Proceedings of the

12th European Lisp Symposium

Hotel Bristol Palace, Genova, Italy
April 1 – 2, 2019

In cooperation with ACM

Co-located with <Programming> 2019

Nicolas Neuss (ed.)

ISBN-13: 978-2-9557474-3-8
ISSN: 2677-3465

Preface

Message from the Program Chair

Welcome to the 12thth European Lisp Symposium!

When Didier Verna asked me to be program chair for the ELS 2019, I felt frightened and honored at the same time. Frightened, because I was not sure if I would have sufficient experience and could afford enough time for doing this job. And honored, because the ELS conferences are an important international meeting point for developers and users of my beloved programming language Lisp.

What is Lisp? Lisp is the language which inspired the world of programming more than any other language. It introduced a multitude of important features like interactivity and introspection, functional programming, and automatic garbage collection. Although most of these features have been imported into other languages, there is one important feature which can only be copied by becoming a 'Lisp', namely the uniform syntax that is both a notation for code and data! As a direct consequence, this property makes program transformation an easily accessible tool for us Lisp programmers.

In fact, several of the contributions in these proceedings rely heavily on this feature. For example, it is extremely convenient if you want to bootstrap a system, and some of our talks consider this problem. It is also very helpful in program analysis and program rewriting, which is the topic of further talks. And then it also opens up new roads towards high performance, because it allows specialized code to be generated and compiled fast at runtime which is the topic of another talk.

Because of this uniqueness of Lisp, I am convinced that Lisp will remain with us forever—although people may call it by another name in the future. So—coming back to the beginning—I fought down my initial fright and accepted the honor of being ELS program chair. I have not regretted this decision, because I have had a lot of support: (1) there was an almost perfect guide for managing ELS conferences which described everything necessary in detail, (2) the well-established conference platform EasyChair made steps like collecting the program committee, paper submission and paper review really convenient, and (3) for any problems which were beyond my knowledge, I obtained rapid help from Didier and other people from the ELS steering committee.

Therefore, being ELS program chair was a pleasant experience, and I want to thank all who made this happen: Didier, the steering committee, the program committee, the invited speakers, and also all you authors for submitting good papers. Last but not least, I want to mention the help from my friend and colleague Marco Heisig who—besides helping out in the program committee—was always a source of inspiration and knowledge.

Erlangen, March 24, 2019 Nicolas Neuss

Organization

Programme Chair

- Nicolas Neuss, FAU Erlangen-Nürnberg, Germany

Local Chair

- Davide Ancona, University of Genova, Italy
- Elena Zucca, University of Genova, Italy

Programme Committee

- Alberto Riva – University of Florida, USA
- Alessio Stalla – ManyDesigns Srl, Italy
- Breanndán Ó Nualláin – University of Amsterdam, Netherlands
- Charlotte Herzeel – IMEC, ExaScience Life Lab, Leuven, Belgium
- François-René Rideau – Alacris.io
- Leonie Dreschler-Fischer – University of Hamburg, Germany
- Marc Battyani – FractalConcept, France
- Marco Antoniotti – Universita Milano Bicocca, Italy
- Marco Heisig – FAU Erlangen-Nürnberg, Germany
- Pascal Costanza – IMEC, ExaScience Life Lab, Leuven, Belgium
- Patrick Krusenotto – Deutsche Welle, Germany
- Philipp Marek – Austria
- Pierre R. Mai – PMSF IT Consulting, Germany
- R. Matthew Emerson – thoughtstuff LLC, USA
- Sacha Chua – Living an Awesome Life, Canada

Sponsors

We gratefully acknowledge the support given to the 12thth European Lisp Symposium by the following sponsors:

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

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

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

Invited Contributions

The Lisp of the Prophet for the One True Editor

Stefan Monnier, Université de Montréal, Québec, Canada

While the editor war is long gone and Emacs's marketshare has undoubtedly shrunk, it has established itself as an important branch in the Lisp family of languages. In this talk, I will look at what gave Emacs Lisp its shape, including what it took from its siblings and ancestors and what makes it different.

Stefan Monnier is a professor in the Département d'Informatique et Recherche Opérationnelle at the Université de Montréal where he works on functional programming languages and type systems. He spends the other 90% of his time contributing to Emacs, of which he was the official maintainer from 2008 to 2015. He received his PhD from Yale University in 2003.

20 More Years of Bootstrapping

Christophe Rhodes, England

Although its history is longer than this, the Steel Bank Common Lisp system was announced to the world in 1999, with its distinguishing characteristic of being written in vanilla ANSI Common Lisp explicitly described in the announcement. We provide a retrospective on 20 years of development, discuss some of the features SBCL provides and the rationale behind them, and offer entirely speculative thoughts about the next 20 years of the project.

Christophe Rhodes received a PhD in maths and physics from Cambridge in 2004, following which he combined other interests by researching into computational understanding of music. A period working on a startup, Teclo Networks, was followed by a period in academic management, being part of the team leading the Transforming Musicology project and acting as Head of Department. He currently works as a software engineer.

Guest Talk

Rebooting Racket

Matthew Flatt, University of Utah, USA

Racket started in 1995 as a fusion of two off-the-shelf C/C++ libraries. From there, things got out of hand. We ended up building and maintaining a large programming language—with ambitions that span from language design to performance, from research to production—on an especially inelegant core implementation. After 2+ years of concerted effort, we have rebuilt Racket's core in a more maintainable form. It's a story as old as programming, and this particular instance looks like it will have a happy ending.

Matthew Flatt is a professor in the School of Computing at the University of Utah, where he works on extensible programming languages, runtime systems, and applications of functional programming. He is one of the developers of the Racket programming language. He received his PhD from Rice University in 1999.

Program overview

Monday, 1.4.2019

09:15	**Welcome**
09:30–10:30	Stefan Monnier: The Lisp of the prophet for the One True Editor (ELS keynote)
10:30–11:00	**Coffee break**
11:00–11:30	Ryan Culpepper: Pattern-Based S-Expression Rewriting in Emacs
11:30–12:00	Léo Valais, Jim Newton and Didier Verna: Implementing Baker's SUBTYPEP decision procedure
12:00–12:30	Irène Anne Durand and Robert Strandh: make-method-lambda revisited
12:30–14:30	**Lunch**
14:30–15:00	Jim Newton and Didier Verna: Finite Automata Theory Based Optimization
15:00–15:30	Marco Heisig: Lazy, parallel multiple value reductions in Common Lisp
15:30.-16:00	**Coffee break**
16:00–16:30	Mikhail Raskin and Christoph Welzel: Working with first-order proofs and provers
16:30–17:00	António Leitão: Plagiarism Detection for Lisp
17:00–17:30	Lightning talks
19:00	**Reception at Aula Magna**

Tuesday, 2.4.2019

09:00–10:00	Christophe Rhodes: 20 more years of bootstrapping (ELS keynote)
10:00–10:30	Irène Anne Durand and Robert Strandh: Bootstrapping Common Lisp using Common Lisp
10:30–11:00	**Coffee Break**
11:00–11:45	Nicolas Hafner. Shader Pipeline and Effect Encapsulation using CLOS
11:45–12:30	Robert P. Goldman and Ugur Kuter. Hierarchical Task Network Planning in Common Lisp
12:30–14:30	**Lunch**
14:30–15:30	Matthew Flatt: Rebooting Racket (Guest talk)
15:30–16:00	**Coffee Break**
16:00–16:30	Alessio Stalla: Symbols as Namespaces in Common Lisp
16:30–17:00	Didier Verna: Parallelizing Quickref
17:00–17:30	Lightning talks
17:30	**Conference end**

Session I: Emacs Lisp

Monday, 1.4.2019

09:30–10:30	Stefan Monnier: The Lisp of the prophet for the One True Editor (ELS keynote)
10:30–11:00	**Coffee break**
11:00–11:30	Ryan Culpepper: Pattern-Based S-Expression Rewriting in Emacs

Pattern-Based S-Expression Rewriting in Emacs

Ryan Culpepper
Czech Technical University in Prague
ryanc@racket-lang.org

ABSTRACT

sexp-rewrite is an Emacs library for doing pattern-based rewriting of S-expression-structured code—ie, code in Lisp, Scheme, and Racket. The library provides a simple but powerful pattern language that enables users to define rewriting rules (called "tactics") and auxiliary nonterminals.

CCS CONCEPTS

• **Software and its engineering** → *Domain specific languages*; *Development frameworks and environments*;

ACM Reference Format:
Ryan Culpepper. 2019. Pattern-Based S-Expression Rewriting in Emacs. In *Proceedings of the 12th European Lisp Symposium (ELS'19)*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.5281/zenodo.2642093

1 INTRODUCTION

sexp-rewrite is an Emacs library that allows users to apply rewriting rules for S-expression structured program text. It also allows users to define their own rewriting rules using a simple but expressive pattern language based on the syntax-parse notation for macros (Culpepper 2012).

This section introduces sexp-rewrite by showing examples of rewriting tactics included with sexp-rewrite for the Racket programming language.

One example is the conversion of nested trees of if expressions to cond expressions. Here is one transformation:

```
(define-sexprw-tactic if-to-cond
  (if $test $then $else)
  (cond [$test $then] !NL
        [else $else]))
```

Another tactic rewrites let followed by an immediate if test:

```
(define-sexprw-tactic let-if-to-cond
  (let ([$name:id $rhs])
    (if $name:id $then $else))
  (cond [$rhs !SL => (lambda ($name) !SL $then)] !NL
        [else !SL $else]))
```

These tactics, along with a few others not shown here, make it possible to transform a tree of ifs and lets into a cond expression with a single command (Figure 1).

