event organized by Barcelona merchants caused events in several commercial areas.

As a final experiment, we study the crowd of events in comparison with their impact on our model. We validate that much through event websites and official sources where the number of participants or attendants to these events can often be found. Figure 3 contains a dispersion chart showing the relationship between the average per time-window and the actual attendance for the 100 top ranked events we have captured, along with the computed linear regression. The Pearson correlation coefficient is approximately 0.82, which is a relevant result considering the dispersion of the data collected. This experiment suggests that we can automatically estimate the number of people at an event detected by our model.

Internally, the implementation is based on independent,

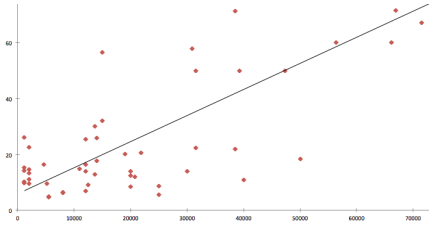


Figure 3: Captured data *vs.* actual attendance for the Top 100 events detected.

autonomous agents that communicate with each other exclusively by asynchronous messages via *Clojure agents*. The agents have the capability to pro-actively assign themselves a particular role: *Crawler* agents assign themselves a target API or web service and manage the reception of data from them, and *Worker* agents schedule periodical aggregation processes. Aggregation processes can be hot-plugged and removed from the Semantic Interpreter at runtime via plug-ins, and can include but are not limited to: crowdedness by area and time interval, crowdedness by Point of Interest and time interval, user trajectories by time interval, disruptive events detection, and so on.

As seen in §3, reliability in data-sensible applications such as the Disruptive Event Detector is a crucial feature. In our system agents are fail-safe in the sense that if a process fails, another agent is taken from a pool to automatically select one or more roles and fulfill them. Scalability is handled by the Semantic Interpreter by not allowing more agents than $n-1$, where n is the number of cores of the host. Please, notice formal tests for obtaining the metrics that will support the benefits of adopting a multi-cored architecture are still to be performed.

An instance of the Semantic Interpreter can be parametrised by setting up the following values in a configuration file: latitude and longitude of the central coordinate, radius of the metropolitan area of the city, counts-as rules (city-specific interpretation rules in RDF), social network API keys, the credentials to MongoDB and Neo4j, and the periodicity of the aggregation processes. This means that, with a small setup of one Java properties file, three instances for Barcelona, Helsinki and Milan have been collecting and aggregating data since July 2013 with minimal downtime.

5. POLICY FRAMEWORK

Information about events occurring in the city have value by themselves, but there is added value in aggregating and interpreting them in order to produce knowledge useful for policy-makers. The SUPERHUB Policy Framework is the component responsible for such processing, providing the perspective of mobility authorities and other stakeholders like transport operators that have relevant contributions to the decision making in mobility policies. As a consequence, it provides tools oriented to improve the quality of the mobility policies adopted by the decision makers in sustainable mobility management. The Policy Framework shall improve the relevance of the information policy makers work with, integrate data and provide a holistic and multi-criteria approach for the analysis of information, facilitate the design and off-line analysis and simulation of the strategies to adopt, and provide metrics and indicators about the success of the policies already applied.

The Policy Framework consists in two main components, the Policy Monitor and the Policy Optimizer. The Policy Monitor extracts the information required to diagnose the current mobility status of the city. The component receives data from the city sensors and analyses it in order to put it in contrast with past data and contextual data. Examples of contextual data include the set of available mobility policies. The analysis allows for inferring higher level information (on-line analysis) such as the relevance of a particular piece of data, relevant situations and events with potential impact on the mobility of the city. The Policy Monitor also collaborates with Policy Optimizer for on-line Policy Optimization. The monitor analyses the performance of active policies *w.r.t.* received data generating an overall picture of the city's situation regarding mobility parameters. In case of low performing policies, the monitor component can start an automatic policy optimization process.

The Policy Optimizer component performs policy optimization triggered manually by mobility experts or automatically by the monitor component. Policy optimization is performed by generating small variations of the policy - and the contextual data related to the policy - and simulating them. Once the results for all variations are available, the Policy Optimizer selects a solution and provides it as the proposal for policy optimization.

Both the Policy Monitor and the Policy Optimizer components rely in a common policy evaluation module. This module is able to put in contrast the data received from the city sensors or the city simulations (social reality) with the policies (political reality). Therefore, policy evaluation module is a core component of the policy framework, and it should be specifically design to be efficient, fast and easy to implement and maintain.

The following code shows the translation maps used to derive low level information into high-level one. As we can see, in order to support these maps we need support for map structures, polymorphism (*e.g.* , values range from String, Vector and function and code is the same) and first class functions (function + and data *'producedCO2InGram'* are treated equally):

```
(def parameter-translation-table
  "Translate␣parameters␣to␣values␣in␣mongoDB"
  {:CO2 "producedCO2InGram", :NOx "producedNOxInGram",
   :SOx "producedSOxInGram", :CO "producedCOInGram",
   :PM10 "producedPM10InGram"
   :public ["METRO" "BUS" "TRAM"]
   :private ["WALK" "BICYCLE" "CAR" "MOTORBIKE"]
   :selfPropelled ["WALK" "BICYCLE"]
   :ALL ["WALK" "METRO" "BUS" "TRAM" "BICYCLE" "CAR" "MOTORBIKE"]})

(def function-translation-table
  "Translate␣function␣names␣to␣clojure␣functions"
  {:sum +, :max max, :min min})
```

Of special interest is the *function translation table* that allows to convert data contained in the policy model (*i.e. "formula" : "(* (/ count_type count_all) 100)"*) into code, effectively computing both concepts equally.

Policies are converted into a normative framework [2] that uses an Ecore metamodel [12] as a representation format. We make use of multi-methods in order to handle the polymorphism required by having to convert input objects of different classes to Clojure structures. Additionally, we take advantage of homoiconicity to dynamically assign to these structures a functional semantic meaning, allowing them to be used as executable formulas:

```
(defmulti operator class)
```

```
(defmethod operator ConjunctionImpl
  [o]
  `(~and
     ~(operator (.getLeftStateFormula o))
     ~(operator (.getRightStateFormula o))))

(defmethod operator ImplicationImpl
  [o]
  `(~or
     ~(cons not
     ~(operator (.getAntecedentStateFormula o)))
     ~(operator (.getConsequentStateFormula o))))

(defmethod operator DisjunctionImpl
  [o]
  `(~or
     ~(operator (.getLeftStateFormula o))
     ~(operator (.getRightStateFormula o))))
```

These formulas allow to obtain the values for the metrics defined in the policies defined by policy-makers in the Policy Framework. However, such metrics are still low-level information about key-performance indicators. [2] defines bridge rules that allow abstracting such low-level information to the level of institutional facts, which in the context of SUPERHUB represent the social and political reality of the city.

In order to implement such bridge rules in our system, we use the *clara-rules* library, which allows to define production rules (similar to CLIPS, Drools or JESS) by purely using Clojure code. Therefore, no additional compiler or interpreter is needed, and run-time changes are simple and straightforward. Again, because the policies can be added, modified or removed in run-time, we make use of homoiconicity to generate the rules in execution time:

```
(eval
  '(defrule holds
     "holds"
     [?h1 <- HasClause (= ?f formula) (= ?f2 clause)]
     [?h2 <- Holds (= ?f2 formula) (= ?theta substitution)]
     =>
     (insert! (->Holds ?f ?theta))))

(eval
  '(defrule norm-instantiation
     "norm_instantiation"
     [?a <- Activation (= ?n norm) (= ?f formula)]
     [?h <- Holds (= ?f formula) (= ?theta substitution)]
     [:not [Instantiated (= ?n norm) (= ?theta substitution)]]
     [:not [Repair (= ?n2 norm) (= ?n repair-norm)]]
     =>
     (insert-unconditional! (->Instantiated ?n ?theta))))
```

6. CONCLUSIONS

This paper has presented one of the key elements of the SUPERHUB project: a mechanism to capture and aggregate information to detect unexpected events. By means of Clojure we have been able to tackle the parallelization problem without extra effort from our side, thus making it possible to capture and interpret this information in almost real-time from multiple, diverse, and schema-free data sources. This allows us to process large amounts of data and provide a more accurate process of detecting unexpected events. This information is used later on by other components used by citizens –to personalise mobility within the city– and policy makers –to react and provide policy adaptation accordingly.

Concretely, using Clojure helped us to easily consume incoming data from data sources in JSON format into language structures; to manipulate data with immutable structures allowing for transparent horizontal partitions of datasets; to assign functional semantic meaning thanks to homoiconicity; to define, at run-time, production rules in the same language, thus allowing a higher level interpretation of information; among others.

Code ported from Java in the first stages from adoption has been reduced to a 10% in lines of code, while a functional approach has reduced the complexity of the code structure, being less dependent on hard-to-refactor class hierarchies. Additionally, for a system of a considerable size, the deployed platforms (using Compojure[4] for the backend and Clojurescript[5] for the frontend) has proven to be robust, *e.g.* the Disruptive Event Detector has currently more than three months of continuous uptime while collecting and processing gigabytes of data per day.

7. REFERENCES

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] S. Alvarez-Napagao, H. Aldewereld, J. Vázquez-Salceda, and F. Dignum. Normative Monitoring: Semantics and Implementation. In *COIN 2010 International Workshops*, pages 321–336. Springer-Verlag, Berlin Heidelberg, May 2011.

[3] I. Carreras, S. Gabrielli, D. Miorandi, A. Tamilin, F. Cartolano, M. Jakob, and S. Marzorati. SUPERHUB: a user-centric perspective on sustainable urban mobility. In *Sense Transport '12: Proc. of the 6th ACM workshop on Next generation mobile computing for dynamic personalised travel planning*. ACM, June 2012.

[4] D. Chakrabarti and K. Punera. Event Summarization using Tweets. *5th International Conference on Weblogs and Social Media, ICWSM*, 2011.

[5] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon. Spatio-temporal indexing in non-relational distributed databases. In *Big Data, 2013 IEEE International Conference on*, pages 291–299, 2013.

[6] D. Garcia-Gasulla, A. Tejeda-Gómez, S. Alvarez-Napagao, L. Oliva-Felipe, and J. Vázquez-Salceda. Detection of events through collaborative social network data. *The 6th International Workshop on Emergent Intelligence on Networked Agents (WEIN'14)*, May 2014.

[7] C. Li, A. Sun, and A. Datta. Twevent: Segment-based Event Detection from Tweets. pages 155–164, 2012.

[8] R. Li, K. H. Lei, R. Khadiwala, and K. C. C. Chang. TEDAS: A Twitter-based Event Detection and Analysis System. pages 1273–1276, Apr. 2012.

[9] T. Sakaki, M. Okazaki, and Y. Matsuo. Earthquake Shakes Twitter Users: Real-time Event Detection by Social Sensors. pages 851–860, 2010.

[10] S. Schockaert, M. De Cock, and E. E. Kerre. Fuzzifying Allen's temporal interval relations. *Fuzzy Systems, IEEE Transactions on*, 16(2):517–533, 2008.

[11] M. Srivastava, T. Abdelzaher, and B. Szymanski. Human-centric sensing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 370(1958):176–197, Nov. 2011.

[12] M. Stephan and M. Antkiewicz. Ecore. fmp: A tool for editing and instantiating class models as feature models. *University of Waterloo, Tech. Rep*, 8:2008, 2008.

[13] J. W. Tukey. Exploratory data analysis, 1977.

[14] J. Weng and B.-S. Lee. Event Detection in Twitter. *ICWSM*, 2011.

[4] https://github.com/weavejester/compojure/wiki

[5] https://github.com/clojure/clojurescript/wiki

A Racket-Based Robot to Teach First-Year Computer Science

K.Androutsopoulos, N. Gorogiannis, M. Loomes, M. Margolis,
G. Primiero, F. Raimondi, P. Varsani, N. Weldin, A.Zivanovic
School of Science and Technology
Middlesex University
London, UK
{K.Androutsopoulos|N.Gkorogiannis|M.Loomes|M.Margolis|G.Primiero|F.Raimondi|P.Varsani|N.Weldin|A.Zivanovic}@mdx.ac.uk

ABSTRACT

A novel approach to teaching Computer Science has been developed for the academic year 2013/14 at Middlesex University, UK. The whole first year is taught in an holistic fashion, with programming at the core, using a number of practical projects to support learning and inspire the students. The Lisp derivative, Racket, has been chosen as the main programming language for the year. An important feature of the approach is the use of physical computing so that the students are not always working "through the screen", but can experience physical manifestations of behaviours resulting from programs. In this paper we describe the MIddlesex Robotic plaTfOrm (MIRTO), an open-source platform built using Raspberry Pi, Arduino, and with Racket as the core coordination mechanism. We describe the architecture of the platform and how it can be used to support teaching of core Computer Science topics, we describe our teaching and assessment strategies, we present students' projects and we provide a preliminary evaluation of our approach.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms

Theory,Human Factors.

Keywords

Educational approaches and perspectives, Experience reports and case studies

1. INTRODUCTION

Designing an undergraduate programme requires a number of choices to be made: what programming language should we teach? Which development environments? Should mathematical foundations play a dominant role, or will they discourage students from attending? Moreover, the current stand of our educational system with respect to industry seems to rely on a discouraging contradiction: on the one hand, it is tempting to market new undergraduate programmes with the claim that they will provide the skills required by industry. On the other hand, we argue that the only certainty is that students will live in a *continuously evolving environment* when they leave education, and that it is not possible to forecast market requests in a few years' time.

In the design of a new Computer Science programme for the academic year 2013/2014 we have been driven by the requirement that we should prepare students for change, and that we should teach them how to *learn new skills* autonomously. Students entering academia may not be prepared for this: they could be arriving from high school where the focus is on achieving good grades in specific tests. How do we achieve the objective of preparing good *learners*?

We decided to employ the Lisp-derivative Racket to support the delivery of a solid mathematical background and the creation of language-independent programming skills. Moreover, we decided to work on *real hardware* so that the students could appreciate the result of executed code. The work is organised around projects involving Arduino, Raspberry Pi, and a Robot that we describe here.

We have completely revised our delivery and assessment methods to support our aims. There are no modules or courses and the activities run seamlessly across the projects. The assessment method is not based on exams, but on *Student Observable Behaviours* (SOBs), that are fine-grained decompositions of learning outcomes providing evidence of students' progress.

Many of the elements in this approach have been tried elsewhere, including: problem-based learning, assessment through profiling and using Lisp as a first programming language. We believe, however, that this programme takes these ideas further than previously, and also blends these in ways that are unique. The integration of Lisp (Scheme) and formalisms in an holistic way was introduced at Hertfordshire by one of the authors many years ago [7], but only in the context of a single module. Several years earlier, a highly integrated curriculum was designed in a project funded by a large company in the UK, to develop formal methods in software engineering practice [8], but this was for small cohorts of students at Master level. From a pedagogical viewpoint, our approach broadly recalls a fine-grained outcome-based

European LISP Symposium 2014 Paris, France

learning path model, but the theoretical implications remain to be assessed in their full meaning, especially for the pedagogical support (see [14] for a recent overview). Finally, an essential aspect of our course structure is the integration of the Lisp-based programming methodology with a range of issues in electrical engineering, robotics and web-based applications. While other educational programmes have often preferred to drop Lisp variants in favour of other more dedicated programming environments (e.g. in the famous case of MIT 6.001 course based on Scheme and [1] redesigned with Python for Robotics applications) we intend to preserve the more in-depth and foundational understanding of programming that a Lisp-style language can offer and at the same time offer a greater flexibility with respect to real-world challenges.

In this paper we focus on how Racket has provided a solid support for our new strategy: in Section 2 we describe the overall structure of the first year and the progress of students from simple examples to more complex scenarios; this progress enables the students to control a real robot, described in Section 3. In section 4 we describe our assessment strategy and we present a tool to support it. An evaluation of our approach is provided in Section 5, where we describe students' projects and various measures for engagement, attendance and overall progress.

2. OVERVIEW OF THE FIRST YEAR

In our new first year of Computer Science, there are no modules or courses and all the activities run across various sessions during the week. The idea is that employing a problem-driven approach, we give students the confidence needed to study independently. In essence, this is our way to teach them *"how to learn"*.

Each week consists of the following structured sessions: *lecture*, *design workshop*, *programming workshop*, *physical computing workshop*, *synoptic workshop*.

General Lecture. A two-hour lecture is given, introducing or developing a topic and related projects. However, this is not where learning should happen: we envisage our lectures as motivational and high-level descriptions of the activities that will follow during the week.

Design Workshop. In these workshops students develop skills required to work in a design environment. Design might be built in software (programming) or hardware, it might involve bolting existing systems together (systems engineering), or developing processes for people who are using the systems (HCI). We cover ways of generating ideas, ways of representing designs so that they can be discussed, professional ways of criticising designs and ways teams of people work together to produce and deliver designs. Delivery happens in an open-space flexible environment, with large tables that can be moved around and arranged in small groups, and the workshop lasts two hours. Students may be asked to present in front of the class the result of their work.

Programming Workshop. In the two-hour programming workshops we help the students with exercises, masterclasses, coaching sessions, to develop their fluency in coding. We have restricted the first year to looking at just one main language, Racket [11], a functional language derived from Lisp. Racket should be new to most students, thus ensuring that the students are all at the same level of experience so that we can focus on teaching best practises rather than undoing bad habits. The choice of a programming language was one of the most carefully debated issues in the design of this new course. Racket was selected for the availability of a number of libraries that support teaching, for its integrated environment (DrRacket) that allows obtaining results with very minimal set-up, and for the availability of a large number of extensions including libraries to interact with networking applications such as Twitter, libraries for Arduino integration and environments for graphics, music and live-coding.

Physical Computing Workshop. The output of software systems increasingly results in tangible actions in the real world. It is very likely that the most common piece of software students will see in their jobs is not a relational database to store sales, but a procedure to manage self-driving cars. As a result, we think that students should be exposed to a wide variety of physical devices that are crucial to understanding computer science. These will range from simple logic gates (the building blocks of every computer currently commercially available), to microcontrollers (Arduino) and other specialist devices. The emphasis is on programming using Racket, not building, these devices. In this two-hour workshop we also explore how to interface these, and how people interact with computers using such devices.

Synoptic Workshop. This is where we "pull everything together" by taking multiple strands of activity and fit all of the bits together. It is longer than the other workshops (4 hours) to allow time to design, build, test and discuss projects. This is not simply about 'applying' what has been learnt - it is about learning and extending what is known in a larger context.