The Quack (Van Dyke 2002) major mode for Scheme and Racket has a function for toggling the function definition at the cursor between implicit and explicit lambda notation. For example:

```
(define (add1 n)        ⇔   (define add1
  (+ 1 n))                    (lambda (n) (+ 1 n)))
```

The main elisp function implementing this transformation, quack-toggle-lambda, is 74 lines, not including helper functions and regular expressions defined elsewhere in quack.el.

Using sexp-rewrite, the editor transformations can be expressed with two rewriting "tactics" of three lines each:[1]

```
(define-sexprw-tactic define-absorb-lambda
  (define $name:id (lambda ($arg ...) $body:rest))
  (define ($name $arg ...) !NL $body))
(define-sexprw-tactic define-split-lambda
  (define ($name:id $arg ...) $body:rest)
  (define $name !NL (lambda ($arg ...) !SL $body)))
```

Another example comes from SXML (Kiselyov 2004), which uses a particular idiom for optional arguments: the formals include a rest argument, then the procedure body starts with a let expression that binds the optional argument to the first element of the rest argument, if there is one, or a default value otherwise. In Racket it is more idomatic (and efficient) to use the language's built-in support for optional arguments. For example:

```
(define (ddo:ancestor test-pred? . nums)
  (let ((num (if (null? nums) 0 (car nums))))
    (do-stuff-with test-pred? num)))
⇒
(define (ddo:ancestor test-pred? [num 0])
  (do-stuff-with test-pred? num))
```

A tactic for translating the former to the latter is shown in Figure 2. The tactic's guard handles a case where a default value of null is written as the rest argument (known to be null on that branch).

2 TACTICS AND NONTERMINALS

A tactic definition consists of a pattern, template, and optional :with or :guard clauses:

```
(define-sexprw-tactic Name
  Pattern WithOrGuard ... Template)

WithOrGuard = :with Pattern Template
            | :guard GuardFunction
```

A :with clause performs an additional pattern match on the text produced by an intermediate template. A guard procedure takes an environment (an association list mapping attribute names to matches) and returns either a singleton list with an environment (possibly extended or modified) to accept or nil to reject.

ELS'19, April 01–02 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2642093

[1] With this ~1500 line supporting library.

```
(if (not k)
    (error)
    (let ([x (assq k env)])
      (if x (cdr x) (error))))
```

⇔

```
(cond [(not k) (error)]
      [(assq k env) => (lambda (x) (cdr x))]
      [else (error)])
```

Fig. 1. Conversion of if and let to cond

```
(define-sexprw-tactic define-rest-to-optional
  (define ($name:id $arg:id ... . $rest:id)
    (let (($optional-arg:id (if (null? $rest:id) $default (car $rest:id))))
      $body:rest))
  :guard (lambda (env)
           ; If $default = $rest, rewrite to null; unsafe if refs to $rest remain.
           (if (sexprw-entry-equal (sexprw-env-ref env '$default) (sexprw-env-ref env '$rest))
               (list (cons (cons '$default (sexprw-template 'null env)) env))
               (list env)))
  (define ($name $arg ... [$optional-arg $default]) !NL $body))
```

Fig. 2. A tactic for optional arguments

A nonterminal definition consists of one or more patterns:

```
(define-sexprw-nt Name
  MaybeAttrs
  (pattern Pattern WithOrGuard ...) ...)

MaybeAttrs = ε
           | :attributes (ATTR-NAME ...)
```

Pattern variables defined inside of a nonterminals patterns are available as attributes of instances of the nonterminal.

A tactic name can also be used as a nonterminal name. In addition to the tactic's pattern variables, it also exports an attribute named $out with the result of the template. This makes it simple to compose tactics.

The following built-in nonterminals are provided:

- id matches any atom (currently includes numbers, etc, too).
- pure-sexp matches a single sexp.
- sexp matches a single sexp, which may have comments preceding it.
- rest matches the rest of the enclosing sexp, including comments and terms; useful for function bodies, for example.

3 PATTERNS AND TEMPLATES

Rewriting tactics use *patterns* to match regions of text to which the rewriting applies. Patterns bind *pattern variables* to subregions of text and *templates* use pattern variables, literal text, and formatting instructions to form the replacement text.

A pattern is one of the following:

- symbol matches that literal symbol. The symbol must not start with a $ character.
- $name matches any S-expression and binds it to the pattern variable $name.
- $name:nt matches an occurrence of the nonterminal nt and binds it to the pattern variable $name.
- (pattern1 ··· patternN) (that is, a list of patterns—the ··· are not literal) matches a parenthesized sequence of *N* terms where each term matches the corresponding pattern.
- (!SPLICE pattern1 ··· patternN) matches a sequence of *N* terms (non-parenthesized) where each term matches the corresponding pattern.
- pattern ... (that is, a pattern followed by literal ellipses) matches zero or more occurrences of pattern. The variables within pattern are bound to a sequence of matches, and they must be used under ellipses in the corresponding template.
- (!OR pattern1 ··· patternN) matches if any of the given patterns match.
- (!AND pattern1 ··· patternN) matches if all of the given patterns match.

The same forms are allowed for templates, except !OR and !AND are not allowed, and pattern variables are written as $name instead of $name:nt. The following additional template forms are supported:

- $name.$attr produces the text bound to the attribute $attr of the pattern variable $name, which must be bound to a nonterminal the defines $attr.
- (!SQ template1 ··· templateN) encloses the results of the *N* templates within square brackets.
- !NL prefers a new line before the next non-empty template.
- !SL prefers a new line before the next template only if its contents span multiple lines.

The square-bracket and spacing instructions are helpful for producing idiomatically formatted output.

4 AVAILABILITY

sexp-rewrite is available at
https://github.com/rmculpepper/sexp-rewrite/

BIBLIOGRAPHY

Ryan Culpepper. Fortifying macros. *Journal of Functional Programming* 4-5(22), pp. 224–243, 2012.

Oleg Kiselyov. SXML. http://okmij.org/ftp/Scheme/SXML.html, 2004.

Neil Van Dyke. Quack. https://www.neilvandyke.org/quack/, 2002.

Session II: Implementation

Monday, 1.4.2019

11:30–12:00	Léo Valais, Jim Newton and Didier Verna: Implementing Baker's SUBTYPEP decision procedure
12:00–12:30	Irène Anne Durand and Robert Strandh: make-method-lambda revisited
12:30–14:30	**Lunch**

Implementing Baker's SUBTYPEP decision procedure

Léo Valais
Jim E. Newton
Didier Verna
lvalais@lrde.epita.fr
jnewton@lrde.epita.fr
didier@lrde.epita.fr
EPITA/LRDE
Le Kremlin-Bicêtre, France

ABSTRACT

We present here our partial implementation of Baker's decision procedure for `subtypep`. In his article "A Decision Procedure for Common Lisp's `SUBTYPEP` Predicate", he claims to provide implementation guidelines to obtain a `subtypep` more accurate and as efficient as the average implementation. However, he did not provide any serious implementation and his description is sometimes obscure. In this paper we present our implementation of part of his procedure, only supporting primitive types, CLOS classes, `member`, range and logical type specifiers. We explain in our words our understanding of his procedure, with much more detail and examples than in Baker's article. We therefore clarify many parts of his description and fill in some of its gaps or omissions. We also argue in favor and against some of his choices and present our alternative solutions. We further provide some proofs that might be missing in his article and some early efficiency results. We have not released any code yet but we plan to open source it as soon as it is presentable.

CCS CONCEPTS

• **Theory of computation** → **Type theory**; *Divide and conquer*; Pattern matching.

ACM Reference Format:
Léo Valais, Jim E. Newton, and Didier Verna. 2019. Implementing Baker's SUBTYPEP decision procedure. In *Proceedings of the 12th European Lisp Symposium (ELS'19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.2646982

1 INTRODUCTION

The Common Lisp standard [1] provides the predicate function `subtypep` for introspecting the sub-typing relationship. Every invocation `(subtypep A B)` either returns the values `(t t)` when `A` is a subtype of `B`, `(nil t)` when not, or `(nil nil)` meaning the predicate could not (or failed to) answer the question. The latter can happen when the type specifier `(satisfies P)` (representing the type $\{x \mid \mathsf{P}(\mathsf{x})\}$ for some predicate and total function[1] `P`) is involved. For example, given two arbitrary predicates `F` and `G`, there is no way `subtypep` can answer the question `(subtypep '(satisfies F) '(satisfies G))`.

```
(sb!xc:deftype keyword ()
  '(and symbol (satisfies keywordp)))
```

Listing 1: The `keyword` type definition in SBCL

However, some implementations abuse the permission to return `(nil nil)`. For example, in SBCL 1.4.10 (the implementation we are currently focusing our efforts on), `(subtypep 'boolean 'keyword)` returns `(nil nil)`, thus violating the standard[2]. The definition of the `keyword` type is responsible for this failure: as shown in Listing 1, it involves a `satisfies` type specifier[3].

Another kind of problem for which `subtypep`'s accuracy matters is the optimization of the `typecase` construct as shown in [7] and [8]. The aim is to remove redundant checks in the construct and the approach is to use binary decision diagrams. However, to build such a structure, `subtypep` is repeatedly used. The unreliability of the predicate leads here to many lost BDD reductions and therefore to the generation of sub-optimal code.

Our implementation is still in active development, currently targets SBCL and focuses almost entirely on result accuracy. It supports *primitive* types, *user-defined* types (`deftype`, classes and structures), `member` *(and* `eql`*)* type specifiers and *ranges* (e.g., `(integer * 12)`). We present our strategy for implementing each one of these while discussing how and why we decided or not to diverge from Baker's [3] approach—or potentially filling some gaps or unclear bits. No optimization work has been done yet and the implementation still has bugs and diverse issues, but we have found some encouraging results about accuracy and even about efficiency.

2 THE COMMON LISP TYPE SYSTEM

2.1 Type specifiers

Common Lisp types are not manipulated directly. Instead, the type to be manipulated is *described* using a *type specifier*. The type specifier Domain-Specific Language (DSL) allows programmers to describe types by writing S-expressions which obey some rules described in the Common Lisp standard [1].

ELS'19, April 01–02 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2646982

[1] A function defined over its entire definition domain.
[2] The Common Lisp standard requires that no invocation of `subtypep` involving only primitive types return `(nil nil)`.
[3] C.f. bug #1533685 in SBCL bug tracker.

```
(deftype except (x)
  `(not (eql ,x)))
```

Listing 2: The `deftype` construct

A subtlety about type specifiers is that *different ones* can represent the *same* type (e.g., `integer`, `(integer * *)` and `(or fixnum bignum)` all describe the same type). This means that *symbolic computation does not suffice* to answer the sub-typing question. Note that one could write a predicate, say `type=`, to determine whether two type specifiers in fact describe the same type using two calls of `subtypep`.

It is possible to define *parametric aliases* using the `deftype` construct. It is then possible to refer to a whole type specifier using its alias. Listing 2 shows an example of parametric `deftype`.

2.2 Vocabulary

type	A set of elements. For any type u: $u \equiv \{x \mid x : u\}$
canonical t.s.	A type specifier without aliases.
primitive type	A standardized type ([2]) that is not necessarily implemented as a class.
symbolic form	A type specifier whose type is `symbol`.
compound form	A type specifier whose type is `list`.
logical form	A compound form whose car is `or`, `and` or `not`.
kingdom	In Baker's terminology, a "type kingdom" designates the types that can be described using *only one kind of type specifier*. `nil` (the empty type) belongs to every type kingdom.

In this article we focus on two particular type kingdoms:

- the *literal type kingdom*, represented using only symbolic, `member` and logical type specifiers, and,
- the *range type kingdom*, represented only using range and logical type specifiers

For example, `(or string symbol)` belongs to the literal type kingdom. `(and number (not real))` belongs to the range type kingdom. However, `(or symbol integer)` belongs to the literal type kingdom while `(or symbol (integer * *))` belongs to both. This situation is handled in section 4.

There are other type kingdoms that Baker mentions in his article, such as the *array* type kingdom, represented using only `array` and logical type specifiers. Note that a type can belong to several kingdoms, as multiple type specifiers can describe it. For example, `integer` belongs to literal *and* range kingdoms as the type specifiers `integer` (symbolic) and `(integer * *)` (range) both describe it. In Section 4, we describe how to guarantee that a given type is only described by one kind of type specifier, hence restricting it to *one* kingdom.

3 PROCEDURE'S MECHANISMS OVERVIEW

Figure 1 shows the internals of our implementation. Every step will be detailed in the following sections. There are three *major stages*:

(1) *The pre-processing* — Both type specifiers are processed in order to simplify further calculations: the aliases are expanded, and each occurrence of numeric types are converted to their equivalent range type specifier. Finally, as explained thereafter, the procedure splits into several sub-procedures, one for each type kingdom, because their internal type representation differ. In order to achieve that, the type specifiers must also be split into equivalent subtype specifiers restricted to each concerned kingdom. This stage is detailed in Section 4.

(2) *Expert sub-procedures* — Once split, each subtype specifier is redirected to the appropriate expert sub-procedure. The job of such a procedure is to *prove*, in its own kingdom, the assertion "*A* is a subtype of *B*" to be *wrong*. Our procedures currently only support literal and range type specifiers—an expert sub-procedure has been implemented only for these two kingdoms. This stage is detailed in Section 5.

(3) *Result conjunction* — Eventually, all expert sub-procedures return (a Boolean) and the results are accumulated using conjunction. (In practice, as soon as one expert procedure returns false, `subtypep` returns.)

4 PRE-PROCESSING

4.1 Alias expansion

The very first step is to ensure that the type specifier is in its *canonical form*, that is, having all its aliases expanded. This is done by the `expand` function. For example, considering the type created in Listing 2, `(expand '(except 12))` should return `(not (eql 12))`.

Unlike macro expansion, `deftype` expansion is not standardized in Common Lisp. Thus a solution must be found for each Common Lisp implementation independently. As our efforts are currently focused on SBCL, we discuss how we implement the `expand` function for that compiler.

SBCL's `subtypep` heavily relies on the function `sb-kernel:specifier-type`, which does type expansion. It also does type simplification—turning `(and integer string)` into `nil`—which could have saved us some work. We hoped we could simplify that function to make it compatible with Baker's algorithm while keeping the `deftype` expansion and the range canonicalization work. However we found, thanks to [7] tools, that the function is responsible for most of the work of `subtypep`, as shown in Figure 2 Considering the lack of efficiency of that function and the fact that it would not be trivial to simplify it to only keep the interesting bits, we decided on another, more cost-effective solution.

The function `sb-ext:typexpand` takes a type specifier and tries to expand it (not recursively). It either returns the expansion result, or the input type specifier if it is not expandable. `(sb-ext:typexpand 'integer)` returns `integer` since it is not a `deftype` alias whereas `(sb-ext:typexpand '(except 12))` returns `(not (eql 12))`. To expand a whole type specifier, it just needs to walk through it, applying `sb-ext:typexpand` on each list or atom manually. One subtlety though is that the result of an expansion may itself be an alias to expand[4]. For example, let's say that we have `(deftype my-type () '(except 0.0))`, then the result of `(sb-ext:typexpand 'my-type)` is `(except 0.0)`, which is of course an alias to expand again.

[4] Fortunately, `sb-ext:typexpand` also returns a Boolean indicating whether or not an expansion happened.

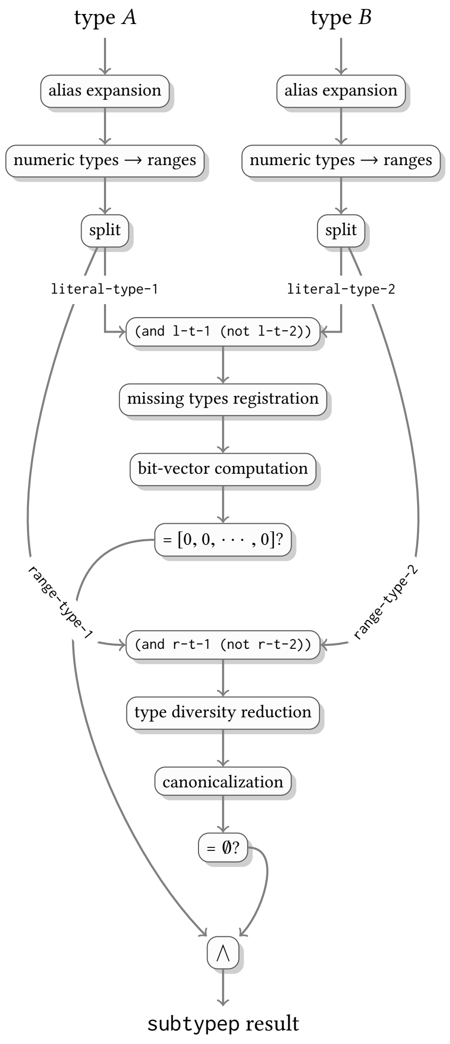


Figure 1: Internal flowchart of `(subtypep` A B`)`

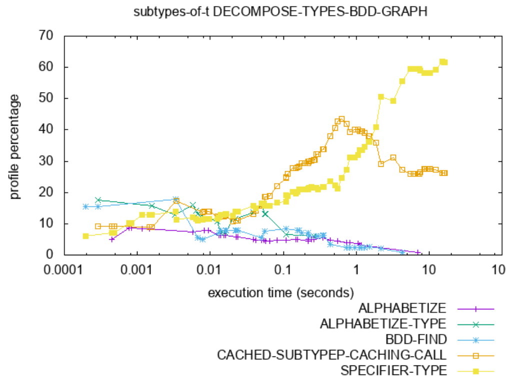


Figure 2: `specifier-type` weight in `cl:subtypep` execution[a]

[a] `cached-subtypep-caching-call` is just a memoizing wrapper around Sbcl's `subtypep` which is a bit more efficient than the raw implementation.

4.2 Numeric type specifiers conversion

As explained in Section 3, after pre-processing both type specifiers, the procedure splits in two expert sub-procedures: one for literal type specifiers and one for range type specifiers. Numeric types—types containing numbers (mathematically speaking)—can have different representations: a *symbol* (e.g., `fixnum`), a `member` *expression* (e.g., `(member 1 2 3)`) or a *range* (e.g., `(integer 1 6)`). However, the first two belong to the literal type kingdom whereas the latter belongs to the range kingdom. Thus, the numerical type information would be distributed over the different expert sub-procedures. For consistency and accuracy, a single internal representation has to be chosen. The symbolic and `member` numeric types must each be converted into an equivalent type specifier, in which numerical data are only represented using ranges.

- *Symbolic* numeric type specifier — say `U`, replace it by `(U * *)`[5]. Note the new "type specifier" is likely *not* to be valid (e.g., `(fixnum * *)` is invalid). Because it is never exposed to the user—as it is an intermediate, internal representation—nothing bad can happen. However, it cannot be used with other functions requiring a type specifier, such as `typep`.
- `member` type specifiers — e.g., `(member a 1 2 :b)` is converted to `(or (member a :b) (bit 1 1) (integer 2 2))`. To do that,
  1. extract the numbers out of the expression,
  2. map each number, say n, to construct the type specifier `((type-of` n`)` n n`)`[6],
  3. and combine the remaining `member` expression and the ranges with the `or` logical type specifier.