In each of the Programming, Physical and Synoptic Workshops, one staff member and two Graduate Teaching Assistants attend to around 20 students. In the Design session the number of students rises to 40. Students do most of their study during class hours, but handouts contain exercises for self-study and they have almost continuous access to the laboratories to work independently on physical computing.

2.1 Growing Racket skills

Our delivery of Racket starts with the aim of supporting the development of a traffic light system built using Arduino boards [2, 9], LEDs and input switches. The final result should be a system with three traffic lights to control a temporary road-work area where cars are only allowed in alternate one-way flow and with a pedestrian crossing with request button.

Arduino is a microcontroller that can run a specific code or can be driven using a protocol called Firmata [3]. We employ this second approach to control Arduino boards from a different machine. To this end, we have extended the Firmata Racket library available on PLaneT [13] to support Windows platforms, to automatically recognise the USB/serial port employed for connection and to support additional kinds of messages for analog output and for controlling a robot (see next section). Our library is available from [12].

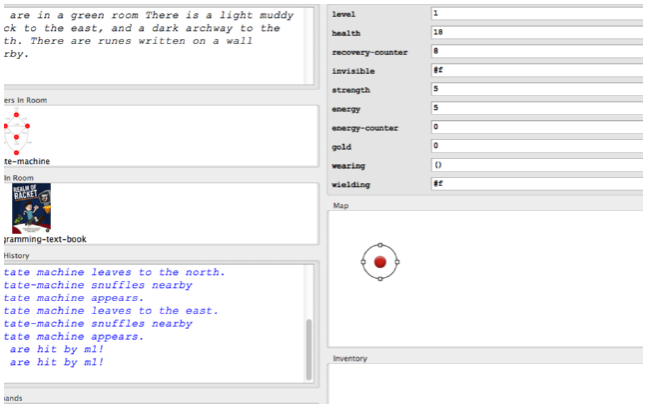


Figure 1: A screenshot of the Dungeon Game Interface

Students employ this library in the first week to start interacting with DrRacket using simple code such as the following:

```
#lang racket
(require "firmata.rkt")
(open-firmata)
(set-pin-mode! 13 OUTPUT_MODE)
(set-arduino-pin! 13)
(sleep 1)
(clear-arduino-pin! 13)
```

This code turns an LED on for a second and then turns it off. Students then start working on lists and see traffic lights as lists of LEDs. High order functions are introduced to perform actions on lists of LEDs, such as in the following code that sets Arduino PINs 7, 8 and 9 to OUTPUT mode:

```
#lang racket
(require "firmata.rkt")
(open-firmata)
(define pins '(7 8 9))
(map (lambda (pin)
       (set-pin-mode! pin OUTPUT_MODE))
     pins)
```

As part of this project students learn how to control events in a timed loop using *clocks* and by making use of the Racket function `(current-inexact-milliseconds)`. This also enables students to *read* the values of input switches and to modify the control loop accordingly.

The result of this project is typically approximately 200 to 500 lines of Racket code with simple data structures, high order functions and the implementation of control loops using clocks.

Following this Arduino project, students explore a number of other Racket applications, including:

- A dungeon game with a GUI to learn Racket data structures. See Figure 1.
- The Racket OAuth library to interact with the Twitter API. A Racket bot is currently running at `https://twitter.com/mdxracket`, posting daily weather forecast for London. A description of this bot is available at `http://jura.mdx.ac.uk/mdxracket/index.php/Racket_and_the_Twitter_API`.

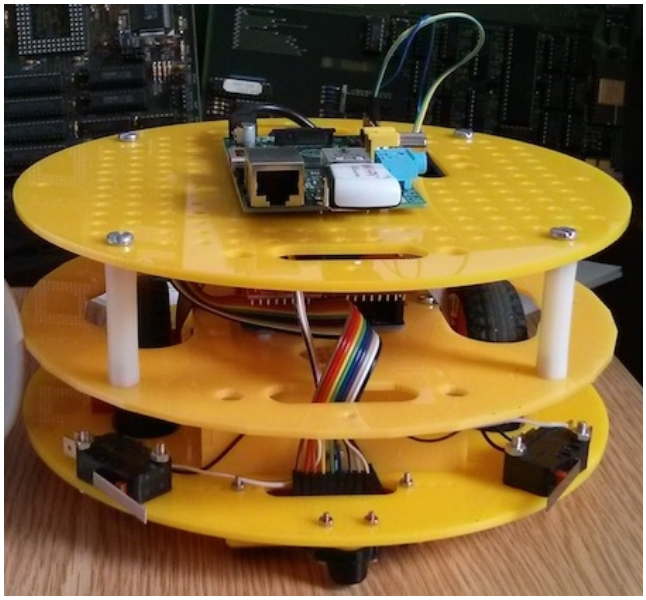


Figure 2: The Middlesex Robotic Platform

- A Racket web server to control an Arduino board. More details about this are available at `http://www.rmnd.net/wp-content/uploads/2014/02/w2-programming.pdf` (this is the handout given to students for their programming and physical computing workshop in one week).

All these elements contribute towards the final project: develop Racket applications for the Middlesex Robotic Platform (MIRTO), described in the next section.

3. MIRTO ARCHITECTURE

The MIddlesex Robotic plaTfOrm (MIRTO, also known as Myrtle), shown in Figure 2, has been developed as a flexible open-source platform that can be used across different courses; its current design and all the source code are available on-line [10]. The Middlesex Robotic platform shown is composed of two units (from bottom to top):

1. The base platform provides wheels, power, basic sensing and low level control. It has two HUB-ee wheels [4], which include motors and encoders (to measure actual rotation) built in, front and rear castors, two bump sensors and an array of six infra-red sensors (mounted under the base), a rechargeable battery pack, which is enough to cover a full day of teaching (8 hours) and an Arduino microcontroller board with shield to interface to all of these. An extended version of Firmata (to read the wheel encoders) is running on the Arduino, which provides a convenient interface for Racket code to control and monitor the robot.
2. The top layer (the panel on top in Figure 2) is where higher level functions are run in Racket and consists of a Raspberry Pi, which is connected to the the Arduino by the serial port available on its interface connection. The Raspberry Pi is running a bespoke Linux image that extends the standard Raspbian image; it includes Racket (current version 5.93), and is using

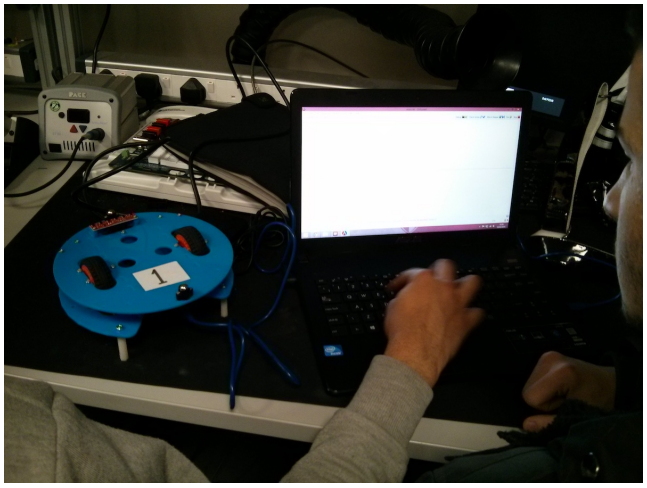


Figure 3: MIRTO Arduino layer connected directly to a PC

a USB WiFi adapter to enable remote connections via SSH and general network activities. This layer enabled us to also use cameras, microphones and text to speech with speakers to extend the range of activities available to students. Additional layers can be added to the modular design to extend the robots capabilities.

The robotic platform is certainly a helpful artifact to engage students more, but it also represents a way to combine our crucial interest in the formal and theoretical aspects underlying computing. In fact, students start using the robot to investigate product of finite state machines (computing the product of the state space of the two wheels) and continue studying all the relevant formal properties that they see implemented on MIRTO. They then move to connecting the Arduino layer directly to a PC, see Figure 3. We have built a bespoke Racket module for this interaction (see Section 3.1); from the students' point of view, this is essentially a step forward with respect to a "simple" traffic light system, and they can re-use the control loops techniques employed for the first project to interact with wheels and sensors. After getting familiar with this library, students progress to study networking and operating systems concepts: this allows the introduction of the top layer, the Raspberry Pi. Students can now transfer their code from a PC to the Raspberry Pi and they control MIRTO over a wireless connection. This allows the introduction of control theory to follow a line and other algorithms (such as maze solving). We present some details of the code in the following section.

3.1 A Racket library for MIRTO

We have built a Racket library for MIRTO that allows students to interact with the robot by abstracting away from the actual messages exchanged at the Firmata level (see the file `MIRTOlib.rkt` available from [10]). The library provides the following functions:

- `setup` is used to initialise the connection between a Racket program and the Arduino layer (this function initialises Firmata and performs some initial set-up for counters). Correspondingly, `shutdown` closes the connection.
- `w1-stopMotor` and `w2-stopMotor` stop the left and the right wheel, respectively. The function `stopMotors` stop both wheels.
- `(setMotor wheel power)` sets `wheel` (either 1 or 2) to a certain power, where `power` ranges between -100 (clockwise full power) and +100 (anti-clockwise full power). `(setMotors power1 power2)` sets both motors with one instruction.
- `(getCount num)` for `num` $\in \{1, 2\}$, returns the "count" for a wheel. This is an integer counter that increases with the rotation of the wheel. A full rotation corresponds to an increase of 64 units for this counter. Given that the wheel has a diameter of 60 mm, it is thus possible to compute the distance travelled by each wheel.
- `enableIR` enables infra-red sensors (these are initialised in an "off" state to save battery); `(getIR num)` (where `num` $\in \{1, 2, 3\}$) returns the value of the infra-red sensor. This is a number between 0 (white, perfectly reflecting surface) and 2000 (black, perfectly absorbing surface).
- `leftBump?` and `rightBump?` are Boolean functions returning true (resp. false) when a bump sensor is pressed (resp. not pressed).

The following is the first exercise that students are asked to do to move the wheels for one second:

```
#lang racket
(require "MIRTOlib.rkt")
(define (simpleTest)
  (setup)
  (setMotors 75 75)
  (sleep 1)
  (stopMotors)
  (shutdown))
```

This code moves the wheels for one second and then stops them. Students test this code using the Arduino layer only, as shown in Figure 3. Similarly to the traffic light project, students then move to more complex control loops and start using the Raspberry Pi layer using SSH and command-line Racket. The following snippet of code extracted from a control loop prints the values of the infra-red sensors every two seconds:

```
;; [...]
  (set! currentTime (current-inexact-milliseconds))
;;
  (cond ( (> (- currentTime previousTime) 2000)
          (map (lambda (i)
                 (printf " IR sensor ~a -> ~a\n" i
                 (getIR i)))
            '(1 2 3))
          (set! previousTime
              (current-inexact-milliseconds))))
;; [...]
```

The functions provided by the library allow the implementation of a Racket-based PID controller [6] for MIRTO. Students are also introduced to maze solving algorithms, which can be implemented using the infra-red sensors and the bump sensors. The Racket code for both programs is available from [10] in the `servos-and-distance` branch.

After these exercises and guided projects, students are asked to develop an independent project. We report some of these projects in Section 5.

4. ASSESSMENT STRATEGY

As mentioned above, the delivery of the first year of Computer Science has been substantially modified, modules have been removed and students are exposed to a range of activities that contribute to *projects*.

As a result, we have introduced a new assessment strategy to check that students have understood and mastered the basic concepts required during the second year and are able to demonstrate these through practical demonstration. We use the term **Student Observable Behaviours** (SOBs) to refer to fine-grained decompositions of learning outcomes that provide the evidence that the students are progressing. Passing the year involves demonstrating SOBs. There are three types of SOBs:

Threshold level SOBs are those that *must* be observed in order to progress and pass the year. Students must pass *all* of these; a continuous monitoring of the progress using the tool described below ensures that any student who is at risk of not doing so is offered extra support to meet this level.

Typical level SOBs represent what we would expect a typical student to achieve in the first year to obtain a good honours degree. Monitoring this level provides a very detailed account of how each student is meeting expectations. Students are supported in their weak areas, encouraged not to hide them and not to focus only on the things they can do well. Our aspiration is to get the majority of students to complete all the typical level SOBs.

Excellent level SOBs identify outstanding achievements. These are used to present real challenges of different types to students who have demonstrated to be ready for them.

Projects were designed to offer assessment opportunities both en-route and in the final project delivery. Projects are posed in such a way as to ensure that students who engage with the process have the opportunity to demonstrate threshold level SOBs. As a result, "failure" to successfully complete a project does not lead to failure to complete the threshold SOBs. Projects have a well-defined set of core ideas and techniques (threshold), with suggestions for enhancements (typical), and open-ended questions (excellent). Note that there is no concept of averaging or summation: in theory a student could complete all of the excellent level SOBs, but fail the year as a consequence of not meeting one threshold SOB. This is virtually impossible in practice, as staff are aware that there are outstanding threshold SOBs, and take the opportunity of observing them en-route. Of course, if a student really can't do something that has been judged threshold, we will deem it a failure.

Students who fail to demonstrate all threshold SOBs by the end of the academic year will, at the discretion of the Examination Board and within the University Regulations, be provided with a subsequent demonstration opportunity. This will normally be over the Summer in the same academic year. Resources including labs and support staff will be made available during this period.

The process of assessment and feedback is thus continuous via a "profiling" method. This method allows us to track every student in detail, to ensure that we are supporting development and progression. This means we have comprehensive feedback to the teaching team available in real time. Also, students have a detailed mechanism available to monitor their own progress. This includes ways of viewing their position relative to our expectations, but also to the rest of the group. The students have multiple opportunities to pass SOBs. There are no deadlines and SOBs can be demonstrated anytime during the year, although each SOB carries a "suggested" date range in which it should be observed. Although the formal aspect of the profiling method appears to be a tick-box exercise, discussion and written comments (where appropriate) are provided at several points throughout the year.

4.1 The Student Observable (SOB) Tool

Overall, we have defined 119 SOBs: 34 threshold, 50 typical and 35 excellent. In terms of Racket-specific SOBs, 10 of them are threshold and include behaviours such as "Use define, lambda and cond, with other language features as appropriate, to create and use a simple function."; 15 SOBs are typical, such as "Define functions to write the contents of a data structure to disk and read them back"; there are 13 SOBs at the excellent level, for instance: "The student can build an advanced navigation system for a robot in Racket that uses different data streams"

Our first year cohort consists of approximately 120 students. An appropriate tool is crucially needed to keep track of the progress of each student and to alert the teaching team as soon as problems arise (students not attending, students not being observed for SOBs, etc.). We have developed an on-line application that takes care of this aspect, in collaboration with research associates in our department.

Figure 4 presents a screenshot of the tool when entering or querying SOBs. The first column identifies the SOB by number; the second the level (threshold, typical, excellent); the third the topic (Racket, Fundamentals, Computer Systems, Project Skills); the fourth offers a description; the fifth and sixth column indicate respectively start and expected completion dates; the last column is an edit option. In addition to this facility, the tool provides a set of graphs to monitor overall progress and attendance. Background processes generate reports for the teaching team about non-attending or non-performing students. As an example, Figure 5 shows in tabular form the list of students (id number, first and last name, email), highlighting those who have (threshold) SOBs that should have been observed at the current date.

Figure 6 shows a screenshot of the "observation" part of the tool. In this case a demo student is selected and then the appropriate SOBs can be searched using the filters on the right. Different colours are used to highlight the most relevant SOBs. In addition, for each level a progress bar displays the overall progress of the student in green against the overall average progress of the cohort (vertical black bar);

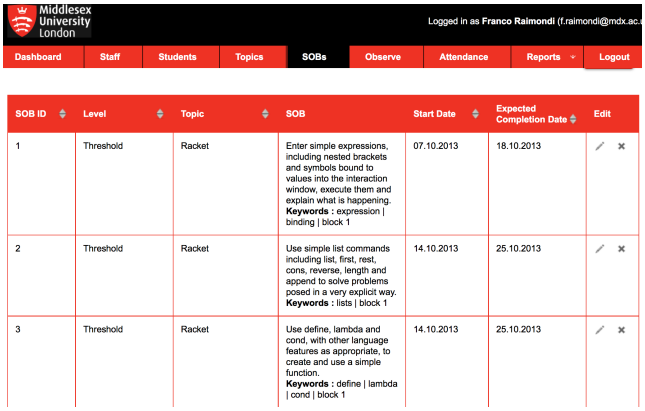


Figure 4: Entering and searching SOBs

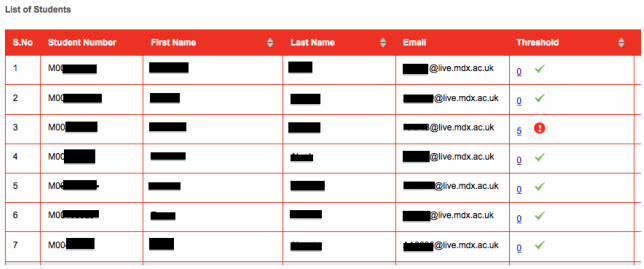


Figure 5: Student list with SOBs

in this case, the student is slightly ahead of the overall class for threshold SOBs. The "Notes" tab can be used to provide feedback and to record intermediate attempts at a SOB. In addition to the design presented in the figure we have also implemented a tablet-friendly design to be used in the labs.

Students are provided a separate access to the database to check their progress. A dashboard provides immediate and quick access to key information (number of SOBs expected to be observed in the coming week, number of SOBs that are "overdue", etc.). More detailed queries are possible for self-assessment with respect to the overall set of SOBs and with respect to the cohort in order to motivate students. As an example, Figure 7 shows the student progress (green bar) with respect to the whole class (yellow bars) for *typical* SOBs.

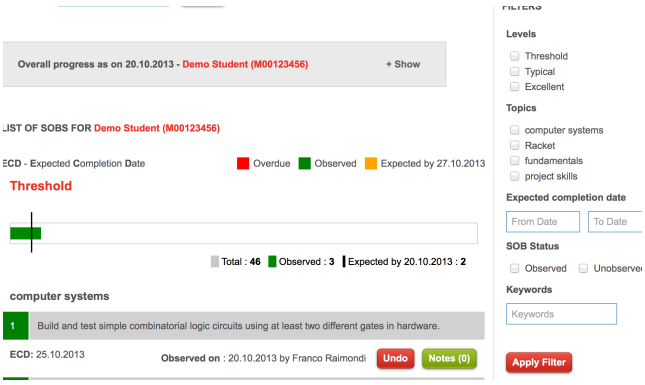


Figure 6: Observing a SOB for a student

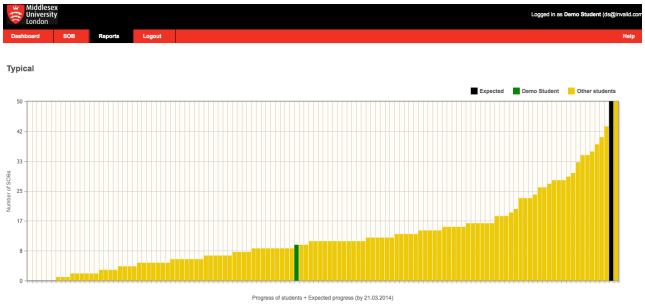


Figure 7: Student view: position with respect to class

As described in the following section, this tool has enabled the teaching team to provide continuous support to the students who needed it most, by identifying non-attending or dis-engaged students very early in the year.

5. EVALUATION

We provide here an overview of two forms of evaluation: a list of students' projects built using Racket and MIRTO, and an evaluation of average attendance, progression rate and engagement.

5.1 Student projects

In the final 3 weeks of their first year, students have been asked to work in teams and submit projects using MIRTO and Racket. Members of staff have provided support, but all the projects have been designed and implemented entirely by the students. The following is a list of some of these final projects.

- Dancing robots: this has been a popular theme, with two groups working at coordinating the movement of multiple robots in a choreography of their choice. Two example videos are available at `https://www.youtube.com/watch?v=V-NfC4WK2Sg` and `https://www.youtube.com/watch?v=nMjdH9TCKOU`.
- A student has developed a GUI running on the Raspberry Pi. By tunnelling an X connection through SSH the robot can be controlled from a remote computer. The project also includes the possibility of taking pictures and a sequence of instructions to be executed. The video is available at the following link: `https://www.youtube.com/watch?v=FDi2TSCe3-4`
- A student has implemented a web server running on the Raspberry Pi, so that the robot can be controlled using a browser. The web interface enables keyboard control of the movements and detects the values of infra-red and bump sensors. Additionally, from the web interface a user could take a picture or start line following (on a separate thread). Finally, the student has also implemented a voice recognition feature by combining Racket and Pocketsphinx [5]: when the name of a UK city is pronounced, the local weather is retrieved. The video is available at this link: `https://www.youtube.com/watch?v=lwsG0lD55wk`.
- Finally, a student has taken a commercially available robotic platform (4tronix initio robot) built on top of

Arduino and has modified it by installing firmata and by adding a Raspberry Pi running Racket. To this end, the student has developed a bespoke version of MIRTOlib.rkt for this new robotic platform, adding support for servo motors. The video of this project is available at this link: `https://www.youtube.com/watch?v=hfByxWhyXkc`.

More importantly, through the projects and the threshold SOBs we have been able to assess the ability of nearly all students to control a robot from Racket, thus ensuring that they have achieved the minimal level of familiarity with the language to progress to the second year.

5.2 Attendance, engagement and progression

The teaching team has been concerned with various risks associated to this new structure of delivery for a whole first year cohort:

- Would students *attend* all the sessions, or only drop-in to tick SOBs?
- Would students *engage* with the new material?
- Would students focus on threshold SOBs only, and not *progress* beyond this level?

The delivery of this year has now nearly completed, with only two weeks left in our academic year. In "standard" programmes these are typically dedicated to revision before the exams. In our case, instead, we are in a position of analysing the data collected over the year to answer the questions above.

5.2.1 Attendance

Figure 8 shows the weekly attendance rate in percentage for the new first year programme (in blue) and for two other first year modules from another programme (in green and red, anonymised). Unfortunately, no aggregated attendance data is available for the other programme. As a result, we can only compare attendance of the whole first year with these two modules, one of which has *compulsory* attendance.

The graph displays attendance per week; a student is considered to have attended in a week if s/he has attended *at least one session* during the week. "Standard" modules have an attendance ranging between 50% and 70% for the "core" module with compulsory attendance, and between 40% and 60% for the "non-core" module. There is also a decreasing trend as weeks progress.

We have been positively surprised by the attendance for the new programme, which has been oscillating between 80% and 90% with only a minimal drop over the year (the two "low" peaks around week 10 and 17 correspond to British "half-term" periods, when family may go on holiday). Unfortunately, no aggregated attendance data is available for other programmes. As a result, we can only compare attendance of the whole first year with a *compulsory* module in another programme and for a standard first year module.

5.2.2 Engagement

Engagement is strictly correlated with attendance, but it may be difficult to provide a direct metric for it. We typically assess engagement by checking log-in rates in our VLE environment and, in our case, we could also measure

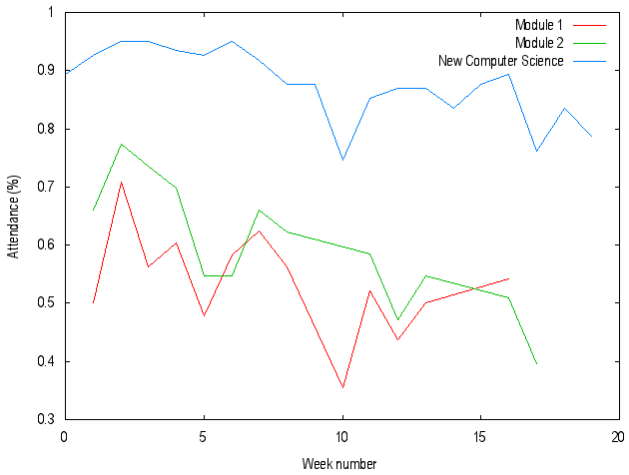


Figure 8: Weekly attendance (comparison)

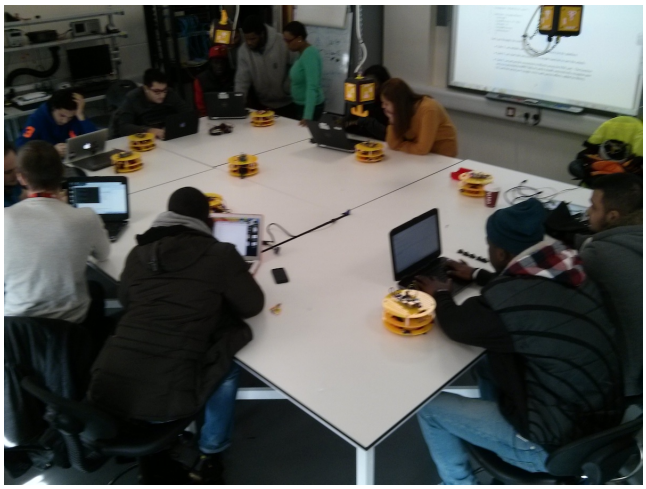


Figure 9: Example lab session

SOB progression. We were able to identify approximately 10% of the cohort being "not engaged". Thanks to our tool, we have been able to address these students individually.

In addition to SOB progression, we could also measure usage of the MIRTO platforms. We have built 10 yellow and 10 blue robots. We have used 4 of these for research and 2 for demo purposes, leaving a total of 7 blue and 7 yellow robots for teaching in the workshops. There are typically 20 students allocated to each workshop, working in groups of 2 or 3 (see Figure 9); all sessions required all robots, showing that all students were engaged with the material.

5.2.3 Progression

Finally, there was a risk that the majority of the class would focus just on the achievement of threshold SOBs. Our first year is not graded and therefore, once the threshold SOBs have been achieved, there is no formal difference between students with different numbers of SOBs.

Besides anecdotal evidence of students working on optional projects, our monitoring tool has allowed us to encourage the best students to work on new challenges for the whole year. This has resulted in the vast majority of stu-

dents progressing beyond the "threshold" level. This is confirmed by the results presented in Figure 10: the majority of students has progressed well beyond the 34 threshold SOB mark (red line in the figure). The same trend is confirmed if Racket-specific SOBs are considered. Figure 11 shows that approximately 70% of the students have completed SOBs beyond the required threshold level (the same distribution occurs for other SOB categories).

The tool has also shown interesting approaches to this new structure, both in general and for Racket-specific SOBs: some students have focussed on threshold SOBs first and only moved to typical and excellent SOBs later. Other students, instead, have worked at typical and excellent SOBs with many threshold SOBs still outstanding.

6. CONCLUSION

In designing a new Computer Science programme for Middlesex University we have decided to make use of Racket and to design and build a robotic platform to support our delivery. To the best of our knowledge, this is the first time that this approach is applied at such a large scale. The preparation of this new programme has required the joint effort of a large team of academics and teaching assistants for more than a year before the actual delivery. However, the results obtained are very encouraging: attendance and engagement are well above average, and the large majority of students are progressing beyond the level required to pass this first year.

7. REFERENCES

[1] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] M. Banzi. *Getting Started with Arduino*. Make Books - Imprint of: O'Reilly Media, Sebastopol, CA, 2008.

[3] The Firmata protocol. `http://firmata.org/`. Accessed: 2014-03-20.

[4] The MIddlesex Robotic plaTfOrm (MIRTO). `http://www.creative-robotics.com/About-HUBee-Wheels`. Accessed: 2014-03-20.

[5] D. Huggins-Daines, M. Kumar, A. Chan, A.W. Black, M. Ravishankar, and A.I. Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2006*, volume 1, pages 185–188, 2006.

[6] M. King. *Process Control: A Practical Approach*. John Wiley & Sons, 2010.

[7] M. Loomes, B. Christianson, and N. Davey. Formal systems, not methods. In *Teaching Formal Methods*, volume 3294 of *Lecture Notes in Computer Science*, pages 47–64. 2004.

[8] M. Loomes, A. Jones, and B. Show. An education programme for software engineers. In *Proceedings of the First British Software Engineering Conference*, 1986.

[9] M. Margolis. *Arduino Cookbook*. O'Reilly Media, 2011.

[10] The MIddlesex Robotic plaTfOrm (MIRTO). `https://github.com/fraimondi/myrtle`. Accessed: 2014-03-20.

[11] The Racket Language. `http://racket-lang.org`. Accessed: 2013-10-21.

[12] Racket Firmata for Middlesex Students. `https://bitbucket.org/fraimondi/racket-firmata`. Accessed: 2014-03-20.

[13] Racket Firmata. `http://planet.racket-lang.org/display.ss?package=firmata.plt&owner=xtofs`. Accessed: 2014-03-20.

[14] F. Yang, F.W.B. Li, and R.W.H. Lau. A fine-grained outcome-based learning path model. *IEEE T. Systems, Man, and Cybernetics: Systems*, 44(2):235–245, 2014.

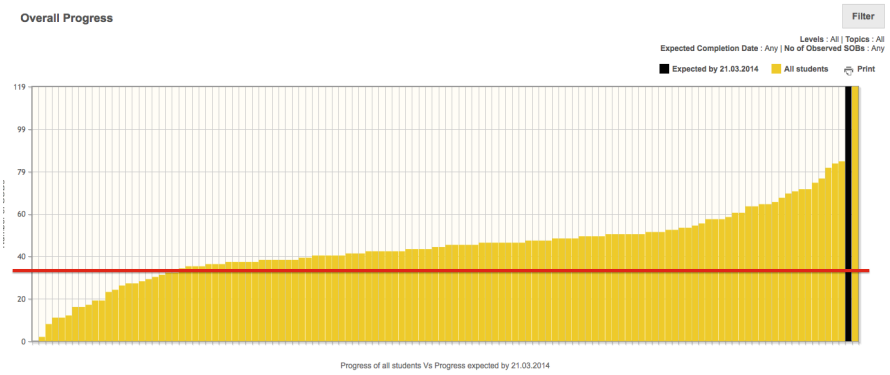


Figure 10: SOB overview (end of year)

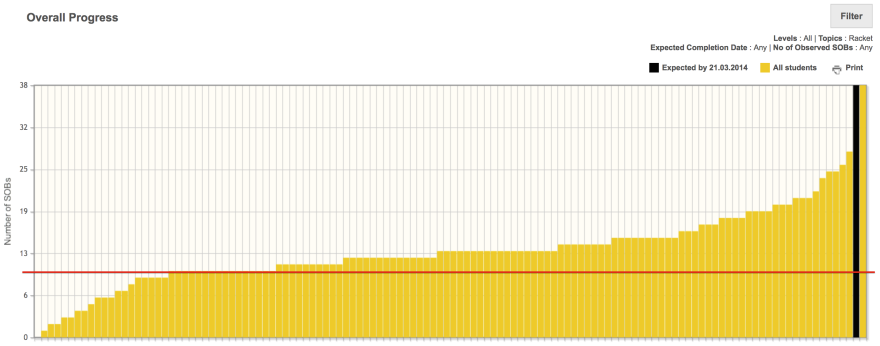


Figure 11: Threshold SOBs for Racket (end of year)

Session IV: Crossing the Language Barrier

A Need for Multilingual Names

Jean-Paul A. Barthès
UMR CNRS 7253 Heudiasyc
Université de Technologie de Compiègne
60205 Compiègne, France
barthes@utc.fr

ABSTRACT
An increasing number of international projects require using or developing multilingual ontologies. This leads to awkward problems when several languages are used concurrently in the same environment. I discuss in this paper the possibility of defining data structures called *multilingual names* to facilitate the programmer's life, hoping that it could be made part of the various Lisp environments. An example of a multilingual ontology is given to illustrate the approach.

Categories and Subject Descriptors
D.3.3 [**Language Constructs and Features**]: Data types and structures

General Terms
Programming structures, Lisp, Multilingual data

1. INTRODUCTION
An increasing number of international projects, in particular in Europe, require multilinguism. A few years ago we were part of such a project, the Terregov project that focused on the development of eGovernment for social services[1]. This particular project required developing an ontology in English, French, Italian and Polish [2]. On the other hand, because we are routinely working with Brazil and Japan, we often have to develop interfaces supporting the English, French, Portuguese or Japanese languages.

A number of researchers have addressed the issue of developing multilingual ontologies, the most popular position being of developing separate ontologies, then performing some alignment on the resulting separate ontologies. With this approach, the MultiFarm dataset provides references for testing multilingual ontologies and alignments [6]. Although this seems to work reasonably well, it taxes the programmer who must be careful in developing common code. The XML approach on the other hand allows defining concepts only once, tagged with lingual attributes in formalisms like OWL (see for example the hotel multilingual ontology developed by Silveira et al. [3]). However, using such tags in the programs is not especially easy, in particular with some ontology editors that were developed essentially for English and added multilinguism as a second thought.

Our ontologies are not built for supporting translation like the Pangloss ontology [5], nor generating descriptions as in [4]. We are not interested in building linguistic resources like in [7], but in the programming aspect for applications where several languages are necessary. In that context, ontologies are used by people for common reference, by content languages in multi-agent systems, and for supporting Human-machine communication in natural language.

In our previous projects we faced the problem of finding a way to express concepts in a multilingual context within Lisp environments. We had problems with linguistic tags, synonyms and versions, which we had to solve. The paper mostly presents the second issue, namely how we dealt with multilingual data at a very low level, from a programmer point of view. The last issue, namely versioning, has to do with the internal representation of concepts and individuals independently of the multilingual approach and is outside the scope of this paper. We would like very much to have standard low level structures and primitives to achieve a better and cleaner programming.

2. REQUIREMENTS
Our goal was to let the Lisp programmer develop applications as if they were using a particular (natural) language and have the same code run in other languages simply by specifying a special environment variable called `*language*`. Thus, we want to obtain the following behavior:

```
(let ((*language* :FR))
  (print greetings))
print -> Bonjour
NIL

(let ((*language* :ZH))
  (print greetings))
print -> 你好
NIL
```

[1]http://cordis.europa.eu/projects/rcn/71114_en.html

where `greetings` is a variable having a multilingual value. We call the value of such variables multilingual names, although they may contain multilingual sentences.

Thus, the programmer should be able to write statements like `(print greetings)`, the output being controlled by the value of the `*language*` special variable.

The second requirement is to use standard language codes. To do so, we adopted the ISO-639-1 standard[2] which specifies two letter codes for most languages, although RFC4646 offers more precise possibilities.

3. DEFINITION AND PROPERTIES OF MULTILINGUAL NAMES

3.1 Definition

A multilingual name, MLN, is a set of subsets T_i containing terms taken from a set of phrases V_j belonging to a particular language j

$$MLN = \{T_1, T_2, ..., T_n\} \quad \text{with} \quad T_i = \{s_1^j, s_2^j, ..., s_k^j\}$$

$$\text{and } s_i^j \in V_j$$

Thus, s_i^j is a phrase of the j-language and T_i is a set of synonym phrases. The set of languages can be augmented by two "artificial" languages: V_ω and $V_?$, where V_ω stands for any languages and $V_?$ represents an unknown language[3]. Each T_i set can be semi-ordered, in the sense that the first value for example, could have a more important role among all synonyms.

In this definition, nothing prevents a term to appear in different languages, for example the same term `"Beijing"` can appear in many different languages.

3.2 Language Environment

The *language environment* is defined to host a particular language including ω, which is a special marker meaning that all languages are accepted in this environment (used for inputs).

The purpose of a language environment is to provide a default language when constructing MLNs from strings or symbols.

3.3 Dominant Language

One of the languages can be privileged and declared as the dominant language.

This is useful when building multilingual ontologies for structuring the concepts around a set of concepts containing terms in the dominant language. For example, when we built the Terregov ontology, using English, French, Italian, and Polish, English was chosen to be the dominant language. Thus, the dominant language can be viewed as a default language.

[2]http://www.mathguide.de/info/tools/languagecode.html.

[3]In addition one could define a V_π denoting a "pending" language, i.e. a language that has not yet been recognized. But we did not use this feature.

3.4 Canonical Form for a Term

The expression "canonical form" is probably misleading but corresponds to the following idea: when extracting a term in a given language from an MLN, we define the *canonical form* as the the first term in the set of language synonyms, or else the first synonym of the dominant language, or else the first name of a randomly chosen language.

When working with an ontology often specialists give a main term for defining a concept, then add additional synonyms. The firs term is usually more significant than the synonyms. Thus if a concept must be represented by a single term the first one is supposedly the best. Now, sometimes, when building an ontology in several languages, a concept may not have been defined in this language, but already exists in another one. It is important to return something to help users to see some term describing a concept that is used in the program. It is also a convenient mechanism allowing not to repeat a term that appears in many languages, for example names found in an onomasticon:

```
(:en "Paris" :zh "巴黎")
```

Here "Paris" can be omitted from most languages in which it is written in the same fashion when English is the dominant language, which is convenient in Europe.