A subtlety to consider is that *super-types* of `number` also contain numerical data that must be extracted. Indeed, the type `atom` contains both numerical data—`(number * *)`—and non-numerical data—`(and atom (not (number * *)))`. Thus, its replacement in the numeric type kingdom is straightforward: `(number * *)`. In the literal type kingdom however, its replacement is `(or stream array character function standard-object symbol structure-object structure-class)`. The type `t`—which is `(or atom sequence)`—must be converted similarly.

Yet another subtlety is that the type specifiers `(and)` and `(or)` respectively describe the types `t` and `nil`. Hence every occurrence of `(and)` must be replaced by the replacement of `t` described in the previous paragraph. In order to remove that annoying corner case completely, `(or)` is also replaced, by `nil`.

4.3 Splitting

Having reached this step, the input now only contains *canonical literal* and *range* type specifiers, numeric types being *only* expressed as ranges. The next stage—expert sub-procedures—requires literal and numeric types to be separated.

Thus the top type `t` is divided into two[7] disjoint subtypes—"kingdoms" as Baker says. The previous step, described in Section 4.2, ensures that the representation (in terms of type specifiers) of the types in each kingdom is different. All numeric types are

[5] Implementations supporting the IEEE floating point raise many concerns with `-0.0`, NaN, +∞ and −∞. Baker explains in detail how to handle these cases.

[6] The results of `type-of` are implementation-dependent. We suppose here that `type-of` only returns the name (as a symbol) of the type of n (n being a `number`).

[7] One per kingdom actually, but since our implementation only supports two—literal and range types—we only focus our attention on these.

represented as ranges, and literal types as symbolic and `member` (without numbers) type specifiers.

This step roughly consists of an in-depth traversal of the type specifier, using pattern-matching to recognize which type specifier represents which type. We use the implementation of [9] because of its simplicity and versatility.

Our implementation uses a function `type-keep-if` which takes a predicate P and a type specifier U and returns:

- U as it is when $P(U) = \top$,
- `nil` when $P(U) = \bot$,
- `(`$op\ U_1\ \cdots\ U_n$`)` where U_i = `(type-keep-if` $P\ U_i$`)` when U = `(`$op\ U_1\ \cdots\ U_n$`)` and $op \in \{\texttt{and}, \texttt{or}, \texttt{not}\}$.

Given the predicate `literal-type-p` and a type U, `type-keep-if` returns U with every inner type specifier that describes a non-literal type replaced by `nil` (interpreted as the *empty type*). The result is then a subtype of `(and (not number) (not (array * *)))`. Likewise, given the predicate `range-type-p`, this function returns U with every non-range inner type specifier replaced by `nil` (interpreted this time as the *empty range*). Thus, the result is a subtype of `number`. Therefore, `split` can easily be implemented in terms of `type-keep-if`.

4.4 Type reformulation

For any types U and V, $U \subseteq V \Leftrightarrow U \cap \overline{V} = \emptyset$. Therefore, for any type specifiers `U` and `V`, when `(subtypep U V)` returns `T T`, then ``(subtypep `(and ,U (not ,V)) nil)`` also returns `T T`.

The results of the `split` function are zipped together using ``(lambda (x y) `(and ,x (not ,y)))`` before being passed to the expert sub-procedures. This way, they will not have to prove that an arbitrary type is a subtype of *another* arbitrary subtype, but rather whether *one* arbitrary type specifier describes the empty type (which is substantially easier to reason about, and implement).

5 EXPERT SUB-PROCEDURES

Listing 3 shows how `subtypep` could be defined from a top-down point of view. It shows that, according to Figure 1, both type specifiers are processed independently, split into two kingdoms (literal and numeric types) and unified in an `(and U (not V))` fashion. The expert sub-procedures, `null-literal-type-p` and `null-numeric-type-p`, each accept one argument—a type specifier, say U—and returns a Boolean indicating whether U describes the empty type (`nil`).

Each sub-procedure answers restricted to its kingdom—as no type can (at this point of the procedure) belong to two different kingdoms, as shown in section 4. With that piece of information, we can (now) safely assert that:

- the literal type kingdom is the type described by `(and (not number) (not (array * *)))`[8], and,
- the numeric type kingdom is the type described by `number`[9].

```
(defun subtypep (a b)
  (reduce (lambda (x y) (and x y))
          (mapcar (lambda (expert t1 t2)
                    (funcall expert `(and ,t1 (not ,t2))))
                  (list #'null-literal-type-p
                        #'null-numeric-type-p)
                  (split (num-types->ranges (expand a)))
                  (split (num-types->ranges (expand b))))))
```

Listing 3: A top-down approach of `subtypep`

There are several properties that are derived from the preceding pre-processing steps. First of all, both kingdoms' procedures are guaranteed to only ever receive argument *canonical* type specifiers. These are also guaranteed to never contain `atom` or `t` type specifiers. The occurrences of `(and)` and `(or)` have been replaced respectively by `t` and `nil`. `eql` type specifiers have been replaced by equivalent `member` expressions. `member` type specifiers only occur in the literal type kingdom and contain no numerical data. Numerical data are only expressed as intervals, which are likely not to be valid type specifiers. Both kingdoms accept the type specifier `nil` but with a *different meaning*: for literal types, `nil` means the empty type which complement is `t` whereas for numeric types it represent the empty range whose complement is `(number * *)`.

In the following sections we describe in detail the implementation of the expert sub-procedures for the literal (Section 5.1) and numeric (Section 5.2) type kingdoms. We also briefly discuss in Section 5.3 the array type kingdom and the `cons` type specifier family, which Baker ignores in his article.

5.1 Procedure for literal types

5.1.1 Theory. To represent types in the literal types kingdom, we suppose at first that there is a way to enumerate every element in `t`, say $e_1, e_2, \ldots, e_\omega$. Then, let $u_1, u_2, \ldots, u_\omega$ be all the (non-strict) subtypes of the top-level type `t`. We associate to each pair (u_i, e_j) the bit b_{ij} with the value 1 when $e_j \in u_i$ and 0 when $e_j \notin u_i$. Let bv_i be the *representative bit-vector* associated to the type u_i, defined by $[b_{i0}, b_{i1}, \ldots, b_{i\omega}]$. These bit-vectors are the rows of the infinite matrix on Eq. $B_{\omega\omega}$ which illustrates the system.

$$\begin{array}{c c} & \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & \cdots & e_\omega \end{array} \\ \begin{array}{c} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_\omega \end{array} & \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 1 \\ 0 & 1 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 1 & 0 & \cdots & 0 \end{pmatrix} \end{array} \tag{$B_{\omega\omega}$}$$

Proof. *Each type has a unique bit-vector representation.*

Let u_i and u_j be two distinct types. Thus, $(u_i \cup u_j) \backslash (u_i \cap u_j) \neq \emptyset$. Let $e_k \in (u_i \cup u_j) \backslash (u_i \cap u_j)$. By definition, we have $b_{ik} \neq b_{jk}$. Hence $bv_i \neq bv_j$. Two distinct types are represented by two different bit-vectors.

Similarly, let bv_i and bv_j be two different bit-vectors. Then it necessarily exists a k such as $b_{ik} \neq b_{jk}$. Therefore $\exists e_k, (e_k \notin u_i \vee e_k \notin u_j) \wedge e_k \notin u_i \cap u_j$. Hence $u_i \neq u_j$. □

[8] Actually this is not completely accurate since the type `string` can be described using array type specifiers. However, since the latter are not supported by our implementation yet, we consider the types `string` and `bit-vector` as being literal types since their symbolic representation is kept through the entire process. This is very likely to change in the future.

[9] Our implementation does not support complex numbers yet, and considers the `complex` type as being empty. Some wrong results arise from that supposition—such as `(subtypep 'number 'real)` returning true. This will change as soon as complex numbers are supported.

Proof. *Type intersection, union and complement are equivalent to bitwise Boolean operations "and", "or" and "not" on representative bit-vectors.*

Let two types u_i and u_j in:

(1) Let $u_k = u_i \cup u_j$. By definition, $\forall l \in \mathbb{N} \cup \{\omega\}, b_{kl} = 1$ iff $b_{il} = 1$ or $b_{jl} = 1$, that is $b_{kl} = b_{il} \vee b_{jl}$. Thus, also by definition:

$$\begin{aligned} bv_k &= [b_{k0}, b_{k1}, \ldots, b_{k\omega}] \\ &= \left[b_{i0} \vee b_{j0}, b_{i1} \vee b_{j1}, \ldots, b_{i\omega} \vee b_{j\omega}\right] \\ &= bv_i \vee bv_j \end{aligned}$$

(2) We proceed similarly for the intersection and the Boolean logical operator "and" ($\wedge$).

(3) Let $u_k = \overline{u_i}$. We have by definition $\forall l \in \mathbb{N} \cup \{\omega\}, b_{kl} = \neg b_{il}$. Then:

$$\begin{aligned} bv_k &= [\neg b_{i0}, \neg b_{i1}, \ldots, \neg b_{i\omega}] \\ &= \neg bv_i \end{aligned}$$

□

5.1.2 Implementation. Common Lisp cannot enumerate all the possible subtypes of `t` nor all of its elements. Fortunately, we do not need them all. We only need to consider the types mentioned in the input type specifier to determine its emptiness.

We also do *not need to enumerate all the elements of these types.* It is that aspect of the procedure of Baker that makes it both powerful and difficult to understand at first. We only need *sufficiently many* elements from a type to distinguish it from the other types. Because we are now considering only a *finite* number of types, say $u_1, \ldots, u_n$, to register a new type u_{n+1} to our (now *finite*) matrix, we only need to find an element $e \in u_{n+1}$ such as $e \notin u_1 \cup \cdots \cup u_n$.

Now let's suppose that the type specifier of u_{n+1} is in fact `(member` e`)`, that e is itself chosen as a representative element for another type, say u_k, and that u_k is only distinguished from the other registered types by that element e. u_{n+1} and u_k would then have the same bit-vector representation when these types are likely to be distinct. The general solution for that kind of problem is to register *all* the elements found inside the `member` type specifier. When there is a conflicting element e already registered as a representative for another types, we generate additional representatives for these types. That precaution ensures that this kind of conflict never happens and greatly simplifies the implementation of `member` type specifiers.

To implement that registration matrix system, we use two functions: B : type name $\longmapsto$ bit-vector, with $B(u_i) = bv_i$, and I : representative $\longmapsto$ bit index, with $I(e_i) = i - 1$. Baker suggests in his small example [3] using the operator `set` which is deprecated in modern Common Lisp programming. Instead, we use hash tables to represent these functions. Type names are `symbols`, bit-vectors are `bit-vectors` and element indexes are positive `integers`. To register a new type u_{n+1}, it is added to the B hash table and its bit-vector content $b_{(n+1)i}$ is evaluated for all the existing representatives ($i \in [\![1; m]\!]$). To register a new representative e_{m+1}, it is added to the I hash table with the index m. Then we add one bit (the m-th bit) to each bit-vector bv_i and evaluate it in respect to the type u_i. Thus, to retrieve the bit-vector of a registered primitive or user-defined type t, we just lookup its value $B(t)$. To compute the bit-vector of a `member` expression `(member` $e_1 \cdots e_n$`)`, we use the value $B($`(member` $e_1 \cdots e_n$`)`$) = \bigvee_{i=1}^{n} \beta\left(I\left(e_i\right)\right)$, where $\beta(x)$ returns the null bit-vector with the x-th bit activated.

The bit-vector of logical type specifiers are given in Eq. 1, Eq. 2 and Eq. 3 thereafter.

$$B(\texttt{(and } U_1 \cdots U_n\texttt{)}) = \bigwedge_{i=1}^{n} B(U_i) \tag{1}$$

$$B(\texttt{(or } U_1 \cdots U_n\texttt{)}) = \bigvee_{i=1}^{n} B(U_i) \tag{2}$$

$$B(\texttt{(not } U\texttt{)}) = \neg B(U) \tag{3}$$

5.1.3 Issues. The method for choosing the representative elements for a type depends of its nature: it can be a primitive type, a user-defined type (class, structure or condition) or a `member` expression.

Since primitive types are known (c.f. table 4.2 of [2]), their representative elements are chosen at compile-time. The u_{n+1} subtlety above should still be kept in mind. For instance, the type `null` is a subtype of both `symbol` and `list` ; so three representative elements are needed: `nil`, a non-empty list and a symbol other than `nil`. Note that some primitive types are an *exhaustive partition* of other types (e.g., `character` ≡ `(or base-char extended-char)`). Obviously, in that case, such a precaution does not apply.

For user-defined types, Baker suggests to extend the type creation mechanism—thus modifying the implementation's internal functions—to register a dummy element as a representative. We decided *not* to follow his approach because of the poor portability of his solution. Indeed, this work, often non-trivial, would have to be repeated for each targeted Common Lisp implementation. (We would like to avoid modifying the SBCL internal mechanisms.) Moreover, it would register a representative for *every* class created, thus increasing bit-vectors' size uselessly since only a few of these classes are likely to appear in a `subtypep` type specifier. But more importantly, the main drawback of his solution is that creating that dummy element might have unexpected side-effects, as it may need to use slot's default values and/or `initialize-instance`. We decided instead to use the *Meta Object Protocol* (MOP) [6], more specifically *class prototypes.* Class prototypes are pseudo-instances of a class, created without executing `initialize-instance` and which `typep` and `eql` view as traditional instances. However, to create a class prototype, the class needs to be *finalized* and it cannot be guaranteed until it is instantiated. Since that class may be involved in a `subtypep` call before that happens, when a new class is encountered, we force its finalization using the function `ensure-finalized` from the (portable) `closer-mop` package[10]. Then, we create the prototype of the class using `sb-mop:class-prototype` and register it. This method is much more portable than Baker's and does not require to hook inside the implementation.

Since (in SBCL[11]) conditions are classes, they are supported automatically. The Common Lisp standard [1] states that "`defstruct` without a `:type` option defines a class with the structure name as its name", hence in that case no additional work is required. The standard also states that "Specifying this option [...] prevents the

[10] http://common-lisp.net/project/closer/

[11] Every major lisp implementations implement conditions as CLOS classes—the most obvious way to do it. We ignore exotic condition implementations.

structure name from becoming a valid type specifier recognizable by typep." Thus, subtypep is not concerned by these types of structures.

To address the misrepresentation problem when member type specifiers are involved, as discussed in Section 5.1.3, we must ensure that a new representative element is generated and registered. The Common Lisp standard ([1]) states that the member type specifier is defined in terms of eql. That is, (typep e '(member $e_1 \cdots e_n$)) uses eql to compare e to the successive e_k to check the membership. That precise property reduces the misrepresentation problem to only two types: symbol and character (and their subtypes).

To better understand why it is the case, first consider a reduced version of the top-level type t: t = (or string list symbol). Then, let R = ("hello" (1 2 3) foo) be our list of representatives.

(1) Let's ask the question (subtypep 'symbol '(member foo)).
(2) As discussed in Section 5.1.3, we add the elements of the member expression to R. To conform with the specification, we first check whether or not foo is already in R eql-wise: foo $\in_{\text{eql}}$ R, so R does not change.
(3) As shown in Eq. 4, the emptiness check passes, meaning that symbol is indeed a subtype of (member foo), which is obviously wrong.

$$\begin{aligned} B(\texttt{symbol}) \wedge \neg B(\texttt{(member foo)}) &= 001 \wedge \neg 001 \\ &= 001 \wedge 110 \\ &= 000 \\ &= B(\texttt{nil}) \end{aligned} \tag{4}$$

However, for lists, that problem does not appear, thanks to the eql-wise comparison.

(1) (subtypep 'list '(member (1 2 3)))
(2) (1 2 3) $\notin_{\text{eql}}$ $R \Rightarrow R$ = ("hello" (1 2 3) foo (1 2 3))
(3) As shown in Eq. 5, the emptiness check fails and the answer is correct.

$$\begin{aligned} B(\texttt{list}) \wedge \neg B(\texttt{(member (1 2 3))}) &= 0101 \wedge \neg 0001 \\ &= 0101 \wedge 1110 \\ &= 0100 \\ &\neq B(\texttt{nil}) \end{aligned} \tag{5}$$

Within the literal types kingdom, the only types for which this problem occurs—since the representatives are not supposed to be accessible to the user of subtypep—are then symbol and character. Therefore, only the representatives of these types need to be actually checked when registering member's elements.

To generate a new symbol, we use alexandria:symbolicate[12]. The keyword subtype of symbol is also subject to the problem. (Actually, solving the problem for keywords also solves the problem for symbols.) To generate a new character, we first need to know whether it is a base-char or an extended-char. Then we pick a character of that type not registered yet. When all the characters of that type are registered there is nothing to do (since the type is fully represented in the matrix, no misinterpretation can occur).

We have not addressed the problem of a type specifier involving a user class C and a member expression containing the class prototype of C yet.

5.2 Procedure for numeric types

Unlike the literal type kingdom, the range type kingdom does not need an internal state to represent numeric types. Indeed, the expert sub-procedure takes as input an already precise enough representation of the type described. Range type specifiers allow to describe which kind of number is specified (its type, e.g., integer, ratio, etc.), its bounds (inclusive and exclusive, e.g., (integer (0) 6)) and is able to represent non-bounded intervals through the symbol * meaning infinity (e.g., (float * 0.0) $\equiv [-\infty; 0.0]$). The range type specifier is *as precise as the mathematical range notation.* Additionally, the mathematical union, the intersection and complement of these ranges can be expressed equally using the corresponding logical type specifier.

Therefore, to assert about the emptiness of the input type specifier, checking whether the *canonicalized* version of this interval expression describes the empty range (i.e., nil) is *sufficient.* The calculation is performed by three successive steps, which we describe in the following sections.

This algorithm suffers from an exponential time and space complexity. However, Baker claims that in practice, that theoretical complexity is not an issue (it only appears for "highly artifical cases"). We have not tried to prove (or invalidate) his statement but Section 6 shows some early results that tend to support his claim.

We use a custom abstraction, the interval class, closer to the mathematical object (with type, bounds and limits slots). Thus we avoid the annoying manipulation of lists (with the many standardized ranges syntaxes). The first step is to write a function range->interval that converts (using pattern matching) a range type specifier to its corresponding interval instance. This function also takes care of the exotic compound forms—such as (unsigned-byte s) which describes the integer range $[0; 2^s - 1]$. We also use a similar structure for interval operations to fully discard the list representation.

We also need the following interval functions:

- (interval-and I_1 I_2) — returns $I_1 \cap I_2$ if their types are eql, or $\emptyset$ otherwise.
- (interval-or I_1 I_2) — returns $I_1 \cup I_2$ if their types are eql and $I_1 \cap I_2 \neq \emptyset$, or $\emptyset$ otherwise.
- (interval-minus I_1 I_2) — returns $I_1 - I_2$ (may return two values when $I_2 \subset I_1$) if their types are eql, or I_1 otherwise.
- (interval-empty-p I) — returns whether $I = \emptyset$.

5.2.1 *Type diversity reduction.* Functions working with intervals must be aware of the relationship of the types of these intervals. For example, the intersection of two integer intervals might be non-empty whereas the intersection of one integer and one single-float intervals is always null as these two types are disjoint. However, integer and fixnum are different types but the intersection of intervals of such types *might be non-empty.* The subtype relationship of the types of intervals needs to be introspected to accurately apply some operations (such as intersection or union).