3.5 Properties

The properties comprise equality, term membership and fusion. They were defined in order to facilitate the creation of indexes. Examples are given in the following section about implementation.

Equality. Two MLN are equal if they share one of the synonyms for a given language, or between a language and the unknown language, or between unknown languages.

Term Membership. A term s_j is said to belong to an MLN if it is one of the synonyms specified by the language environment. This property is however not essential.

Fusion. A new MLN can be obtained by merging two MLNs. Each of its language sets T_i is obtained by merging the corresponding T_is eliminating duplicates. The order within each set of synonyms is kept with respect to the order of the MLN arguments.

4. OUR IMPLEMENTATION

This section describes our solution for implementing MLNs and presents the different functions associated with this choice.

4.1 Multilingual Name Format

4.1.1 Format

One needs to create a data structure for holding the names in the different languages. Among the different possibilities, we selected a property list format using keywords for language tags and strings for synonyms, e.g.

```
(:en "Beijing" :fr "Pékin; Beijing" :zh "北京")
```

Note that in the MLN two French synonyms appear separated by a semi-column for labeling the same concept. This choice is debatable and is discussed in Section 6.1.

In our implementation the language codes (LTAG) are taken from ISO-639-1already mentioned[4]. All the language tags used in an application are kept in `*language-tags*` a special list that defines the set of legal languages for the application.

The tag corresponding to ω is set to `:all` and the one corresponding to "?" is set to `:unknown` for dealing with external inputs. The first one, `:all`, is used when inputing data from a file containing several languages, the second one, `:unknown`, is used when the language data is not known, but we want an MLN format.

The environment is defined by the `*language*` special variable that can take any value of legal language tags and ω.

Extensions: Two extensions are possible:

1. the concept of multilingual name can be extended to include whole sentences instead of simply names, which is convenient for handling dialogs;
2. MLNs can include simple strings with the following understanding:
   - if `*language*` is defined, then a string s can be considered as equivalent to (val(*language*) s);
   - if `*language*` is undefined, then a string s can be considered equivalent to `(:unknown` s`)`.

Thus, with this convention, MLN operators can also work on simple strings.

4.1.2 Equality

With the proposed formalism:
`(:en "name" :fr "nom; patronyme")`
and
`(:en "surname" :fr "patronyme")`
are equal, meaning that they represent the same concept.

Or:
`(:unknown "patronyme")`
and
`(:en "name" :fr "nom; patronyme")`
are considered equal.

4.1.3 Term Membership

For example, if the environment language is French
`"cité"` belongs to `(:en "city;town" :fr "ville;cité")`.

4.1.4 Fusion

For example:
`(:en "surname" :fr "patronyme")`
`(:fr "nom" :de "Name")`
when fused yield
`(:en "surname" :fr "patronyme; nom" :de "Name")`

[4]We only worked with English, French, Japanese Polish, Portuguese, and Spanish.

4.2 The MLN Library of Functions

The functions to deal with the proposed format belong to two groups: those dealing with language tags, and those dealing with synonyms. Details are given in the appendix.

4.2.1 Functions Dealing with Multilingual Tags

The following functions dealing with the language tags (LTAG) were found useful. They implement different features:

- constructor: building an MLN from parts
- predicates: type-checking, equality, term membership
- editors: adding or removing synonym values, setting synonym values, removing a language entry, fusing MLNs
- extractors (accessors): extracting synonyms, obtaining a canonical name
- printer

The `%MLN-EXTRACT` function allowing extraction is specially useful. It takes three arguments: `MLN`, `LTAG` (key) and `ALWAYS` (key), and extracts from `MLN` the string of synonyms corresponding to the language specified by `LTAG` (defaulting to the current environment language: `*language*`). It works as follows:

1. If MLN is a string returns the string.
2. If language is `:all`, returns a string concatenating all the languages.
3. If language is `:unknown`, returns the string associated with the `:unknown` tag.
4. If `always` is `t`, then tries to return something: tries first the specified language, then tries English, then `:unknown`, then first recorded language.

The `:always` option is interesting when dealing with multilingual ontologies, when one wants to obtain some value even when the concept has no entry in the specified language.

4.2.2 Functions Dealing with Synonyms

We also need some functions to take care of the set of synonyms, for adding, removing, retrieving values, and for merging sets of synonyms.

4.3 Extension of Standard Primitives

Since MLN can be considered as a new datatype some of the Lisp primitive could be extended to include MLNs, for example functions like equal, +, -, member, etc. Some should work in a combination of strings and MLN as seen in the MLN library functions.

Consider for example the `concatenate` primitive. It has already been extended in the Allegro^{TM} environment to `string+` to simplify programming, arguments being coerced to strings. It could be extended easily to include MLNs by

applying the `mln-get-canonical-name` function to the MLN arguments before concatenating them.

The `equal` primitive can be extended to include MLN equality. For example the NEWS application consists of collecting news items produced by different people in different countries. Each participant has a Personal Assistant agent (PAs). PAs and their staff agents have each an ontology in the language spoken by the participant, e.g. English, French, Portuguese, etc. A Service Agent, named PUBLISHER, collects information sent be the various PAs and in turn sends back information like categories of news , keywords. PUBLISHER has a multilingual ontology and knowledge base. The list of categories if asked by a French speaking PA will be extracted as follows:

```
(mapcar
   #'(lambda (xx)
        (%mln-extract xx :language :FR :always t))
   (access '("category")))
```

Now, if a PA send a message asking to subscribe to a category, a keyword or follow a person, the PUBLISHER can test which function to invoke by comparing the message argument to a predefined MLN:

```
(let ((*language* message-language))
...
(cond
  ((equal+ message-arg *e-category*) ...)
  ((equal+ message-arg *e-keyword*)...)
  ((equal+ message-arg *e-person*)...)
  ...))
```

where `*e-category*`, `*e-message*` or `*e-person*` are MLN containing the allowed synonyms for designating categories, keywords or persons in the legal languages. In a way the corresponding MLNs are interned using the special variables.

In the MOSS knowledge representation that we use [1] MLNs are values associated to a particular attribute. Since MOSS attributes are multi-valued, MLNs are assimilated to multiple values. And the functions and methods handling attribute values can be extended to accommodate MLNs.

4.4 Example of Multilingual Ontology Items

The examples of this section are taken from the past European FP6 TerreGov project, and the definitions are those given by the different specialists.

The following concept definition of a country uses multilingual data: English, French, Italian and Polish.

```
(defconcept
  (:en "Country; Land" :fr "Pays" :it "Paese; Stato"
   :pl "Kraj; Państwo")
  (:is-a "territory")
  (:att (:en "name") (:unique))
  (:doc
   :en "a Country or a state is an administrative entity."
   :fr "Territoire qui appartient à une nation, qui est
        administré par un gouvernement et dont les
        frontières terrestres et maritimes ont clairement
        été établies."
   :pl "Kraj lub Państwo to jednostka administracyjna."
   :it "Un Paese o Stato è un'entità amministrativa."))
```

Figure 1: Individual of the ontology in an English context.

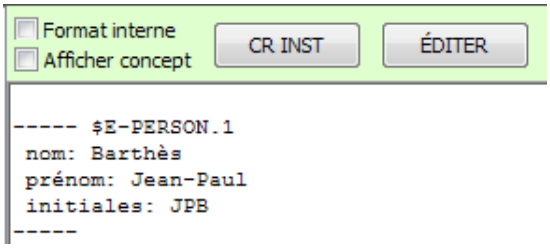


Figure 2: The same individual in a French context.

The `:is-a` property indicates a subsumption between the concept of "Country" and the supposedly more general concept of "Territory". Note that "Stato" is the term given by the Italian specialist to designate a country. If American "states" were part of the application, then most probably the concept would be defined as "American State" distinct from the concept of "State".

The following item defines an individual country:

```
(defindividual "country"
  (:en "united kingdom" :fr "Royaume Uni"
   :it "Inghilterra" :pl "Anglia"))
```

this gives examples for concept names and attributes. Relations are expressed in a similar manner. For example, within the concept of person the relation between a person and an individual representing the gender of a person is expressed as:

```
(:rel (:en "gender" :fr "sexe" :pl "p■leć" :it "sesso")
   (:one-of (:en "male" :fr "masculin" :pl "mę■czyzna"
             :it "maschio")}
```

Figures 1 and 2 show the same individual (`$E-PERSON.1`: internal ID) in an English and French context. The names of the properties belonging to the multilingual ontology adapt to the active context. The same is true for the names of the buttons and the check boxes of the displaying window. The show-concept widget for example is simply programmed as:

```
(make-instance 'check-box
  :name :if-check-box
  :font (MAKE-FONT-EX NIL "Tahoma / ANSI" 11 NIL)
  :left (+ (floor (/ (interior-width win) 2)) 4)
  ;:left 502
  :top 18
  :title (%mln-extract *WOVR-show-concept*)
  :on-change 'ow-class-check-box-on-change
  :width 105
  :bottom-attachment :top
  :left-attachment :scale
  :right-attachment :scale
  :tab-position 4)
```

The only difference with programming using a single natural language is the line associated with the title option. This permits to centralize all data tied to languages, like `*wovr-show-concept*`, in a single place.

5. TRANSLATING INTO OWL

The proposed formalism is routinely used in connection to the MOSS language [1] for representing ontologies and knowledge bases. It can be combined with some of the MOSS features like versioning, indexing, automatic maintenance of inverse links, virtual classes and virtual properties.[5]

However, because most European projects require using standard approaches we could not deliver the final product in Lisp in the TerreGov project and were led to build a MOSS-to-OWL compiler which produced the same information in an OWL format. For the concept of country it yields the following structures:

```
<owl:Class rdf:ID="Z-Country">
  <rdfs:label xml:lang="en">Country; Land</rdfs:label>
  <rdfs:label xml:lang="fr">Pays</rdfs:label>
  <rdfs:label xml:lang="it">Stato</rdfs:label>
  <rdfs:label xml:lang="pl">Kraj; Państwo</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Z-Territory"/>
  <rdfs:comment xml:lang="en">a Country or a State is
     an administrative entity.</rdfs:comment>
  <rdfs:comment xml:lang="fr">Territoire qui appartient
     à une nation, qui est administré par un gouvernement
     et dont les frontières terrestres et maritimes ont
     clairement été établies.</rdfs:comment>
  <rdfs:comment xml:lang="pl">Kraj lub Państwo to
       jednostka administracyjna.</rdfs:comment>
  <rdfs:comment xml:lang="it">Un Paese o Stato è
       un'entità amministrativa.</rdfs:comment>
</owl:Class>
```

Here we use labels for the different names, accepting synonyms, and comments for the documentation. Note that the concept of country here is very simple. Of course one must define the attribute name, here called `has-name` which is shared by a number of other concepts (not all shown here):

```
  <owl:DatatypeProperty rdf:ID="hasName">
    <rdfs:label xml:lang="en">Name</rdfs:label>
    <rdfs:label xml:lang="fr">Nom</rdfs:label>
    <rdfs:label xml:lang="pl">Nazwa</rdfs:label>
    <rdfs:label xml:lang="it">Nome</rdfs:label>
    <rdfs:domain>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#Z-Territory"/>
          <owl:Class rdf:about="#Z-Conurbation"/>
          <owl:Class rdf:about="#Z-Country"/>
          <owl:Class rdf:about="#Z-ElectedBody"/>
         ...
          <owl:Class rdf:about="#Z-Month"/>
          <owl:Class rdf:about="#Z-Project"/>
          <owl:Class rdf:about="#Z-Tool"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
    <rdfs:range rdf:resource="http://www.w3.org/2001/
       XMLSchema#Name"/>
  </owl:DatatypeProperty>
```

In addition one must express the cardinality restriction:

```
<owl:Class rdf:about="#Z-Country">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasName"/>
      <owl:cardinality
          rdf:datatype="&xsd;nonNegativeInteger">1
              </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The paper by Barthès and Moulin [1] gives details on how the formalism can be used to develop multilingual ontologies and how the declarations can be translated into OWL with the addition of JENA or SPARQL rules.

6. LESSONS LEARNED

We have been using this formalism successfully through several projects requesting multilingual treatment and could develop a single code addressing the different languages simultaneously. The main advantage is that one does not need to perform any ontology alignment. However, the proposed formalism clearly does not solve the problem of concepts that do not exist in all languages or have not the same meaning in different languages. For example the name `prefecture` relates to different concepts in France, Italy or Japan.

We have developed windows for displaying ontologies, editors for editing them, and web interfaces. MLNs were helpful to simplify programming. The MLN formalism is also used in multi-agent systems where personal assistants agents interface people using different natural languages in their dialogs, but access the same shared multilingual information, like in the NEWS project.

When modeling concepts, we first thought of an MLN as another value with a type different from a string or a number. The realization that it was equivalent to a multiple value by itself came only later. Indeed assigning several MLNs to a single property has no meaning unless one tags along meta information as who provided the data.

Alphabetical order. Another difficult point occurs when we want to display information in alphabetical order, since the

[5]MOSS with documentation are available at http://www.utc.fr/∼barthes/MOSS/

order is not simply the order of the UTF-8 codes. To obtain a proper order for the different languages, we first capitalize the string, then use a table giving the proper order in each language.

6.1 Discussion of the Different Formats

When designing the multilingual approach we thought of different possibilities. We could use packages, structures, pure strings, tagged lists, hash tables, property lists, a-lists, etc. All possibilities have advantages and drawbacks.

Using different packages is an interesting idea. However, it was conflicting with the use of packages in our multi-agent platform and was not further considered.

Using pure strings like

```
":en Beijing :fr Pékin;Beijing :zh 北京"
```

is elegant and easy to read but more difficult with sentences. On would have to use special separators and handling the string is somewhat expensive.

Using structs is too rigid. Using hash tables has some overhead for a small number of languages. Using a-lists is efficient but not very nice and adds some complexity with our treatment of versions (not discussed here):

```
((:en "Beijing") (:fr "Pékin" "Beijing") (:zh "北京"))
```

Using a tagged alternated list was our first choice:

```
(:name :en "Beijing" :fr "Pékin;Beijing" :zh "北京")
```

but we realized that the first element of the list was not really needed and thus subsequently removed it.

Thus our choice was on a simple disembodied property list, using a simple string to host the various synonyms. The drawback is the use of the reserved symbol semi-column for separating synonyms. We found that we could live with it.

One of the reviewers of this paper suggested the following format:

```
(:en ("Beijing") :fr ("Pékin" "Beijing") :zh ("北京"))
```

which could be a good compromise adding efficiency to the handling of synonyms.

Finally, considering simple strings as a particular case of MLN turns out to be quite useful.

6.2 Conclusion

The approach described in this paper has no linguistic ambition and is merely a simple tool to simplify programming. We would like to see it improved and inserted in the various Lisp dialects.

Clearly our experimental approach could be improved and more work needs to be done on issues like the choice of an efficient format for representing MLNs, the possibility of adding annotations like the origin of the value represented by an MLN, the interest of having `:all` and `:unknown` tags at run time in addition to input time, whether GUID codes could be used to intern MLNs and how this would fly with persistent storage, etc.

Acknowledgments

I would like to thanks the reviewers for their numerous inspiring remarks pointing to many possible ways of generalizing the work presented here, extending it to domains other that ontologies and knowledge bases.

Appendix - Function Library

We give a more detailed description of the functions we developed for dealing with MLNs, then for dealing wit synonyms.

Functions Dealing with Multilingual Names

All functions in this section are prefixed by %MLN- and we agree that we could drop this prefix and defined the functions in a specific "MLN" package.

Constructor

`%MAKE-MLN-FROM-REF (REF)` builds an MLN structure. If the argument is a string or a symbol, then uses the value of the `*language*` variable as the language tag. If the value is `:all`, then uses the `:unknown` tag. If the argument is already an MLN, then leaves it unchanged.

```
(%make-mln-from-ref "Paris") -> (:EN "Paris")
```

Predicates

`%MLN? (EXPR)` uses language tags to check if `EXPR` is a valid MLN, meaning that all language tags must belong to the list of valid languages for the application, `*language-tags*`.

`%MLN-EQUAL (MLN1 MLN2 &key LTAG)` Equality between two MLNs is true if they share some synonym for a given language. If one of the language tag of `MLN1` is `:unknown`, then the associated synonyms will be checked against the synonyms of all the languages of `MLN2`. If one of the values is a string then `*language*` is used to build the corresponding MLN before the comparison.

```
(%mln-equal '(:en "Beijing" :fr "Pékin") '(:fr
            "Beijing; Pékin")) -> T
```

`%MLN-IDENTICAL? (MLN1 MLN2)` Two MLNs are identical if they have the same synonyms for the same language.

`%MLN-INCLUDED? (MLN1 MLN2)` Checks if all synonyms of MLN1 are included those of MLN22 for all languages of MLN1.

`%MLN-IN? (INPUT-STRING LTAG MLN)` checks whether the input string is one of the synonyms of the MLN in the language specified by `LTAG`. If the tag is `:all` or `:unknown`, then we check against any synonym in any language.

```
(%mln-in?  "Beijing" '(:fr "Beijing; Pékin")) ->
                    NIL but
(let ((*language* :FR)) (%mln-in?  "Beijing" '(:fr
              "Beijing; Pékin")) -> T
```

Modifiers

The following functions are used to edit the values of an MLN.

`%MLN-ADD-VALUE (MLN VALUE LTAG)` adds a synonym corresponding to a specific language at the end of the list of synonyms.

```
(%mln-add-value '(:en "Beijing" :fr "Pékin")
 "Pékin" :FR) -> (:EN "Beijing" :FR "Pékin;
                  Beijing")
```

`%MLN-REMOVE-VALUE (MLN VALUE LTAG)` removes a synonym corresponding to a specific language from the list of synonyms.

```
(%mln-remove-value '(:en "Beijing" :fr "Pékin")
        "Beijing" :EN) -> (:FR "Pékin")
```

`%MLN-REMOVE-LANGUAGE (MLN LTAG)` removes the set of synonyms corresponding to a particular language.

```
(%mln-remove-language '(:en "Beijing" :fr "Pékin;
       Beijing") :FR) -> (:EN "Beijing")
```

`%MLN-SET-VALUE (MLN LTAG SYN-STRING)` sets the synonyms corresponding to a specific language. Synonyms are supposed to have the standard form of a string containing terms separated by semi-columns.