The type number (complex numbers being ignored) is an exhaustive partition of six mutually disjoint types: integer, ratio, single-float, short-float, double-float, and long-float. Baker advises to define what he calls "simple intervals", that is intervals guaranteed to have their type equal to one of these six types. This

[12] https://common-lisp.net/project/alexandria/

Supertype	*Conversion*
`number`	`(or rational float)`
`real`	`(or rational float)`
`rational`	`(or integer ratio)`
`float`	`(or short-float single-float double-float long-float)`
`bignum`	`(or integer (not fixnum))`

Table 1: Conversion table for supertypes

way, as these types are mutually disjoint, operations on intervals of such types have their implementation greatly simplified.

To convert each numeric type into its equivalent using only the six types above, a two-step conversion is required.

1. For intervals whose `type` is a *supertype* of one of these types, the conversion table 1 is used. E.g.: the conversion of `(rational` a b`)` gives `(or (integer` a b`) (ratio` a b`))`.
2. For intervals whose `type` is a *bounded subtype* (i.e.: having defined bounds, not infinity) of these six types, their actual bounds have to be constrained to fit within the bounds of their type, before being converted to their corresponding supertype. For example, `(fixnum 12` 2^{100}`)`, has to be converted to `(integer most-negative-fixnum most-positive-fixnum)`, where `most-positive-fixnum` $< 2^{100}$, as 2^{100} is a `bignum`, thus discarding the numbers in between. A similar procedure is applied to the types `bit`, `short-float`, `single-float`, `double-float` and `long-float`.

Eventually, the `type` of every `interval` is constrained to one of the six types above, with the bounds (if some) of their original type preserved.

5.2.2 Canonicalization. To check the emptiness of the interval expression, it is canonicalized. Let Γ be the canonicalization function. Its parameter is either an `interval` I or an operation on intervals χ (intersection, union or complement). Γ either returns $\emptyset$, an `interval` or a *union of disjoint intervals*—the three possible outcomes of a mathematical interval canonicalization.

First and foremost, anytime Γ encounters or returns a union, it must ensure that it is *flattened* (no nested unions). It must also ensure that the intervals inside the union are *disjoint*. As shown in Section 5.2.1, intervals with different types are necessarily disjoint. *Touching intervals* [3] are *merged* using `interval-or`.

$\Gamma(\emptyset)$ and $\Gamma(I)$ are straightforward, as shown in Eq. end-$\emptyset$ and Eq. end-I. These are the terminal cases of the recursion of Γ.

$$\Gamma(\emptyset) = \emptyset \qquad \text{(end-}\emptyset\text{)}$$

$$\Gamma(I) = I \qquad \text{(end-}I\text{)}$$

Intersections (`and` logical type specifiers) are reduced as soon as they are encountered. Their operands need to be processed by Γ first (hence the implicit mapping "$k \longrightarrow n$"). Eq. `and`-apply shows how to reduce intersections. The Φ_f operator denotes a *fold* [5] operation using the function f. $\Gamma \circ \cap$ denotes the composition of the Γ function and the intersection operator. To break it down in a bottom-up fashion:

1. Eq. `and`-final — the application of the intersection function.
2. Eq. `and`-distribution′ — the distribution of the intersection over the union. Next step is Eq. `and`-final.
3. Eq. `and`-distribution — also the distribution of the intersection over the union. However, $\Gamma(\chi)$ may return an union, leading the execution either to Eq. `and`-distribution′ or directly to Eq. `and`-final.
4. Eq. `and`-apply — the canonicalization of the χ_n forms using mapping. The results are then folded using $\Gamma \circ \cap$, thus initiating the recursive intersection distribution.

$$\Gamma\left(\bigcap_n \chi_n\right) = \Phi_{\Gamma\circ\cap}\,\Gamma(\chi_k)_{k\longrightarrow n} \qquad \text{(and-apply)}$$

$$\Gamma\left(\chi \cap \bigcup_n I_n\right) = \bigcup_n \Gamma\left(\Gamma(\chi) \cap I_n\right) \qquad \text{(and-distribution)}$$

$$\Gamma\left(\bigcup_n I_n \cap I\right) = \bigcup_n \Gamma(I_n \cap I) \qquad \text{(and-distribution}'\text{)}$$

$$\Gamma(I_1 \cap I_2) = \texttt{(interval-and } I_1\ I_2\texttt{)} \qquad \text{(and-final)}$$

Complements (`not` logical type specifiers) are also reduced as soon as they are encountered. Their only operand is first canonicalized. Complementing U in `number` (the top-level type of the range type kingdom) is equivalent to the difference `number` $- U$, as shown in Eq. `not`-apply. The difference canonicalization goes through a similar recursive distribution path than the intersection, that is Eq. `minus`-distribution and then Eq. `minus`-apply. Note that this path is taken every time since the interval difference is an internal operation and that its left-hand operand is always $\mathcal{U}$.

$$\Gamma(\overline{\chi}) = \Gamma\left(\mathcal{U} - \Gamma(\chi)\right) \qquad \text{(not-apply)}$$

$$\mathcal{U} = \langle \textit{type diversity reduction of } \texttt{(number * *)} \rangle$$

$$\Gamma\left(\chi - \bigcup_n I_n\right) = \bigcup_n \Gamma(\chi - I_n) \qquad \text{(minus-distribution)}$$

$$\Gamma\left(\bigcup_n I_n - I\right) = \bigcup_n \texttt{(interval-minus } I_n\ I\texttt{)} \qquad \text{(minus-apply)}$$

5.2.3 Range emptiness check. Once an interval expression canonicalized, checking its emptiness is trivial. The predicate `interval-empty-p`, given the result of the first Γ call, just returns the Boolean that `null-numeric-type-p` has to return.

5.3 Array types and `cons` type specifiers

This section presents some preliminary work and research results found on array and `cons` type specifiers. Obviously, since the implementation of the expert sub-procedures for these kingdoms is still a work in progress, no result nor implementation guidelines are provided here. It does, however, give some insights about how Baker procedure applies to modern Common Lisp implementations such as SBCL.

Array type specifiers are complex to handle because they are bi-dimensional: it has an element type and bounds (e.g., `(array integer (* 2 *))`). Internally, Common Lisp implementations do not store which exact type specifier is specified but rather only store

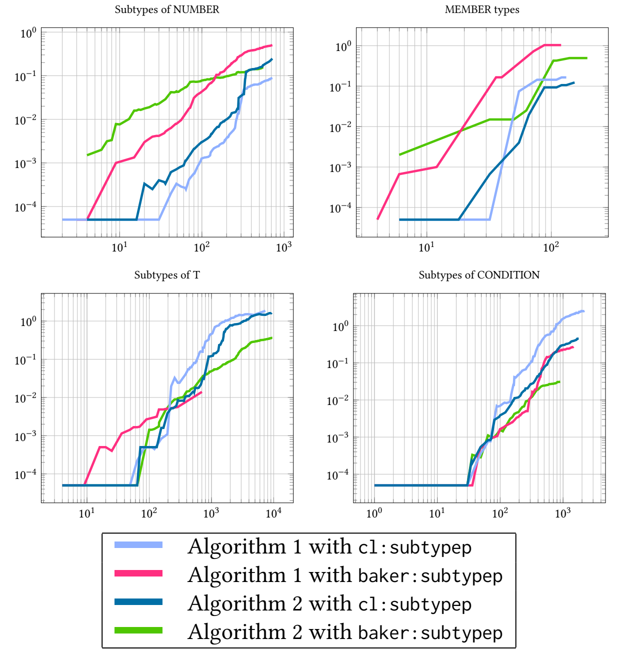


Figure 3: Comparative efficiency measures of our `subtypep` implementation

the result of the function `upgraded-array-element-type` returns giving that type. E.g, for `(make-array 2 :element-type 'list)`, the implementation does not makes an array of `list` but rather an array of `(upgraded-array-element-type 'list)`. For every value that might return this function, Baker requires that we store a bit matrix (instead of bit vectors) because of the complex bounds logic of the type specifier. As for the literal type procedure, it seems to be an efficient type representation system—albeit more complex—which nonetheless requires an extra registration step and a global state.

Baker does not mention the `cons` type specifier family at all in his article because it appeared after he released his article [4]. An accurate expert sub-procedure for this kingdom would have an exponential complexity. More investigation is needed to assert whether or not that exponential time is "acceptable" (as it is for ranges) before rejecting it. The accuracy of existing `subtypep` procedures for the `cons` type specifier also needs to be studied.

6 EARLY RESULTS

Our implementation of `subtypep` is still in active development and very experimental. No serious optimization work has been made. Nonetheless, Newton has compared in [7] the performances of several `subtypep` highly dependent algorithms, both using the implementation of Sbcl and ours.

These results, shown in Figure 3, are only presented here as complementary information. On the horizontal axis is the size of the type specifiers and on the vertical axis is the measured execution time. Hence, the lower a curve is, the better. As expected, our implementation is often slower, but not dramatically, which is encouraging.

- Our implementation is overall slower in the range type kingdom.
- Heavy users of `member` seems to experience a slower execution. Perhaps, as predicted by Baker, the reason is that the systematic registration of the elements makes the size of the bit-vectors grow quickly, thus making every subsequent operation slower.
- For the symbolic type specifiers—primitive types, Clos classes and conditions—our implementation already outperforms Sbcl's.

7 CONCLUSION AND FUTURE WORK

Throughout this article we presented our implementation of Baker's decision procedure. In Section 2 we introduced the Common Lisp type system, the notion of type specifier and some vocabulary. In Section 4 we explained how to pre-process the caller's type specifiers to make the work of the expert sub-procedures presented in Section 5 easier. We described our implementation for the symbolic, `member`, range and logical type specifiers. We also gave some insights about the implementation for the `array` and `cons` type specifiers. We finally presented some early efficiency measures, which are globally encouraging.

Our implementation is still a work in progress and highly experimental. But with some cleaning and the implementation of both `array` and `cons` expert sub-procedures, it could be a viable alternative to existing `subtypep` implementations. We *will* have open sourced its code by then. We still have to find a solution for the `satisfies` type specifier and the related uncertainty. Indeed, in some situations, `subtypep` still can answer even though the type specifier is involved. For example, in `(subtypep 'string '(and number (satisfies evenp)))`, as the second operand is guaranteed to be a subtype of `number`, the predicate can safely return false. Finally, a lot of measures on accuracy and efficiency are needed to assert whether Baker's intuition about his procedure was correct or not.

Even if, in the future, we are to conclude that our implementation is less efficient than those which already exists, Baker's algorithm would still likely to improve the predicate's accuracy. Lispers would then have the ability to choose whichever `subtypep` implementation fits their needs the best.

REFERENCES

[1] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.
[2] Ansi. American National Standard: Programming Language – Common Lisp – Type Specifiers (Section 4.2.3). ANSI X3.226:1994 (R1999), 1994. http://www.lispworks.com/documentation/lw50/CLHS/Body/04_bc.htm.
[3] Henry G. Baker. A Decision Procedure for Common Lisp's SUBTYPEP Predicate. *Lisp and Symbolic Computation*, 1992.
[4] Paul F. Dietz. "subtypep tests" discussion on gcl-devel, 2005. https://lists.gnu.org/archive/html/gcl-devel/2005-07/msg00038.html.
[5] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999. URL http://dblp.uni-trier.de/db/journals/jfp/jfp9.html#Hutton99.
[6] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.
[7] Jim Newton. *Representing and Computing with Types in Dynamically Typed Languages*. PhD thesis, Sorbonne Université, Paris, France, November 2018.
[8] Jim Newton and Didier Verna. Approaches in typecase optimization. In *European Lisp Symposium*, Marbella, Spain, April 2018.
[9] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.

make-method-lambda revisited

Irène Durand
Robert Strandh
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ABSTRACT

The Common Lisp metaobject protocol specifies a generic function named `make-method-lambda` to be called at macro-expansion time of the macro `defmethod`. In an article by Costanza and Herzeel, a number of problems with this generic function are discussed, and a solution is proposed.

In this paper, we show that the alleged problems are due to the fact that existing implementations do not include proper compile-time processing of the associated macro `defgeneric`, and that with proper compile-time processing, the problems indicated in the paper by Costanza and Herzeel simply vanish.

The main characteristic of our proposed solution is for the compile-time side effects of `defgeneric` to include saving the name of the method class given as an option to that macro call. With this additional information, no difference exists between the behavior of direct evaluation and that of file compilation of a `defgeneric` form and a `defmethod` form mentioning the same name of the generic function.

CCS CONCEPTS

• **Software and its engineering → Abstraction, modeling and modularity**; **Software performance**; **Compilers**;

KEYWORDS

Common Lisp, Meta-Object Protocol

ACM Reference Format:
Irène Durand and Robert Strandh. 2019. make-method-lambda revisited. In *Proceedings of the 12th European Lisp Symposium (ELS'19).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.2634303

1 INTRODUCTION

In the definition of the Common Lisp [1] metaobject protocol in the book by Kiczales et al [3] (also known as the AMOP), the generic function `make-method-lambda` plays a role that is very different from most of the other generic functions that are part of the metaobject protocol.

According to the book, the function has four parameters, all required:

(1) A generic function metaobject.
(2) A (possibly uninitialized) method metaobject.
(3) A lambda expression.
(4) An environment object.

The main difference between `make-method-lambda` and other generic functions defined by the metaobject protocol is that `make--method-lambda` is called as part of the expansion code for the `defmethod` macro, whereas other generic functions are called at execution time.

The AMOP book states that the generic function passed as the first argument may be different from the one that the method is ultimately going to be added to. This possibility seems to exist to handle the situation where a `defgeneric` form is followed by a `defmethod` form in the same file. In this situation, the Common Lisp standard clearly states that the file compiler does not create the generic function at compile time. Therefore, when the corresponding `defmethod` form is expanded (and therefore `make-method-lambda` is called), the generic function does not yet exist. It will be created only when the compiled file is loaded into the Common Lisp system.

The AMOP book also states that the method object passed as second argument may be uninitialized, suggesting that the *class prototype* of the method class to be instantiated may be passed as the second argument.

The third argument is a lambda expression corresponding to the body of the `defmethod` form. The purpose of `make-method-lambda` is to wrap this lambda expression in another lambda expression called the *method lambda* which is ultimately compiled in order to yield the *method function*.

The default method lambda returned by an invocation of `make--method-lambda` is a lambda expression with two parameters. The first parameter is a list of all the arguments to the generic function. The second parameter is a list of next methods that can be invoked using `call-next-method` from the body of the method. Therefore `make-method-lambda` also provides definitions of `call-next--method` and `next-method-p` that are lexically inside the lambda expression it returns.

It is important that the method lambda is returned as part of the expansion of the `defmethod` macro and that it is then processed in the same environment as that of the `defmethod` form itself, so that when the `defmethod` macro call is evaluated in an environment that is not the *null lexical environment*, that environment is taken into account when the method lambda is processed. For example, code like this one:

```
(let ((x 10))
  (defmethod foo ((y integer))
    (+ x y)))
```

should work as expected.

ELS'19, April 01–02 2019, Genova, Italy

ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2634303

Finally, the fourth argument to make-method-lambda is an environment object.

2 PREVIOUS WORK

In their article [2], Costanza and Herzeel give a simple example of this simple defmethod form:

```
(defmethod foo ((x integer) (y integer))
  (do-something x y))
```

and at the end of section 2.1, on page 3, they claim that the expansion of that form is "something like" the follow form:

```
(let ((gf (ensure-generic-function 'foo)))
  (multiple-value-bind
      (lambda-expression extra-initargs)
      (make-method-lambda
        gf
        (class-prototype
          (generic-function-method-class gf))
        '(lambda (x y) (do-something x y))
        lexical-environment-of-defmethod-form)
    (add-method
      gf
      (apply #'make-instance
             (generic-function-method-class gf)
             :qualifiers '()
             :lambda-list '(x y)
             :specializers (list (find-class 'integer)
                                 (find-class 'integer))
             :function (compile nil lambda-expression)
             extra-initargs))
```

except that we have formatted the code to fit the page, and we have added two missing closing parentheses at the end of the form.

This example is a slight variation on the code that is shown in section 5.5.1 of the AMOP book. However, in that section, it is not claimed that this code is the result of expanding a defmethod form. Rather, it is given as "an example of creating a generic function and a method metaobject, and then adding the method to the generic function".

Indeed, this expansion is not possible, at least not in a compiling implementation, which is the premise of both the paper by Costanza and Herzeel and this one. It has two fundamental problems:

(1) The call to make-method-lambda must be made at macro-expansion time, whereas in their example, the call is present in the expansion, so it will be made at load time.
(2) In their example, the resulting method lambda is compiled in the null lexical environment. However, compiling in the null lexical environment would violate the semantics of the Common Lisp standard, which requires that the body of the defmethod form be compiled in the lexical environment in which it appears.

In section 5.4.3 of the AMOP book, an example of an expansion is shown, and figure 5.4 clearly mentions that make-method-lambda is called during the macro-expansion phase. Furthermore, in figure 5.3, which shows the expansion of the defmethod macro, no call to compile is made. The result of calling make-method-lambda, i.e., the *method lambda* is simply present in the expanded code.

As Costanza and Herzeel point out, the defmethod macro does not allow the programmer to specify a class for the method to be created. That class must be determined by the generic function to which the method is ultimately going to be added. Therefore, in the case of a defgeneric form followed by a defmethod form, the method class must be the one indicated in the defgeneric form.

The conundrum, then, is that the file compiler does not create the generic function as a result of compiling the defgeneric form, so when a defmethod form with the same name is encountered later in the same file, the method class can not be taken from the generic function metaobject. Otherwise, the normal way of obtaining the method class would be to call the accessor generic-function-method-class, passing it the generic function metaobject with the name indicated in the defmethod form. If there is no way for the file compiler to determine the method class when the defmethod form is encountered, then clearly the only choice is to call make-method-lambda with the class prototype of the class named standard-method as the second argument. However, the analysis by Costanza and Herzeel is that this behavior is a result of the file compiler calling ensure-generic-function to obtain a generic-function metaobject and then querying that object to obtain the method class. A simple experiment shows that this is not the case in SBCL for instance.

When the following code is compiled with the SBCL file compiler:

```
(defclass hello (standard-method) ())

(defgeneric foo (x y)
  (:method-class hello))

(defmethod foo (x y)
  (+ x y))

(eval-when (:compile-toplevel)
  (print (fdefinition 'foo)))
```

the compilation fails when an attempt is made to find the definition of foo in the last top-level form. Thus, after the defmethod form has been compiled, the generic-function metaobject still does not exist in the compilation environment.

However, tracing make-method-lambda prior to compiling the code above in a fresh compilation environment reveals that make-method-lambda is indeed called as a result of compiling the defmethod form, and that the second argument passed to the call is an instance of standard-method.

This situation can lead to some problems in client code that are amply described in the paper by Costanza and Herzeel. The essence of the problem is that, when a defgeneric form with a non-standard :method-class option is followed by a defmethod form in the same file, the file compiler may generate an expansion of the defmethod form that creates an instance of standard-method when the compiled file is ultimately loaded, as opposed to an instance of the method class with the name that was explicitly mentioned in the defgeneric form.