```
(%mln-set-value '(:en "Beijing" :fr "Beijing") :FR
     "Pékin") -> (:EN "Beijing" :FR "Pékin")
```

`%MLN-MERGE (&rest MLN)` merges a set of MLNs removing duplicated synonyms within the same language. This function is equivalent to an addition of two MLNs.

```
(%mln-merge '(:en "UK" :fr "Royaume Uni") '(:it
"Inghilterra") '(:fr "Angleterre") (:pl "Anglia"))
 -> (:EN "UK" :FR "Royaume Uni:  Angleterre" :IT
          "Inghilterra" :PL "Anglia")
```

Extractors

One of the problems when extracting a value from an MLN is related to what happens when the requested language has no entry.
`%MLN-EXTRACT-ALL-SYNONYMS (MLN)` extracts all synonyms as a list of strings regardless of the language.

```
(%mln-extract-all-synonyms '(:en "Beijing" :fr
       "Pékin")) -> ("Beijing" "Pékin")
```

`%MLN-FILTER-LANGUAGE (MLN LTAG)` extracts from the MLN the string corresponding to specified language. If MLN is a string returns the string. If language is `:all`, returns a string concatenating all the languages. If language is `:unknown`, returns the string associated with the `:unknown` tag. If language is not present, then returns nil.

```
(%mln-filter-language '(:en "Beijing" :fr
"Beijing; Pékin") :fr) -> "Beijing; Pékin"
```

`%MLN-GET-CANONICAL-NAME (MLN)` extracts from a multilingual name the canonical name. By default it is the first name corresponding to the value of `*language*`, or else the first name of the English entry, or else the name of the list. An error occurs when the argument is not multilingual name.

```
 (let ((*language* :fr))(%mln-get-canonical-name
'(:en "Beijing" :fr "Pékin; Beijing"))) -> "Pékin"
 (let ((*language* :it))(%mln-get-canonical-name
   '(:en "Beijing" :fr "Pékin; Beijing"))) ->
                  "Beijing"
```

`%MLN-EXTRACT (MLN &key (LTAG *LANGUAGE*) ALWAYS)` extracts from the MLN the string of synonyms corresponding to specified language. If MLN is a string returns the string. If language is `:all`, returns a string concatenating all the languages. If language is `:unknown`, returns the string associated with the `:unknown` tag. If `always` is t, then tries to return something: tries English, then `:unknown`, then first recorded language.
The `:always` option is interesting when dealing with multilingual ontologies, when one wants to obtain some value even when the concept has no marker in the current language.

```
(%mln-extract :it '(:en "Beijing" :fr "Pékin;
               Beijing")))-> NIL
(%mln-extract :it '(:en "Beijing" :fr "Pékin;
      Beijing") :always t) -> "Beijing"
```

Printer

`%MLN-PRINT-STRING (MLN &optional LTAG)` returns a nicely formatted string, e.g.

```
(%mln-print-string '(:en "Beijing" :fr "Pékin;
                Beijing")) ->
      "EN: Beijing - FR: Pékin ; Beijing"
```

Functions Dealing with Synonyms

We also need some functions to take care of the set of synonyms. As could be seen in the previous examples, synonyms are encoded in a single string, terms being separated by semi-columns. The functions dealing with synonyms are what one could expect, therefore no further examples are given here.

`%SYNONYM-ADD (SYN-STRING VALUE)` adds a value (term) to a string at the end, sending a warning if language tag does not exist. In that case does not add value.

`%SYNONYM-EXPLODE (TEXT)` takes a string, considers it as a synonym string and extracts items separated by a semicolumn. Returns the list of string items.

`%SYNONYM-MEMBER (VALUE SYN-STRING)` Checks if value, a string, is part of the synonym list. Uses the `%string-norm` function to normalize the strings before comparing.

`%SYNONYM-MERGE-STRINGS (&REST NAMES)` merges several synonym strings, removing duplicates (using a norm-string comparison) and preserving the order.

`%SYNONYM-REMOVE (VALUE SYN-STRING)` removes a synonym from the set of synonyms. If nothing is left returns the empty string.

`%SYNONYM-MAKE (&REST ITEM-LIST)` builds a synonym string with a list of items. Each item is coerced to a string. Returns a synonym string.

The set of synonyms is (partially) ordered, because in ontologies there is often a preferred term for labeling a concept. This term will appear first in the list and the extractor functions will take advantage of this position to retrieve this synonym.

Note. Representing synonyms as a list of strings would simplify the functions, namely `%synonym-explode` would simply be a `getf`, `%synonym-make` would not be needed.

7. REFERENCES

[1] J.-P. A. Barthès and C. Moulin. Moss: A formalism for ontologies including multilingual features. In *Proc. KSE*, volume 2, pages 95–107, 2013.

[2] F. Bettahar, C. Moulin, and J.-P. A. Barthès. Towards a semantic interoperability in an e-government application. *Electronic Journal of e-Government*, 7(3):209–226, 2009.

[3] M. S. Chaves, L. A. Freitas, and R. Vieira. Hontology: a multilingual ontology for the accommodation sector in the tourism industry. In *Proceedings of the 4th International Conference on Knowledge Engineering and Ontology Development*, pages 149–154. 4th International Conference on Knowledge Engineering and Ontology Development, Octobre 2012.

[4] D. Galanis and I. Androutsopoulos. Generating multilingual descriptions from linguistically annotated owl ontologies: the naturalowl system. In *in Proceedings of the 11th European Workshop on Natural Language Generation (ENLG 2007), Schloss Dagstuhl*, pages 143–146, 2007.

[5] K. Knight. Building a large ontology for machine translation. In *Proceedings of the Workshop on Human Language Technology*, HLT '93, pages 185–190, Stroudsburg, PA, USA, 1993. Association for Computational Linguistics.

[6] C. Meilicke, R. García Castro, F. Freitas, W. R. van Hage, E. Montiel-Ponsoda, R. Ribeiro de Azevedo, H. Stuckenschmidt, O. Sváb-Zamazal, V. Svátek, A. Tamilin, C. Trojahn, and S. Wang. MultiFarm: A benchmark for multilingual ontology matching. *Journal of Web Semantics*, 15(3):62–68, 2012. meilicke2012a Infra-Seals.

[7] D. Picca, A. M. Gliozzo, and A. Gangemi. Lmm: an owl-dl metamodel to represent heterogeneous lexical knowledge. In *LREC*, 2008.

An Implementation of Python for Racket

Pedro Palma Ramos
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
pedropramos@tecnico.ulisboa.pt

António Menezes Leitão
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

ABSTRACT

Racket is a descendent of Scheme that is widely used as a first language for teaching computer science. To this end, Racket provides DrRacket, a simple but pedagogic IDE. On the other hand, Python is becoming increasingly popular in a variety of areas, most notably among novice programmers. This paper presents an implementation of Python for Racket which allows programmers to use DrRacket with Python code, as well as adding Python support for other DrRacket based tools. Our implementation also allows Racket programs to take advantage of Python libraries, thus significantly enlarging the number of usable libraries in Racket.

Our proposed solution involves compiling Python code into semantically equivalent Racket source code. For the runtime implementation, we present two different strategies: (1) using a foreign function interface to directly access the Python virtual machine, therefore borrowing its data types and primitives or (2) implementing all of Python's data model purely over Racket data types.

The first strategy provides immediate support for Python's standard library and existing third-party libraries. The second strategy requires a Racket-based reimplementation of all of Python's features, but provides native interoperability between Python and Racket code.

Our experimental results show that the second strategy far outmatches the first in terms of speed. Furthermore, it is more portable since it has no dependencies on Python's virtual machine.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

General Terms

Languages

Keywords

Python; Racket; Language implementations; Compilers

1. INTRODUCTION

The Racket programming language is a descendent of Scheme, a language that is well-known for its use in introductory programming courses. Racket comes with DrRacket, a pedagogic IDE [2], used in many schools around the world, as it provides a simple and straightforward interface aimed at inexperienced programmers. Racket provides different language levels, each one supporting more advanced features, that are used in different phases of the courses, allowing students to benefit from a smoother learning curve. Furthermore, Racket and DrRacket support the development of additional programming languages [13].

More recently, the Python programming language is being promoted as a good replacement for Scheme (and Racket) in computer science courses. Python is a high-level, dynamically typed programming language [16, p. 3]. It supports the functional, imperative and object-oriented programming paradigms and features automatic memory management. It is mostly used for scripting, but it can also be used to build large scale applications. Its reference implementation, CPython, is written in C and it is maintained by the Python Software Foundation. There are also alternative implementations such as Jython (written in Java) and IronPython (written in C#).

According to Peter Norvig [11], Python is an excellent language for pedagogical purposes and is easier to read than Lisp for someone with no experience in either language. He describes Python as a dialect of Lisp with infix syntax, as it supports all of Lisp's essential features except macros. Python's greatest downside is its performance. Compared to, e.g., Common Lisp, Python is around 3 to 85 times slower for most tasks.

Despite its slow performance, Python is becoming an increasingly popular programming language on many areas, due to its large standard library, expressive syntax and focus on code readability.

In order to allow programmers to easily move between Racket and Python, we are developing an implementation of Python for Racket, that preserves the pedagogic advantages of DrRacket's IDE and provides access to the countless Python libraries.

As a practical application of this implementation, we are developing Rosetta [8], a DrRacket-based IDE, aimed at architects and designers, that promotes a programming-based approach for modelling three-dimensional structures. Although Rosetta's modelling primitives are defined in Racket, Rosetta integrates multiple programming languages, including Racket, JavaScript, and AutoLISP, with multiple computer-aided design applications, including AutoCAD and Rhinoceros 3D.

Our implementation adds Python support for Rosetta, allowing Rosetta users to program in Python. Therefore, this implementation must support calling Racket code from Python, using Racket as an interoperability platform. Being able to call Python code from Racket is also an interesting feature for Racket developers, by allowing them to benefit from the vast pool of existing Python libraries.

In the next sections, we will briefly examine the strengths and weaknesses of other Python implementations, describe the approaches we took for our own implementation and showcase the results we have obtained so far.

2. RELATED WORK

There are a number of Python implementations that are good sources of ideas for our own implementation. In this section we describe the most relevant ones.

2.1 CPython

CPython, written in the C programming language, has been the reference implementation of Python since its first release. It parses Python source code (from `.py` files or interactive mode) and compiles it to bytecode, which is then interpreted on a virtual machine.

The Python standard library is implemented both in Python and C. In fact, CPython makes it easy to write third-party module extensions in C to be used in Python code. The inverse is also possible: one can embed Python functionality in C code, using the Python/C API [15].

CPython's virtual machine is a simple stack-based machine, where the byte codes operate on a stack of `PyObject` pointers [14]. At runtime, every Python object has a corresponding `PyObject` instance. A `PyObject` contains a reference counter, used for garbage collection, and a pointer to a `PyTypeObject`, which specifies the object's type (and is also a `PyObject`). In order for every value to be treated as a `PyObject`, each built-in type is declared as a structure containing these two fields, plus any additional fields specific to that type. This means that everything is allocated on the heap, even basic types.

To avoid relying too much on expensive dynamic memory allocation, CPython makes use of memory pools for small memory requests. Additionally, it also pre-allocates commonly used immutable objects (such as the integers from -5 to 256), so that new references will point to these instances instead of allocating new ones.

Garbage collection in CPython is performed through reference counting. Whenever a new Python object is allocated or whenever a new reference to it is made, its reference counter is incremented. When a reference to an object is discarded, its reference counter is decremented. When it reaches zero, the object's finalizer is called and the space is reclaimed.

Reference counting, however, does not work well with reference cycles [17, ch. 3.1]. Consider the example of a list containing itself. When its last reference goes out of scope, its counter is decremented, however the circular reference inside the list is still present, so the reference counter will never reach zero and the list will not be garbage collected, even though it is already unreachable.

2.2 Jython and IronPython

Jython is an alternative Python implementation, written in Java and first released in 2000. Similarly to how CPython compiles Python source-code to bytecode that can be run on its virtual machine, Jython compiles Python source-code to Java bytecode, which can then be run on the Java Virtual Machine (JVM).

Jython programs cannot use extension modules written for CPython, but they can import Java classes, using the same syntax that is used for importing Python modules. It is worth mentioning that since Clojure targets the JVM, Jython makes it possible to import and use Clojure libraries from Python and vice-versa [5]. There is also work being done by a third-party [12] to integrate CPython module extensions with Jython, through the use of the Python/C API. This would allow popular C-based libraries such as NumPy and SciPy to be used with Jython.

Garbage collection in Jython is performed by the JVM and does not suffer from the issues with reference cycles that plague CPython [7, p. 57]. In terms of speed, Jython claims to be approximately as fast as CPython. Some libraries are known to be slower because they are currently implemented in Python instead of Java (in CPython these are written in C). Jython's performance is also deeply tied to performance gains in the Java Virtual Machine.

IronPython is another alternative implementation of Python, this one for Microsoft's Common Language Infrastructure (CLI). It is written in C# and was first released in 2006. Similarly to what Jython does for the JVM, IronPython compiles Python source-code to CLI bytecode, which can be run on the .NET framework. It claims to be 1.8 times faster than CPython on `pystone`, a Python benchmark for showcasing Python's features.

IronPython provides support for importing .NET libraries and using them with Python code [10]. There is also work being done by a third-party in order to integrate CPython module extensions with IronPython [6].

2.3 CLPython

CLPython (not to be confused with CPython, described above) is yet another Python implementation, written in Common Lisp. Its development was first started in 2006, but stopped in 2013. It supports six Common Lisp implementations: Allegro CL, Clozure CL, CMU Common Lisp, ECL, LispWorks and SBCL [1]. Its main goal was to bridge Python and Common Lisp development, by allowing access

to Python libraries from Lisp, access to Lisp libraries from Python and mixing Python and Lisp code.

CLPython compiles Python source-code to Common Lisp code, i.e. a sequence of s-expressions. These s-expressions can be interpreted or compiled to `.fasl` files, depending on the Common Lisp implementation used. Python objects are represented by equivalent Common Lisp values, whenever possible, and CLOS instances otherwise.

Unlike other Python implementations, there is no official performance comparison with a state-of-the-art implementation. Our tests (using SBCL with Lisp code compilation) show that CLPython is around 2 times slower than CPython on the `pystone` benchmark. However it outperforms CPython on handling recursive function calls, as shown by a benchmark with the Ackermann function.

2.4 PLT Spy

PLT Spy is an experimental Python implementation written in PLT Scheme and C, first released in 2003. It parses and compiles Python source-code into equivalent PLT Scheme code [9].

PLT Spy's runtime library is written in C and extended to Scheme via the PLT Scheme C API. It implements Python's built-in types and operations by mapping them to CPython's virtual machine, through the use of the Python/C API. This allows PLT Spy to support every library that CPython supports (including NumPy and SciPy).

This extended support has a big trade-off in portability, though, as it led to a strong dependence on the 2.3 version of the Python/C API library and does not seem to work out-of-the-box with newer versions. More importantly, the repetitive use of Python/C API calls and conversions between Python and Scheme types severely limited PLT Spy's performance. PLT Spy's authors use anecdotal evidence to claim that it is around three orders of magnitude slower than CPython.

2.5 Comparison

Table 1 displays a rough comparison between the implementations discussed above.

	Platform(s) targeted	Speedup (vs. CPython)	Std. library support
CPython	CPython's VM	1×	Full
Jython	JVM	∼ 1×	Most
IronPython	CLI	∼ 1.8×	Most
CLPython	Common Lisp	∼ 0.5×	Most
PLT Spy	Scheme	∼ 0.001×	Full

Table 1: Comparison between implementations

PLT Spy can interface Python code with Scheme code and is the only alternative implementation which can effortlessly support all of CPython's standard library and third-party modules extensions, through its use of the Python/C API. Unfortunately, there is a considerable performance cost that results from the repeated conversion of data from Scheme's internal representation to CPython's internal representation.

On the other hand, Jython, IronPython and CLPython show us that it is possible to implement Python's semantics over high-level languages, with very acceptable performances and still providing the means for importing that language's functionality into Python programs. However, Python's standard library needs to be manually ported.

Taking this into consideration, we developed a Python implementation for Racket that we present in the next section.

3. SOLUTION

Our proposed solution consists of two compilation phases: (1) Python source-code is compiled to Racket source-code and (2) Racket source-code is compiled to Racket bytecode.

In phase 1, the Python source code is parsed into a list of abstract syntax trees, which are then expanded into semantically equivalent Racket code.

In phase 2, the Racket source-code generated above is fed to a bytecode compiler which performs a series of optimizations (including constant propagation, constant folding, inlining, and dead-code removal). This bytecode is interpreted on the Racket VM, where it may be further optimized by a JIT compiler.

Note that phase 2 is automatically performed by Racket, therefore our implementation effort relies only on a source-to-source compiler from Python to Racket.

3.1 General Architecture

Fig. 1 summarises the dependencies between the different Racket modules of the proposed solution. The next paragraphs provide a more detailed explanation of these modules.

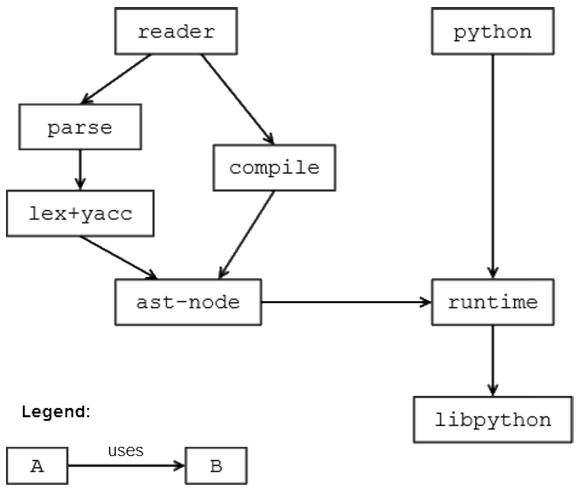


Figure 1: Dependencies between modules. The arrows indicate that a module uses functionality that is defined on the module it points to.

3.1.1 Racket Interfacing

A Racket file usually starts with the line `#lang <language>` to specify which language is being used (in our case, it will be `#lang python`). The entry-point for a `#lang` is at the `reader` module, visible at the top of **Fig. 1**. This module must provide the functions `read` and `read-syntax` [4, ch. 17.2].

The `read-syntax` function takes the name of the source file and an input port as arguments and returns a list of syntax objects, which correspond to the Racket code compiled from the input port. It uses the `parse` and `compile` modules to do so.

Syntax objects [4, ch. 16.2.1] are Racket's built-in data type for representing code. They contain the quoted form of the code (an s-expression), source location information (line number, column number and span) and lexical-binding information. By keeping the original source location information on every syntax object generated by the compiler, DrRacket can map each compiled s-expression to its corresponding Python code. This way, DrRacket's features for Racket code will also work for Python. Such features include the syntax checker, debugger, displaying source location for errors, tacking and untacking arrows for bindings and renaming variables.

3.1.2 Parse and Compile Modules

The `lex+yacc` module defines a set of Lex and Yacc rules for parsing Python code, using the `parser-tools` library. This outputs a list of abstract syntax trees (ASTs), which are defined in the `ast-node` module. These nodes are implemented as Racket objects. Each subclass of an AST node defines its own `to-racket` method, responsible for generating a syntax object with the compiled code and respective source location. A call to `to-racket` works in a top-down recursive manner, as each node will eventually call `to-racket` on its children.

The `parse` module simply defines a practical interface of functions for converting the Python code from an input port into a list of ASTs, using the functionality from the `lex+yacc` module. In a similar way, the `compile` module defines a practical interface for converting lists of ASTs into syntax objects with the compiled code, by calling the `to-racket` method on each AST.

3.1.3 Runtime Modules

The `libpython` module defines a foreign function interface to the functions provided by the Python/C API. Its use will be explained in detail on the next section.

Compiled code contains references to Racket functions and macros, as well as some additional functions which implement Python's primitives. For instance, we define `py-add` as the function which implements the semantics of Python's `+` operator. These primitive functions are defined in the `runtime` module.

Finally, the `python` module simply provides everything defined at the `runtime` module, along with all the bindings from the `racket` language. Thus, every identifier needed for the compiled code is provided by the `python` module.

3.2 Runtime Implementation using FFI

For the runtime, we started by following a similar approach to PLT Spy, by mapping Python's data types and primitive functions to the Python/C API. The way we interact with this API, however, is radically different.

On PLT Spy, this was done via the PLT Scheme C API, and therefore the runtime is implemented in C. This entails converting Scheme values into Python objects and vice-versa for each runtime call. Besides the performance issue (described on the Related Work section), this method lacks portability and is somewhat cumbersome for development, since it requires compiling the runtime module with a platform specific C compiler, and to do so each time this module is modified.

Instead, we used the Racket Foreign Function Interface (FFI) to directly interact with the foreign data types created by the Python/C API, therefore our runtime is implemented in Racket. These foreign functions are defined on the `libpython` modules, according to their C signatures, and are called by the functions and macros defined on the `runtime` module.

The values passed around correspond to pointers to objects in CPython's virtual machine, but there is sometimes the need to convert them back to Racket data types, so they can be used as conditions in flow control forms like `if`s and `cond`s.

As with PLT Spy, this approach only requires implementing the Python language constructs, because the standard library and other libraries installed on CPython's implementation are readily accessible.

Unfortunately, as we will show in the Performance section, the repetitive use of these foreign functions introduces a significant overhead on our primitive operators, resulting in a very slow implementation.

Another issue is that the Python objects allocated on CPython's VM must have their reference counters explicitly decremented or they will not be garbage collected. This issue can be solved by attaching a Racket finalizer to every FFI function that returns a new reference to a Python object. This finalizer will decrement the object's reference counter whenever Racket's GC proves that there are no more live references to the Python object. On the other hand, this introduces another significant performance overhead.

3.3 Runtime Implementation using Racket

Our second approach is a pure Racket implementation of Python's data model. Comparing it to the FFI approach, this one entails implementing all of Python's standard library in Racket, but, on the other hand, it is a much faster implementation and provides reliable memory management of Python's objects, since it does not need to coordinate with another virtual machine.

3.3.1 Object Model

In Python, every object has an associated *type-object* (where every type-object's type is the `type` type-object). A type-object contains a list of base types and a hash table which maps operation names (strings) to the functions that type supports (function pointers, in CPython).

As a practical example, in the expression `obj1 + obj2`, the behaviour of the plus operator depends on the type of its operands. If `obj1` is a number this will be the addition operator. If it is a string, this will be a string concatenation. Additionally, a user-defined class can specify another behaviour for the plus operator by defining the method `__add__`. This is typically done inside a class definition, but can also be done after the class is defined, through reflection.

CPython stores each object's type as a pointer in the `PyObject` structure. Since an object's type is not known at compile-time, method dispatching must be done at runtime, by obtaining `obj1`'s type-object and looking up the function that is mapped by the string `__add__` on its hash table. If there is no such entry, the search continues on that type-object's base types.

While the same mechanics would work in Racket, there is room for optimization. In Racket, one can recognize a value's type through its predicate (`number?`, `string?`, etc.). In Python, a built-in object's type is not allowed to change, so we can directly map basic Racket types into Python's basic types. Their types are computed through a pattern matching function, which returns the most appropriate type-object, according to the predicates that value satisfies. Complex built-in types are still implemented through Racket structures (which include a reference to the corresponding type-object).

This way, we avoid the overhead from constantly wrapping and unwrapping frequently used values from the structures that hold them. Interoperability with Racket data types is also greatly simplified, eliminating the need to wrap/unwrap values when using them as arguments or return values from functions imported from Racket.

There is also an optimization in place concerning method dispatching. Despite the ability to add new behaviour for operators in user-defined classes, a typical Python program will mostly use these operators for numbers (and strings, in some cases). Therefore, each operator implements an early dispatch mechanism for the most typical argument types, which skips the heavier dispatching mechanism described above. For instance, the plus operator is implemented as such:

```
(define (py-add x y)
  (cond
    [(and (number? x) (number? y)) (+ x y)]
    [(and (string? x) (string? y)) (string-append x y)]
    [else (py-method-call x "__add__" y)]))
```

3.3.2 Importing Modules

In Python, files can be imported as modules, which contain bindings for defined functions, defined classes and global assignments. Unlike in Racket, Python modules are first-class citizens. There are 3 ways to import modules in Python: (1) the `import <module>` syntax, which imports <module> as a module object whose bindings are accessible as attributes; (2) the `from <module> import <binding>` syntax, which only imports the declared <binding> from <module>; (3) the `from <module> import *` syntax, which imports all bindings from <module>.

To implement the first syntax, we make use of `module->exports` to get a list of the bindings provided by a module and `dynamic-require` to import each one of them and store them in a new module object. The other two syntaxes are semantically similar to Racket's importing model and, therefore, are implemented with `require` forms.

This implementation of the import system was designed to allow importing both Python and Racket modules. We have come up with a slightly different syntax for referring to Racket modules. They are specified as a string literal containing a Racket module path (following the syntax used for a `require` form [3, ch. 3.2]).

This way we support importing bindings from the Racket library, Racket files or packages hosted on PLaneT (Racket's centralized package distribution system), using any of the Python importing syntaxes mentioned above. The following example shows a way to access the Racket functions `cons`, `car` and `cdr` in a Python program.

```
1  #lang python
2  import "racket" as racket
3
4  def add_cons(c):
5    return racket.car(c) + racket.cdr(c)
6
7  c1 = racket.cons(2, 3)
8  c2 = racket.cons("abc", "def")
```

```
> add_cons(c1)
5
> add_cons(c2)
"abcdef"
```

Since the second and third syntaxes above map to `require` forms (which are evaluated before macro expansion), it is also possible to use Racket-defined macros with Python code.

Predictably, importing Python modules into Racket programs is also possible and straightforward. Function definitions, class definitions and top-level assignments are `define`'d and `provide`'d in the compiled Racket code, therefore they can be `require`'d in Racket.

3.3.3 Class Definitions

A class definition in Python is just syntactic sugar for defining a new type-object. Its hash table will contain the variables and methods defined within the class definition. Therefore, an instance of a class is an object like any other, whose type-object is its class. The main distinction is that an instance of a class also contains its own hash table, where its attributes are mapped to their values.

3.3.4 Exception Handling

Both Python and Racket support exceptions in a similar way. In Python, one can only raise objects whose type derives from `BaseException`, while in Racket, any value can be raised and caught.

In Python, exceptions are raised with the `raise` statement and caught with the `try...except` statement (with optional

`else` and `finally` clauses). Their semantics can be implemented with Racket's `raise` and `with-handlers` forms, respectively. The latter expects an arbitrary number of pairs of predicate and procedure. Each predicate is responsible for recognizing a specific exception type and the procedure is responsible for handling it.

The exceptions themselves can be implemented as Racket exceptions. In fact, some of Python's built-in exceptions can be defined as their equivalents in Racket, for added interoperability. For instance, Python's `ZeroDivisionError` can be mapped to Racket's `exn:fail:contract:divide-by-zero` and Python's `NameError` is mapped to Racket's `exn:fail:contract:variable`.

4. EXAMPLES

In this section we provide some examples of the current state of the translation between Python and Racket. Note that this is still a work in progress and, therefore, the compilation results of these examples may change in the future.

4.1 Ackermann

Consider the following program in Racket which implements the Ackermann function and calls it with arguments $m = 3$ and $n = 9$:

```
(define (ackermann m n)
  (cond
   [(= m 0) (+ n 1)]
   [(and (> m 0) (= n 0)) (ackermann (- m 1) 1)]
   [else (ackermann (- m 1) (ackermann m (- n 1)))]))

(ackermann 3 9)
```

Its equivalent in Python would be:

```
def ackermann(m,n):
    if m == 0: return n+1
    elif m > 0 and n == 0: return ackermann(m-1,1)
    else: return ackermann(m-1, ackermann(m,n-1))

print ackermann(3,9)
```

Currently, this code is compiled to:

```
(provide :ackermann)
(define-py-function :ackermann with-params (m n)
  (lambda (:m :n)
    (cond
      [(py-truth (py-eq :m 0))
       (py-add :n 1)]
      [(py-truth (py-and (py-gt :m 0) (py-eq :n 0)))
       (py-call :ackermann (py-sub :m 1) 1)]
      [else
       (py-call
        :ackermann
        (py-sub :m 1)
        (py-call :ackermann :m (py-sub :n 1)))])))

(py-print (py-call :ackermann 3 9))
```

The first thing one might notice is the colon prefixing the identifiers `ackermann`, `m` and `n`. This has no syntactic meaning in Racket; it is simply a name mangling technique to avoid replacing Racket's bindings with bindings defined in Python. For example, one might set a variable `cond` in Python, which would then be compiled to `:cond` and therefore would not interfere with Racket's built-in `cond`.

The `(define-py-function ... with-params ...)` macro builds a function structure, which is essentially a wrapper for a lambda and a list of the argument names. The need to store a function's argument names arises from the fact that in Python a function can be called both with positional or keyword arguments. A function call without keyword arguments is handled by the `py-call` macro, which simply expands to a traditional Racket function call. If the function is called with keyword arguments, this is handled by `py-call/keywords`, which rearranges the arguments' order at runtime.

This way, we can use the same syntax for calling both Python user-defined functions and Racket functions. On the other hand, since the argument names are only stored with Python user-defined functions, it is not possible to use keyword arguments for calling Racket functions.

The functions/macros `py-eq`, `py-and`, `py-gt`, `py-add` and `py-sub` are defined on the `runtime` module and implement the semantics of the Python operators `==`, `and`, `>`, `+`, `-`, respectively.

The function `py-truth` takes a Python object as argument and returns a Racket boolean value, `#t` or `#f`, according to Python's semantics for boolean values. This conversion is necessary because, in Racket, only `#f` is treated as false, while, in Python, the boolean value false, zero, the empty list and the empty dictionary, among others, are all treated as false when used on the condition of an `if`, `for` or `while` statement. Finally, the function `py-print` implements the semantics of the print statement.

4.2 Mandelbrot

Consider now a Racket program which defines and calls a function that computes the number of iterations needed to determine if a complex number c belongs to the Mandelbrot set, given a limited number of *limit* iterations.

```
(define (mandelbrot limit c)
  (let loop ([i 0]
             [z 0+0i])
    (cond
     [(> i limit) i]
     [(> (magnitude z) 2) i]
     [else (loop (add1 i)
                 (+ (* z z) c))])))

(mandelbrot 1000000 .2+.3i)
```

Its Python equivalent could be implemented like such:

```
def mandelbrot(limit, c):
    z = 0+0j
    for i in range(limit+1):
        if abs(z) > 2:
            return i
        z = z*z + c
    return i+1

print mandelbrot(1000000, .2+.3j)
```

This program demonstrates some features which are not straightforward to map in Racket. For example, in Python we can assign new local variables anywhere, as shown in line 2, while in Racket they become parameters of a named `let` form.

Another feature, present in most programming languages but not in Racket, is the `return` keyword, which immediately returns to the point where the function was called, with a given value. On the former example, all returns were tail statements, while on this one we have an early return, on line 5.

The program is compiled to:

```
1  (provide :mandelbrot)
2  (define-py-function :mandelbrot with-params (limit c)
3    (lambda (:limit :c)
4      (let ([:i (void)]
5            [:z (void)])
6        (let/ec return9008
7          (set! :z (py-add 0 0))
8          (py-for continue9007
9            [:i (py-call :range (py-add :limit 1))]
10           (begin
11             (cond
12               [(py-truth (py-gt (py-call :abs :z) 2))
13                (return9008 :i)]
14               [else py-None])
15             (set! :z (py-add (py-mul :z :z) :c))))
16         (return9008 (py-add :i 1))))))
17
18 (py-print
19  (py-call :mandelbrot 1000000 (py-add 0.2 0+0.3i)))
```

You will notice the `let` form on lines 4-5. The variables `:i` and `:z` are declared with a void value at the start of the function definition, allowing us to simply map Python assignments to `set!` forms.

Early returns are implemented as escape continuations, as seen on line 6: there is a `let/ec` form (syntactic sugar for a `let` and a `call-with-escape-continuation`) wrapping the body of the function definition. With this approach, a return statement is as straightforward as calling the escape continuation, as seen on line 13.

Finally, `py-for` is a macro which implements Python's for loop. It expands to a named `let` which updates the control variables, evaluates the `for`'s body and recursively calls itself, repeating the cycle with the next iteration. Note that calling this named `let` has the same semantics as a `continue` statement.

In fact, although there was already a `for` form in Racket with similar semantics as Python's, the latter allows the use of `break` and `continue` as flow control statements. The `break` statement can be implemented as an escape continuation and `continue` is implemented by calling the named `let`, thus starting a new iteration of the loop.

5. PERFORMANCE

The charts on **Fig. 2** compare the running time of these examples for:

- **(a)** Racket code running on Racket;
- **(b)** Python code running on CPython;
- **(c.1)** Python code running on Racket with the FFI runtime approach, without finalizers
- **(c.2)** Python code running on Racket with the FFI runtime approach, with finalizers for garbage collecting Python objects
- **(d.1)** Python code running on Racket with the pure Racket runtime approach
- **(d.2)** Python code running on Racket with the pure Racket runtime approach, using early dispatch for operators

These benchmarks were performed on an Intel® Core™ i7 processor at 3.2GHz running under Windows 7. The times below represent the minimum out of 3 samples.

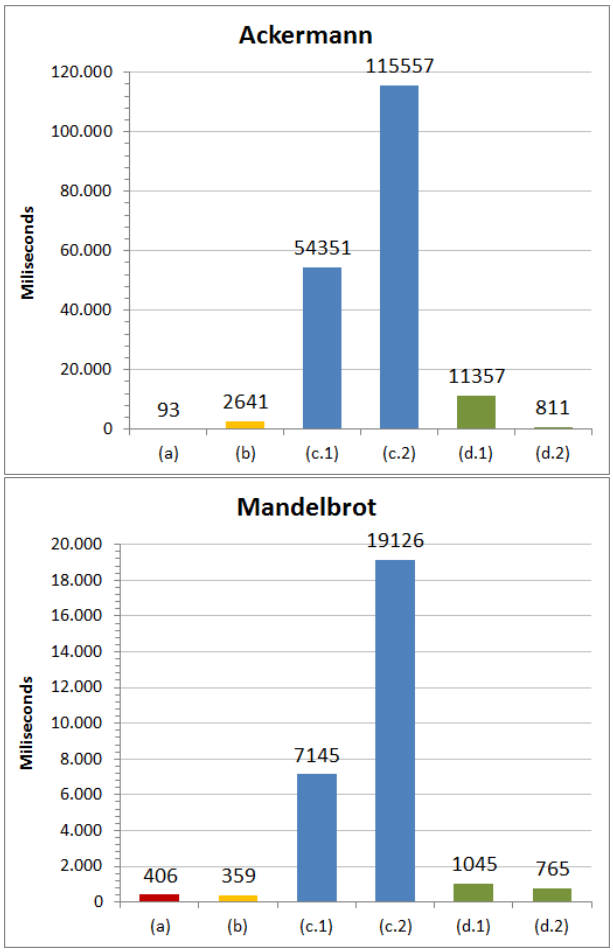


Figure 2: Benchmarks of the Ackermann and Mandelbrot examples

The Racket implementation of the Ackermann example is about 28 times faster than Python's implementation, but the Mandelbrot example's implementation happens to be slightly slower than Python's. This is most likely due to

Racket's lighter function calls and operators, since the Ackermann example heavily depends on them.

Since the FFI based runtime uses CPython's primitives, we have to endure with sluggish foreign function calls for every Python operation and we also cannot take advantage of Racket's lightweight mechanics, therefore the same Python code runs about 20 times slower on our implementation than in CPython, for both examples. This figure more than doubles if we consider the use of finalizers, in order to avoid a memory leak.

Moving to a pure Racket runtime yielded a great improvement over the FFI runtime, since it eliminated the need for foreign function calls, synchronizing garbage collection with another virtual machine and type conversions. With this transition, both examples run at around 3 to 4 times slower than in CPython, which is very tolerable for our goals.

Optimizing the dispatching mechanism of operators for common types further led to huge gains in the Ackermann example pushing it below the running time for CPython. The Mandelbrot example is still slower than in CPython, but nonetheless it has also benefited from this optimization.

6. CONCLUSIONS

A Racket implementation of Python would benefit Racket developers giving them access to Python's huge standard library and the ever-growing universe of third-party libraries, as well as Python developers by providing them with a pedagogic IDE in DrRacket. To be usable, this implementation must allow interoperability between Racket and Python programs and should be as close as possible to other state-of-the-art implementations in terms of performance.

Our solution tries to achieve these qualities by compiling Python source-code to semantically equivalent Racket source-code, using a traditional compiler's approach: a pipeline of scanner, parser and code generation. This Racket source-code is then handled by Racket's bytecode compiler, JIT compiler and interpreter.

We have come up with two alternative solutions for implementing Python's runtime semantics in Racket. The first one consists of using Racket's Foreign Interface and the Python/C API to manipulate Python objects in Python's virtual machine. This allows our implementation to effortlessly support all of Python's standard library and even third-party libraries written in C. On the other hand, it suffers from bad performance (at least one order of magnitude slower than CPython).

Our second approach consists of implementing Python's data model and standard library purely in Racket. This leads to a greater implementation effort, but offers a greater performance, currently standing at around the same speed as CPython, depending on the application. Additionally, it allows for a better integration with Racket code, since many Python data types are directly mapped to Racket data types.

Our current strategy consists of implementing the language's essential features and core libraries using the second approach (for performance and interoperability). Future efforts may include developing a mechanism to import modules from CPython through the FFI approach, in a way that is compatible with our current data model.

7. ACKNOWLEDGMENTS

This work was partially supported by Portuguese national funds through Fundação para a Ciência e a Tecnologia under contract Pest-OE/EEI/LA0021/2013 and by the Rosetta project under contract PTDC/ATP-AQI/5224/2012.

8. REFERENCES

[1] W. Broekema. CLPython - an implementation of Python in Common Lisp. `http://common-lisp.net/project/clpython/`. [Online; retrieved on March 2014].

[2] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of functional programming*, 12(2):159–182, 2002.

[3] M. Flatt. *The Racket Reference*, 2013.

[4] M. Flatt and R. B. Findler. *The Racket Guide*, 2013.

[5] E. Franchi. Interoperability: from Python to Clojure and the other way round. In *EuroPython 2011*, Florence, Italy, 2011.

[6] Ironclad - Resolver Systems. `http://www.resolversystems.com/products/ironclad/`. [Online; retrieved on January 2014].

[7] J. Juneau, J. Baker, F. Wierzbicki, L. M. Soto, and V. Ng. *The definitive guide to Jython*. Springer, 2010.

[8] J. Lopes and A. Leitão. Portable generative design for CAD applications. In *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture*, pages 196–203, 2011.

[9] P. Meunier and D. Silva. From Python to PLT Scheme. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, pages 24–29, 2003.

[10] Microsoft Corporation. *IronPython .NET Integration documentation*. `http://ironpython.net/documentation/`. [Online; retrieved on January 2014].

[11] P. Norvig. Python for Lisp programmers. `http://norvig.com/python-lisp.html`. [Online; retrieved on March 2014].

[12] S. Richthofer. JyNI - using native CPython-extensions in Jython. In *EuroSciPi 2013*, Brussels, Belgium, 2013.

[13] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. *ACM SIGPLAN Notices*, 46(6):132–141, 2011.

[14] P. Tröger. Python 2.5 virtual machine. `http://www.troeger.eu/files/teaching/pythonvm08.pdf`, April 2008. [Lecture at Blekinge Institute of Technology].

[15] G. van Rossum and F. L. Drake. *Extending and embedding the Python interpreter*. Centrum voor Wiskunde en Informatica, 1995.

[16] G. van Rossum and F. L. Drake. *An introduction to Python*. Network Theory Ltd., 2003.

[17] G. van Rossum and F. L. Drake. *The Python Language Reference*. Python Software Foundation, 2010.

Defmacro for C: Lightweight, Ad Hoc Code Generation

Kai Selgrad[1] Alexander Lier[1] Markus Wittmann[2] Daniel Lohmann[1] Marc Stamminger[1]

[1] Friedrich-Alexander University Erlangen-Nuremberg

[2] Erlangen Regional Computing Center

{kai.selgrad, alexander.lier, markus.wittmann, daniel.lohmann, marc.stamminger}@fau.de

ABSTRACT

We describe the design and implementation of CGen, a C code generator with support for Common Lisp-style macro expansion. Our code generator supports the simple and efficient management of variants, ad hoc code generation to capture reoccurring patterns, composable abstractions as well as the implementation of embedded domain specific languages by using the Common Lisp macro system. We demonstrate the applicability of our approach by numerous examples from small scale convenience macros over embedded languages to real-world applications in high-performance computing.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*pretty printers*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*evolutionary prototyping*

General Terms

Design, Languages, Experimentation, Management, Performance

Keywords

Code Generation, Common Lisp, Configurability, Maintenance, Macros, Meta Programming

1. INTRODUCTION

Code generation and its application in domain-specific languages is a long-established method to help reduce the amount of code to write, as well as to express solutions much closer to the problem at hand. In Lisp the former is provided by `defmacro`, while the latter is usually accomplished through its application. In this paper we present a formulation of C (and C-like languages in general) that is amenable to transformation by the Lisp macro processor.

With this formulation we strive towards providing more elegant and flexible methods of code configuration and easing investigation of different variants during evaluation (e.g. to satisfy performance requirements) without additional costs at run-time. The significance of this can be seen by the vast amount of work on variant management [26, 30], code generation and domain-specific languages for heterogeneous systems (e.g. [21, 17]) and code optimization in general [23] in the last years. It is, e.g., mandatory in performance critical applications to reevaluate different versions of an algorithm as required by advances in hardware and systems design (see e.g. [9, 18]). We believe that those approaches to algorithm evaluation will become ever more common in an increasing number of computational disciplines. Our contribution is the description and demonstration of a system that leverages the well established Common Lisp macro system to the benefit of the C family of languages. Additionally this extends to lowering the entry barrier for using meta code by providing a system that is much more suited to ad hoc code generation than current large-scale approaches.

In contrast to stand-alone domain-specific languages that generate C code, such as Yacc [12], most general purpose generative programming methods for C can be placed into two categories: string-based approaches, and systems based on completely parsed and type-checked syntax trees (ASTs). The systems of the former category (e.g. [5, 4]) tend to be suitable for ad hoc code generation, and for simple cases tackling combinatoric complexity (e.g. [9]), but lack layering capabilities (i.e. transforming generated code). Furthermore they suffer from using different languages in the same file (a problem described by [18], too), and thereby encompass problems including complicated scoping schemes. AST-based systems, on the other hand, are very large systems which are not suitable for ad hoc code generation. Even though such systems are very powerful, they are mostly suited for highly specialized tasks. Examples of such larger scopes include product line parameterization [26] and DSLs embedded into syntactically challenging languages [28, 21].

With respect to this classification our approach covers a middle-ground between these two extremes. Our formulation of C facilitates the use of Common Lisp macros and thereby light-weight structural and layered meta programming. Yet, we neither provide nor strive for a completely analyzed syntax tree as this would introduce a much larger gap between the actual language used and its meta code.

European Lisp Symposium 2014, Paris, France

Based on our reformulation of C, and tight integration into the Common Lisp system we present a framework that is most suitable for describing domain-specific languages in the C family. We therefore adopt the notion of our system being a meta DSL.

This paper focuses on the basic characteristics of CGen, our implementation of the approach described above. Section 3 presents CGen's syntax and shows simple macro examples to illustrate how input Lisp code is mapped to C code. Section 4 discusses the merits and challenges of directly integrating the CGen language into a Common Lisp system and shows various implementation details. A systematic presentation of more advanced applications with our method is given in Section 5, focusing on how our code generator works on different levels of abstraction and in which way they can be composed. Section 6 evaluates two rather complete and relevant examples found in high performance computing applications [3]. We analyze the abstractions achieved and compare the results to hand-crafted code in terms of variant management, extensibility and maintenance overhead. Section 7 concludes our paper by reflecting our results. Throughout this paper we provide numerous examples to illustrate our generator's capabilities as well as the style of programming it enables.

2. RELATED WORK

Since C is the de facto assembly language for higher level abstractions and common ground in programming and since generative programming [8] is as old as programming itself, there is an enormous amount of previous work in code generation targetting C. We therefore limit our scope to describe the context of our work and describe its relation to established approaches.

Code generation in C is most frequently implemented using the C preprocessor (and template meta programming in C++ [1]). These generators are most commonly used because they are ubiquitous and well known. They are, however, neither simple to use nor easily maintained [25, 8].

The traditional compiler tools, Yacc [12] and Lex [20], generate C from a very high level of abstraction while still allowing for embedding arbitrary C code fragments. Using our framework such applications could be remodelled to be embedded in C (similar to [7]), instead of the other way around. For such specialized applications (and established tools) this may, however, not be appropriate.

Our approach is more comparable to general purpose code generators. As detailed in Section 1 we divide this area into two categories: ad hoc string-based generators, very popular in dynamic languages (e.g. the Python based frameworks Cog [4] and Mako [5]); and large-scale systems (e.g. Clang [28], an extensible C++ parser based on LLVM[19]; AspectC++ [26], an extension of C++ to support aspect-oriented programming (AOP)[1]; XVCL [30], a language-agnostic XML-based frame processor) which are most appropriate when tackling large-scale problems. An approach that is conceptually similar to ours is "Selective Embedded JIT Specialization" [6] where C code is generated on the fly and in a programmable fashion and Parenscript [24], an S-Expression notation for JavaScript.

Regarding the entry barrier and our system's applicability to implement simple abstractions in a simple manner our system is close to string and scripting language-based methods. Due to homoiconicity we do not, however, suffer from problems arising because of mixed languages. Furthermore our approach readily supports layering abstractions and modifying generated code to the extent of implementing domain-specific languages in a manner only possible using large-scale systems. The key limitation of our approach is that the macro processor does not know about C types and cannot infer complicated type relations. Using the CLOS [13] based representation of the AST generated internally after macro expansion (see Section 4), any application supported by large-scale systems becomes possible; this is, however, not covered in this paper.

[1]The origins of AOP are from the Lisp community, see [16].

3. AN S-EXPRESSION SYNTAX FOR C

The key component facilitating our approach is a straightforward reformulation of C code in the form of S-Expressions. The following two examples, taken from the classic K&R [14], illustrate the basic syntax.

The first example, shown in Figure 1, is a simple line counting program. Even though the syntax is completely S-Expression-based, it still resembles C at a more detailed level. Functions are introduced with their name first, followed by a potentially empty list of parameters and (notationally inspired by the new C++11 [27] syntax) a return type after the parameter list. Local variables are declared by `decl` which is analogous to `let`.

```
(function main () -> int
  (decl ((int c)
         (int nl 0))
    (while (!= (set c (getchar)) EOF)
      (if (== c #\newline)
          ++nl))
    (printf "%d\n" nl))
  (return 0))
```

```
int main(void) {
    int c;
    int nl = 0;
    while ((c = getchar()) != EOF) {
        if (c == '\n')
            ++nl;
    }
    printf("%d\n", nl);
    return 0;
}

```

Figure 1: A simple line counting program, followed by the C program generated from it.

```
(function strcat ((char p[]) (char q[])) -> void
  (decl ((int i 0) (int j 0))
    (while (!= p[i] #\null)
      i++)
    (while (!= (set p[i++] q[j++]) #\null))))
```

```
void strcat(char p[], char q[]) {
    int i = 0;
    int j = 0;
    while (p[i] != '\0')
        i++;
    while ((p[i++] = q[j++]) != '\0');
}

```

Figure 2: Implementation of the standard library's `strcat`, followed by the generated code.

The resemblance of C is even more pronounced in the second example presented in Figure 2, which shows an implementation of the standard `strcat` function. The use of arrays, with possibly simple expressions embedded, is a shorthand syntax that is converted into the more general `(aref <array> <index>)` notation. This more elaborate notation is required for complicated index computations and ready to be analyzed in macro code.

To illustrate the application of simple macros we show how to add a function definition syntax with the same ordering as in C.

```
(defmacro function* (rt name params &body body)
  `(function ,name ,params -> ,rt
     ,@body))
```

With this the following two definitions are equivalent:

```
(function  foo () -> int (return 0))
(function* int foo () (return 0))
```

As another example consider an implementation of `swap` which exchanges two values and can be configured to use an external variable for intermediate storage. This can be implemented by generating a call to an appropriately surrounded internal macro.

```
(defmacro swap (a b &key (tmp (gensym) tmp-set))
  `(macrolet ((swap# (a b tmp)
                `(set ,tmp ,a
                      ,a   ,b
                      ,b   ,tmp)))
     (lisp (if ,tmp-set
               (cgen (swap# ,a ,b ,tmp))
               (cgen (decl ((int ,tmp))
                       (swap# ,a ,b ,tmp)))))))
```

The `lisp` form is outlined in Section 4. The following examples illustrate the two use cases (input code left, corresponding output code right).

```
(decl ((int x)                  int x;
       (int y))                 int y;
  (swap x y))                   int g209;   // gensym
                                g209 = x;
                                x = y;
                                y = g209;

(decl ((int x)                  int x;
       (int y)                  int y;
       (int z))                 int z;
  (swap x y :tmp z))            z = x;
                                x = y;
                                y = z;
```

Note the use of a gensym for the name of the temporary variable to avoid symbol clashes. More advanced applications of the macro system are demonstrated in Section 5 and 6.

4. IMPLEMENTATION DETAILS

Our system is an embedded domain-specific language for generating C code, tightly integrated into the Common Lisp environment. The internal data structure is an AST which is constructed by evaluating the primitive CGen forms. This implies that arbitrary Lisp forms can be evaluated during the AST's construction; consider e.g. further syntactic enhancements implemented by cl-yacc [7]:

```
(function foo ((int a) (int b) (int c)) -> int
  (return (yacc-parse (a + b * a / c))))
```

All CGen top level forms are compiled into a single AST which is, in turn, processed to generate the desired C output. The following listing shows the successive evaluation steps of a simple arithmetic form which is converted into a single branch of the enclosing AST.

```
(* (+ 1 2) x)
(* #<arith :op '+ :lhs 1 :rhs 2>
   #<name :name "x">)
#<arith :op '*
        :lhs #<arith :op '+ :lhs 1 :rhs 2>
        :rhs #<name :name "x">>
```

Naturally, the implementation of this evaluation scheme must carefully handle ambiguous symbols (i.e. symbols used for Lisp and CGen code), including arithmetic operations as shown in the example above, as well as standard Common Lisp symbols such as `function`, `return`, etc. We chose not to use awkward naming schemes and to default to the CGen interpretation for the sake of convenience. If the Lisp interpretation of an overloaded name is to be used, the corresponding form can be evaluated in a `lisp` form. Similarly the `cgen` form can be used to change back to its original context from inside a Lisp context. This scheme is implemented using the package system. CGen initially uses the `cg-user` package which does not include any of the standard Common Lisp symbols but defines separate versions defaulting to the CGen interpretation. Note that while ambiguous names depend on the current context, unique symbols are available in both contexts.

Considering the above example, we see that the symbol `x` is converted into a node containing the string `"x"`. While Lisp systems usually up-case symbols as they are read, this behavior would not be tolerated with C, especially when the generated code is to interact with native C code. To this end we set the reader to use `:invert` mode case conversion (`:preserve` would not be desirable as this would require using upper case symbol names for all of the standard symbols in most Common Lisp implementations). This scheme leaves the symbol names of CGen code in an inverted state which can easily be compensated for by inverting the symbol names again when they are printed out.

The AST itself is represented as a hierarchy of objects for which certain methods, e.g. traversal and printing, are defined. Naturally, this representation is well suited for extensions. To this end we implemented two different languages which we consider able to be classified as part of the family of C languages. The first language is a notation for CUDA [22], a language used for applying graphics processing units (GPUs) to general purpose computing. Support for CUDA was completed by adding a few node types, e.g. to support the syntax for calling a GPU function from the host side. The second extended C language is GLSL [15], a language used to implement GPU shader code for computer graphics applications. Supporting GLSL was a matter of adding a few additional qualifiers to the declaration syntax (to support handling uniform storage). These examples show how easily our method can be used to provide code for heterogeneous platforms, i.e. to support generating code that can run on different hardware where different (C-like) languages are used for program specification.

As noted previously, our system's AST representation is easily extensible to support any operation expected from a compiler. Our focus is, however, the application of the supported macro system and we therefore leave most of the backend operation to the system's C compiler. Since the AST is only available after macro expansion compilation errors are reported in terms of the expanded code.

5. APPLICATION

In this section we demonstrate how our generator can be applied to a number of different problems. We chose to show unrelated examples on different abstraction levels to illustrate its broad spectrum.

5.1 Ad Hoc Code Generation

A key aspect of our method is the support for ad hoc code generation, i.e. the implementation of localized abstractions as they become apparent during programming.

A simple example of this would be unrolling certain loops or collecting series of expressions. This can be accomplished by the following macro (`defcollector`) which generates macros (`unroll`, `collect`) that take as parameters the name of the variable to use for the current iteration counter, the start and end of the range and the loop body which will be inserted repeatedly.

```
(defmacro defcollector (name list-op)
  `(defmacro ,name ((var start end) &body code)
     `(,',list-op
       ,@(loop for i from start to end collect
           `(symbol-macrolet ((,var ,i))
              ,@code)))))

(defcollector unroll progn)
(defcollector collect clist)
```

The above defined `collect` macro can be used, e.g., to generate tables:

```
(decl ((double sin[360]
          (collect (u 0 359)
            (lisp (sin (* pi (/ u 180.0))))))))
```

The resulting code is entirely static and should not require run-time overhead to initialize the table:

```
double sin[360] = {0.00000, 0.01745, 0.03490, ...};
```

Clearly, many more advanced loop transformation methods could be applied, such as 'peeling' as demonstrated in Section 6.2.

5.2 Configuring Variants

The most straight-forward application of variant-selection is using templates. This can be as simple as providing basic type names, e.g. in a matrix function, and as elaborate as redefining key properties of the algorithm at hand, as shown in the following as well as in Section 6.

Figure 3 shows a rather contrived example where the manner in which a graph is traversed is decoupled from the action at each node. This is not an unusual setup. In our approach, however, there is no run-time cost associated with this flexibility. In this example the traversal method used is given to a macro (`find-max`) which embeds its own code into the body of the expansion of this traversal. This kind of expansion is somewhat similar to compile-time `:before` and `:after` methods.

We assert that having this kind of flexibility without any run-time costs at all allows for more experimentation in performance-critical code (which we demonstrate in Section 6.2). This is especially useful as changes to the code automatically propagate to all versions generated from it, which enables the maintenance of multitudinous versions over an extended period of time. Another application of this technique is in embedded systems where the code size has influence on the system performance and where run-time configuration is not an option.

```
(defmacro find-max (graph trav)
  `(decl ((int max (val (root ,graph))))
     (,trav ,graph
        (if (> (val curr) max)
            (set max (val curr))))))

(defmacro push-stack (v)
  `(if ,v (set stack[++sp] ,v)))

(defmacro preorder-traversal (graph &body code)
  `(decl ((node* stack[N])
          (int sp 0))
     (set stack[0] (root ,graph))
     (while (>= sp 0)
       (decl ((node* curr stack[sp--]))
         ,@code
         (push-stack (left curr))
         (push-stack (right curr))))))

(defmacro breath-first-traversal (graph &body code)
  `(decl ((queue* q (make-queue)))
     (enqueue q ,graph)
     (while (not (empty q))
       (decl ((node* curr (dequeue q)))
         ,@code
         (if (left curr)
             (enqueue q (left curr)))
         (if (right curr)
             (enqueue q (right curr)))))))

(function foo ((graph *g)) -> int
  (find-max g
    preorder-traversal))
```

Figure 3: This example illustrates the configuration of an operation (`find-max`) with two different graph traversal algorithms. Note that this setup does not incur run-time overhead.

5.3 Domain-Specific Languages

To illustrate the definition and use of embedded domain-specific languages we present a syntax to embed elegant and concise regular expression handling in CGen code. Figure 4 provides a very simple implementation with the following syntax.

```
(match text
  ("([^.]*)" (printf "proper list.\n"))
  (".*\." (printf "improper list.\n")))
```

The generated code can be seen in Figure 5. Note how the output code is structured to only compute the regular expression representations that are required.

```
(defmacro match (expression &rest clauses)
  `(macrolet
     ((match-int (expression &rest clauses)
        `(progn
           (set reg_err (regcomp &reg
                                 ,(caar clauses)
                                 REG_EXTENDED))
           (if (regexec &reg ,expression 0 0 0)
               (progn ,@(cdar clauses))
               ,(lisp (if (cdr clauses)
                          `(match-int
                             ,expression
                             ,@(cdr clauses))))))))
     (decl ((regex_t reg)
            (int reg_err))
       (match-int ,expression ,@clauses))))
```

Figure 4: An example of an embedded domain-specific language for providing an elegant syntax for checking a string against a set of regular expressions.

```
regex_t reg;
int reg_err;
reg_err = regcomp(&reg, "([^.]*)", REG_EXTENDED);
if (regexec(&reg, text, 0, 0, 0))
    printf("proper list.\n");
else {
    reg_err = regcomp(&reg, ".*\.", REG_EXTENDED);
    if (regexec(&reg, text, 0, 0, 0))
        printf("improper list.\n");
}
```

Figure 5: Code resulting from application of the syntax defined in Figure 4.

Clearly, more elaborate variants are possible. Consider, e.g., the popular CL-PPCRE [29] library which analyzes the individual regular expressions and, if static, precomputes the representation. This is not directly applicable to the C regular expression library used here but can be understood as selectively removing the call to `regcomp`.

5.4 Layered Abstractions

One of the canonical examples of aspect-oriented programming is the integration of logging into a system. Without language support it is tedious work to integrate consistent logging into all functions that require it.

Figure 6 presents a macro that automatically logs function calls and the names and values of the parameters, simply by defining the function with a different form:

```
(function  foo (...) ...)                ; does not log
(function+ foo (...) ...)                        ; logs
```

With this definition in place the following form

```
(function+ foo ((int n) (float delta)) -> void
  (return (bar n delta)))
```

evaluates to the requested function:

```
(function foo ((int n) (float delta)) -> void
  (progn
    (printf
      "called foo(n = %d, delta = %f)\n" n delta)
    (return (bar n delta))))
```

With this technique it is easily possible to redefine and combine different language features while honoring the separation of concerns principle. The most simple implementation facilitating this kind of combination would be defining a macro that applies all requested extensions to a given primitive. This could be managed by specifying a set of globally requested aspects which are then integrated into each function (overwriting the standard definition).

```
(defmacro function+ (name param arr ret &body body)
  `(function ,name ,param ,arr ,ret
     (progn
       (printf
         ,(format
            nil "called ~a(~{~a = ~a~^, ~})\n" name
            (loop for item in parameter append
              (list (format nil "~a"
                            (first (reverse item)))
                    (map-type-to-printf
                      (second (reverse item))))))
         ,@(loop for item in parameter collect
             (first (reverse item))))
       ,@body)))
```

Figure 6: Implementation of the logging aspect.

6. EVALUATION

> It is hard to overestimate the importance of concise notation for common operations.
>
> B. Stroustrup [27]

As already exemplified in Section 5.3, the quoted text is certainly true, and we agree that the language user, not the designer, knows what operations are to be considered 'common' the best.

In the following we will first present a natural notation for SIMD expressions which are very common in high-performance code. This is followed by an application of our system to a classical problem of high-performance computing which demonstrates how redundancy can be avoided with separation of concerns thereby being applied.

6.1 A Natural Notation for SIMD Arithmetic

SIMD (single instruction, multiple data) is a very common approach to data parallelism, applied in modern CPUs by the SSE [10], AVX [11] and Neon [2] instruction sets. These allow applying a single arithmetic or logic operation (e.g. an addition) to multiple (2, 4, 8, or 16) registers in a single cycle. Naturally, such instruction sets are very popular in high-performance applications where they enable the system to do more work in the same amount of time. The examples in this section will make use of so-called intrinsics, which are functions recognized by the compiler to map directly to assembly instructions.

As an example the following code loads two floating point values from consecutive memory locations into an SSE register and adds another register to it.

```
__m128d reg_a = _mm_load_pd(pointer);
reg_a = _mm_add_pd(reg_a, reg_b);
```

Obviously, more complicated expressions soon become unreadable and require disciplined documentation. Consider, e.g., the expression `(x+y+z)*.5` which would be written as:

```
_mm_mul_pd(
    _mm_add_pd(
        x,
        _mm_add_pd(y, z)),
    .5);
```

There are, of course, many approaches to solving this problem. We compare the light-weightedness and quality of abstraction in our method to a hand-crafted DSL implemented in C using the traditional compiler tools, as well as to an ad hoc code generator framework such as Mako [5]. We argue that the scope of this problem (with the exception of the extreme case of auto-vectorization [17]) does not justify the application of large scale-systems such as writing a source to source compiler using the Clang framework [28].

Traditional Solution.

Our first approach to supply a more readable and configurable notation of SIMD instructions employed traditional compiler technology. The `intrinsify` program reads a file and copies it to its output while transforming expressions that are marked for conversion to intrinsics (after generating an AST for the sub expression using [12] and [20]). The marker is a simple comment in the code, e.g. we transform the following code

```
__m128d accum, factor;
for (int i = 0; i < N; i++) {
    __m128d curr = _mm_load_pd(base + i);
    //#INT accum = accum + factor * curr;
}
```

to produce code that contains the appropriate intrinsics:

```
__m128d accum, factor;
for (int i = 0; i < N; i++) {
    __m128d curr = _mm_load_pd(base + i);
    //#INT accum = accum + factor * curr;
    accum = _mm_add_pd(
                accum,
                _mm_mul_pd(
                    factor,
                    curr
                )
            );
}
```

The instruction set (SSE or AVX) to generate code for can be selected at compile-time.

String-Based Approach.

Using Mako [5] we implemented an ad hoc code generator which runs the input data through Python. In this process the input file is simply copied to the output file and embedded Python code is evaluated on the fly. The previous example is now written as:

```
__m128d accum, factor;
for (int i = 0; i < N; i++) {
    __m128d curr = _mm_load_pd(base + i);
    ${with_sse(set_var "accum"
                    (add "accum"
                         (mul "factor" "curr")))};
}
```

Note how all the data handed to the Python function is entirely string based.

Using CGen.

With our system the extended notation is directly embedded in the source language as follows:

```
(decl ((__m128d accum)
       (__m128d factor))
  (for ((int i 0) (< i N) i++)
    (intrinsify
      (decl ((mm curr (load-val (aref base i))))
        (set accum (+ accum (* factor curr)))))))
```

Comparison.

The implementation of the `intrinsify` program is around 1,500 lines of C/Lex/Yacc code. Using those tools the calculator grammar is very manageable and can be extended in numerous ways to provide a number of different features. Our use case is to automatically convert numeric constants into SIMD format, i.e. converting `//#INT x = 0.5 * x;` to

```
_m128d c_0_500 = _mm_set1_pd(0.5);
x = _mm_mul_pd(c_0_500, x);
```

Since keeping track of names that have already been generated is straight-forward, this is a robust approach to further simplify the notation. Note that it is not easily possible to move such temporaries out of loops as this would require the construction of a rather complete AST which was clearly not the intention of writing such a tool. This example demonstrates that once the initial work is completed such a system can be easily extended and maintained.

The string-based version, on the other hand, is very lightweight and only takes up 60 lines of code. Even though this shows that such abstractions can be constructed on demand and the return on the work invested is obtained very quickly, the resulting syntax is not very far from writing the expressions themselves. The extension to extract numeric constants heavily relies on regular expressions and can only be considered maintainable as long as the code does not grow much larger. Further code inspection and moving generated expressions out of loops is not easily integrated.

The implementation of our `intrinsify` macro consists of 45 lines of code, which is comparable to the Python implementation. The notation, however, is very elegant and convenient and the extraction and replacement of constants are simple list operations. As an example, obtaining the list of numbers in an expression is concisely written as:

```
(remove-duplicates
  (remove-if-not #'numberp (flatten body))
```

6.2 A Configurable Jacobi Solver

In the field of high performance computing a large class of algorithms rely on stencil computations [3]. As a simple example we consider a 2-D Jacobi kernel for solving the heat equation. Hereby a point in the destination grid is updated with the mean value of its direct neighbors from the source grid. After all points have been updated in this way the grids are swapped and the iteration starts over.

Whereas for the chosen example, shown in Figure 7, state-of-the-art compilers can perform vectorization of the code, they fail at more complicated kernels as they appear, e.g. in computational fluid dynamics. This often leads to hand-crafted and hand-tuned variants of such kernels for several architectures and instruction sets, for example with the use of intrinsics. In all further examples we assume that the alignment of the source and destination grid differ by 8-bytes, i.e. the size of a double precision value.

```
#define I(x, y) (((y) * NX) + (x))
double dest[NX * NY], src[NX * NY], * tmp;

void Kernel(double *dst, double *top,
            double *center, double *bottom,
            int len) {
    for (int x = 0; x < len; ++x)
        dst[x] = 0.25 * (top[x] + center[x-1] +
                         center[x+1] + bottom[x]);
}

void Iterate() {
    while (iterate) {
        for (int y = 1; y < NY - 1; ++y)
            Kernel(&dest[I(1,y)], &src[I(1,y-1)],
                   &src[I(1,y)],  &src[I(1,y+1)],
                   NX-2);
        swap(src, dest);
    }
}
```

Figure 7: A simple 2-D Jacobi kernel without any optimizations applied.

Figure 8 shows how a hand-crafted version using intrinsics targetting SSE may look. In this example double precision floating point numbers are used, i.e. the intrinsics work on two values at a time. At the end of the function there is a 'peeled off', non-vectorized version of the stencil operation to support data sets of uneven width. Even for this very simple example the code already becomes rather complex.

```
void KernelSSE(double *d, double *top, double *center,
               double *bottom, int len) {
    const __m128d c_0_25 = _mm_set_pd(0.25);
    __m128d t, cl, cr, b;

    for (int x = 0; x < len - (len % 2); x += 2) {
        t  = _mm_loadu_pd(&top[x]);
        cl = _mm_loadu_pd(&center[x - 1]);
        cr = _mm_loadu_pd(&center[x + 1]);
        b  = _mm_loadu_pd(&bottom[x]);

        _mm_storeu_pd(&dst[x],
            _mm_mul_pd(
                _mm_add_pd(
                    _mm_add_pd(t, cl),
                    _mm_add_pd(cr, b)),
                c_0_25));
    }

    if (len % 2) {
        int x = len - 1;
        dst[x] = 0.25 * (top[x] + center[x - 1] +
                         center[x + 1] + bottom[x]);
    }
}
```

Figure 8: The same operation as shown in Figure 7 but targetting SSE.

A further optimized version could use the non-temporal store instruction (MOVNTPD) to by-pass the cache when writing to memory, which in turn would require a 16-byte alignment of the store address. This would necessitate a manual update of the first element in the front of the loop if its address is incorrectly aligned. Further, for AVX-variants of the kernel the loop increment becomes four since four elements are processed at once. The peeling of elements in front (for non-temporal stores) and after the loop (for left-over elements) would make further loops necessary.

In the following we show how a simple domain-specific approach can implement separation on concerns, i.e. separate the intrinsics optimizations from the actual stencil used. This frees the application programmer from a tedious reimplementation of these optimizations for different stencils and cumbersome maintenance of a number of different versions of each kernel.

We implemented a set of macros to generate the different combinations of aligned/unaligned and scalar/SSE/AVX kernels in 260 lines of code (not further compacted by meta macros). The invocation

```
(defkernel KernelScalar (:arch :scalar)
  (* 0.25 (+ (left) (right) (top) (bottom))))
```

produces the following kernel, virtually identical to Figure 7:

```
void KernelScalar(double *dst, double *top,
                  double *center, double *bottom,
                  int len) {
    for (int x = 0; x < len; x += 1)
        dst[x] = 0.25 * (center[x-1] + center[x+1]
                         + top[x] + bottom[x]);
    return;
}
```

The invocation of

```
(defkernel KernelSSE (:arch :sse)
  (* 0.25 (+ (left) (right) (top) (bottom))))
```

generates code very similar to Figure 8 (not shown again for a more compact representation), and the most elaborate version, an AVX kernel with alignment, can be constructed using

```
(defkernel KernelAlignedAVX (:arch :avx :align t)
  (* 0.25 (+ (left) (right) (top) (bottom))))
```

The resulting code is shown in Figure 9. Note how for each kernel generated exactly the same input routine was specified. The vectorization is implemented analogous to the method described in Section 6.1. In this version, however, we extracted the numeric constants of the complete function and moved them before the loop.

```
void KernelAlignedAVX(double *dst, double *top,
                      double *center, double *bottom,
                      int len) {
    int x = 0;
    const __m256d avx_c_0_25_2713
                    = _mm256_set1_pd(0.25);
    __m256d avx_tmp1590;
    __m256d avx_tmp1437;
    __m256d avx_tmp1131;
    __m256d avx_tmp1284;
    int v_start = 0;
    while (((ulong)dst) % 32 != 0) {
        dst[v_start] = 0.25 * (center[v_start-1]
                               + center[v_start+1]
                               + top[v_start]
                               + bottom[v_start]);
        ++v_start;
    }
    int v_len = len - v_start;
    v_len = (v_len - (v_len % 4)) + v_start;
    for (int x = v_start; x < v_len; x += 4) {
        avx_tmp1590 = _mm256_load_pd(center[x-1]);
        avx_tmp1437 = _mm256_load_pd(center[x+1]);
        avx_tmp1131 = _mm256_load_pd(top[x]);
        avx_tmp1284 = _mm256_load_pd(bottom[x]);
        _mm256_store_pd(
            &dst[x],
            _mm256_mul_pd(
                avx_c_0_25_2713,
                _mm256_add_pd(
                    _mm256_add_pd(
                        _mm256_add_pd(
                            avx_tmp1590,
                            avx_tmp1437),
                        avx_tmp1131),
                    avx_tmp1284)));
    }
    for (int x = v_len; x < len; ++x)
        dst[x] = 0.25 * (center[x-1] + center[x+1]
                         + top[x] + bottom[x]);
    return;
}
```

Figure 9: The same operation as shown in Figure 7 but targetting aligned AVX and generated by CGen.

7. CONCLUSION

In this paper we presented a code generator that enables Common Lisp-style meta programming for C-like platforms and presented numerous examples illustrating its broad applicability. We also showed how it can be applied to real-world high-performance computing applications. We showed how our approach is superior to simple string-based methods and to what extend it reaches towards large-scale systems requiring considerable initial overhead. Furthermore, we showed that our approach is well suited for lowering the entry barrier of using code generation for situations in which taking the large-scale approach can't be justified and simple string-based applications fail to meet the required demands.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive feedback and suggestions, Christian Heckl for insightful discussions and gratefully acknowledge the generous funding by the German Research Foundation (GRK 1773).

9. REFERENCES

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[2] ARM. Introducing NEON, 2009.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] N. Batchelder. Python Success Stories. http://www.python.org/about/success/cog/, 2014.

[5] M. Bayer. Mako Templates for Python. http://www.makotemplates.org/, 2014.

[6] B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

[7] J. Chroboczek. *The CL-Yacc Manual*, 2008.

[8] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, Nov 2008.

[10] Intel. *SSE4 Programming Reference*, 2007.

[11] Intel. *Intel Advanced Vector Extensions Programming Reference*, January 2014.

[12] S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[13] S. E. Keene. *Object-oriented programming in COMMON LISP - a programmer's guide to CLOS*. Addison-Wesley, 1989.

[14] B. W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[15] J. Kessenich, D. Baldwin, and R. Randi. *The OpenGL Shading Language*, January 2014.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[17] O. Krzikalla, K. Feldhoff, R. Müller-Pfefferkorn, and W. Nagel. Scout: A source-to-source transformator for SIMD-optimizations. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7156 of *Lecture Notes in Computer Science*, pages 137–145. Springer Berlin Heidelberg, 2012.

[18] M. Köster, R. Leißa, S. Hack, R. Membarth, and P. Slusallek. Platform-Specific Optimization and Mapping of Stencil Codes through Refinement. In *In Proceedings of the First International Workshop on High-Performance Stencil Computations (HiStencils)*, pages 1–6, Vienna, Austria.

[19] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[20] M. E. L. Lesk and E. Schmidt. Lex — a lexical analyzer generator. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[21] R. Membarth, A. Lokhmotov, and J. Teich. Generating GPU Code from a High-level Representation for Image Processing Kernels. In *Proceedings of the 5th Workshop on Highly Parallel Processing on a Chip (HPPC)*, pages 270–280, Bordeaux, France. Springer.

[22] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[23] M. Pharr and W. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13, May 2012.

[24] V. Sedach. Parenscript. http://common-lisp.net/project/parenscript/, 2014.

[25] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 1992. USENIX Association.

[26] O. Spinczyk and D. Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[27] B. Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, 4 edition, May 2013.

[28] The Clang Developers. Clang: A C Language Family Frontend for LLVM. http://clang.llvm.org, 2014.

[29] E. Weitz. CL-PPCRE - Portable Perl-compatible regular expressions for Common Lisp. http://www.weitz.de/cl-ppcre/, 2014.

[30] H. Zhang, S. Jarzabek, and S. M. Swe. Xvcl approach to separating concerns in product family assets. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, GCSE '01, pages 36–47, London, UK, 2001. Springer-Verlag.