Furthermore, this behavior is inconsistent with the behavior when the source file is processed using load. The reason is that, contrary to the file compiler, load completely processes and evaluates each top-level form in order. As a result, when load is used, the generic function metaobject *is* created as a result of evaluating the defgeneric form, so that it *does* exist when the defmethod form is ultimately evaluated. Clearly, such inconsistent behavior between directly loading a source file and loading the result of applying the file compiler to it first is highly undesirable.

Perhaps even worse, even when the file compiler is used consistently, if the file is recompiled after having been loaded previously, the existing generic-function metaobject is reinitialized to have the correct method class, and the code works as when `load` is used.

The ultimate conclusion by Costanza and Herzeel is that, in order for the behavior of the file compiler to be consistent with that of loading the source file directly, and indeed for that behavior to be correct, the file compiler *must* create the generic function metaobject at compile time, so that it can be queried for the desired method class when the `defmethod` form is encountered. In the next section, we propose an alternative solution to this conundrum.

To solve the perceived problems with `make-method-lambda`, Costanza and Herzeel first analyze what desirable features this function has, and conclude that the following two are essential:

1. It can add new lexical definitions inside method bodies. This is the feature that is used to introduce definitions of `next-method-p` and `call-next-method`.
2. It can create lambda expressions for method functions with parameters in addition to the usual two, namely one for holding the arguments to the generic function and another for holding a list of next methods.

With these essential features in mind, Costanza and Herzeel then propose an alternative to `make-method-lambda` that does not have the perceived problem that this function has.

Their proposed solution has two parts:

1. They use custom method-defining macros. Such a macro would expand to a `defmethod` form, but this form can contain additional lexical definitions into the method body, introduced by the custom macro.
2. They propose that method functions always be able to take additional parameters in the form of Common Lisp keyword parameters. Furthermore, the use of the lambda-list keyword `&allow-other-keys` would make it easier to combine method functions that accept different additional arguments.

While it is able to solve the problem of the inconsistent behavior between `compile-file` and `load`, this solution has two major disadvantages as pointed out by Costanza and Herzeel:

1. With this solution, method functions have a lambda list that includes keyword parameters. Processing keyword arguments imposes a significant performance penalty on the invocation of method functions.
2. Existing CLOS implementations that use a lambda list without any keyword parameter for method functions are incompatible with this solution.

In the next section, we propose a solution that has neither of these disadvantages.

3 OUR TECHNIQUE

As permitted by the Common Lisp standard, the `defgeneric` macro may store information provided in the `defgeneric` form so as to make better error reporting possible when subsequent forms are compiled. In particular, the standard mentions storing information about the lambda list given, so that subsequent calls to the generic function can be checked for correct argument count. This information is kept in an implementation-specific format that does not contain the full generic-function metaobject, as this object is created when the compiled code resulting from the file compilation is loaded.

However, just as it is possible to keep information about the lambda list at compile time, it is also possible to keep information about the `:method-class` option given, or, when no option was supplied, the fact that the method class is `standard-method`.

With this additional information, during the expansion of the `defmethod` macro, the name of the method class can be retrieved, then a class metaobject from the name, and finally a class prototype from the class metaobject.

While the first parameter of `make-method-lambda` is indicated by the AMOP book as a generic-function metaobject, it is not specifically indicated that this object might be uninitialized, contrary to the method object that must be passed as the second argument. It is, however, indicated that the generic-function object passed as the first argument may not be the generic-function object to which the new method will eventually be added. Therefore, there is not much information that `make-method-lambda` can make use of. The exception would be the exact class of the generic function and the exact method class. It would be awkward for a method on `make--method-lambda` to access this information explicitly, rather than as specializers of its parameters. For that reason, the first argument to `make-method-lambda` might as well be a class prototype, just as the second argument might be.

As an example of how to accomplish this additional information, we suggest a solution with two parts:

1. The first part involves the possibility for the compiler to store information about a generic function in the compilation environment, as the result of compiling a `defgeneric` form. Specifically, the name of the generic-function class and the name of the method class would need to be stored, and later retrieved.
2. The second part requires a modification to the protocol used by the compiler to query the compilation environment in order to determine how a form is to be compiled.

For a solution to the first part, in Appendix A we show how additional information about a generic function can be stored and retrieved in the context of the protocol described in Strandh's paper on first-class global environments [5].

For the second part, recall that in section 8.5 of the second edition of Guy Steele's book on Common Lisp [4], the author describes a protocol for this kind of environment query. This protocol contains three functions for environment query, namely `function-information`, `variable-information` and `declaration-information`.

Not only are these functions inadequate for all the information that a compiler needs to determine about a function or a variable, but they are also hard to extend in a backward-compatible way. A modern version of this protocol would likely return standard objects as opposed to multiple values, thereby allowing for backward-compatible extensions on a per-implementation basis.

For the second part of our solution, the environment function `function-information`, when given the name that has previously been encountered in a defgeneric form, would have to return information about the name of the generic-function class and the method class. With this additional information, the expander for the macro defmethod would query the environment for this information, access the corresponding classes, and then the class prototypes, and finally call make-method-lambda with those prototypes as arguments.

While our solution is an improvement on the existing situation, it is clearly not perfect. For one thing, both the generic-function class and the method class mentioned in the defgeneric form must exist when the defmethod form is encountered, so that the class prototypes of these classes can be passed as arguments to make-method-lambda.

Also, when a custom generic-function class is used, it is possible that there exist custom methods on various generic functions for initializing instances of this custom class, and these custom methods could conceivably intercept and alter the method class in the generic-function metaobject thus created. In such a situation, our technique would then use incorrect information about the method class, and pass the wrong class prototype as the second argument to make-method-lambda.

4 CONCLUSIONS AND FUTURE WORK

We have defined a technique that alleviates a problem encountered in current Common Lisp implementations when a defgeneric form is followed by a defmethod form in the same compilation unit. When the defgeneric form mentions a method class other than standard-method, and the compilation unit is processed in a fresh compilation environment, current implementations do not propagate the information about the method class to the macro expander for defmethod, resulting in make-method-lambda being called with a method argument of the wrong class.

Our solution requires the compiler of the Common Lisp implementation to store a small amount of additional information about the generic-function class and the method class when the defgeneric form is encountered, and requires the macro expander for defmethod to retrieve this information by querying the compilation environment.

Contrary to the proposal by Costanza and Herzeel, our suggested solution does not introduce any incompatibilities that would render some existing code obsolete. Furthermore, our solution does not have the potential performance problem of the proposal by Costanza and Herzeel, i.e. the additional cost of processing keyword arguments to method functions.

However, there are still some situations where our technique does not work. In particular, when a custom generic-function class is used, and the initialization of instances of this class intercepts and alters the information about the method class as given in the defgeneric form. For cases like this, we suggest that the author of the custom generic function class also add a method on add-method, specialized to the custom generic-function class so as to verify that the class of the method being added is indeed correct, and signal an error otherwise.

Future work includes adding the functions defined in Appendix A to the SICL[1] protocol for first-class global environments.

The Cleavir compiler framework which is part of SICL defines a modern version of the protocol for environment query defined in the second edition of Guy Steele's book [4]. We plan to extend this protocol to include information about the name of the generic-function class and of the method class given (explicitly or implicitly) in a defgeneric form previously encountered in the current compilation environment. Since our existing protocol returns standard objects, no modifications to the existing Cleavir code will be required as a result of this extension. The extension will allow us to define the macro defmethod in SICL to query the environment, and to invoke make-method-lambda with appropriate arguments.

A PROTOCOL

In this appendix we present the additional generic functions making up the protocol for our first-class global environments.

In order for our definitions to fit in a column, we have abbreviated "Generic Function" as "GF".

function-class-name *fname env* [*GF*]

This generic function returns the name of the class of the function associated with *fname* in *env*.

If *fname* is not associated with an ordinary function or a generic function in *env*, then an error is signaled.

(setf function-class-name) *class-name fname env* [*GF*]

This generic function is used to set the class name of the function associated with *fname* in *env* to *class-name*.

If *fname* is associated with a macro or a special operator in *env*, then an error is signaled.

method-class-name *fname env* [*GF*]

This generic function returns the name of the method class of the function associated with *fname* in *env*.

If *fname* is not associated with a generic function in *env*, then an error is signaled.

(setf method-class-name) *class-name fname env* [*GF*]

This generic function is used to set the class name for methods of the function associated with *fname* in *env* to *class-name*.

class-name must be a symbol naming a class.

If *fname* is not associated with a generic function in *env*, then an error is signaled.

REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp.* American National Standards Institute, 1994.

[2] Pascal Costanza and Charlotte Herzeel. make-method-lambda considered harmful. Technical report, Vrije Univrsiteit Brussels, Belgium, 2008.

[3] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991. ISBN 0262111586.

[4] Guy L. Steele, Jr. *Common LISP: The Language (2Nd Ed.).* Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

[5] Robert Strandh. First-class global environments in common lisp. In *Proceedings of the 8th European Lisp Symposium*, ELS '15, pages 79 – 86, April 2015. URL http://www.european-lisp-symposium.org/editions/2015/ELS2015.pdf.

[1] See https://github.com/robert-strandh/SICL

Session III: Metaprogramming

Monday, 1.4.2019

14:30–15:00	Jim Newton and Didier Verna: Finite Automata Theory Based Optimization of Conditional Variable Bin
15:00–15:30	Marco Heisig: Lazy, parallel multiple value reductions in Common Lisp
15:30.-16:00	**Coffee break**

Finite Automata Theory Based Optimization of Conditional Variable Binding

Jim E. Newton
Didier Verna
jnewton@lrde.epita.fr
didier@lrde.epita.fr
EPITA/LRDE
Le Kremlin-Bicêtre, France

ABSTRACT

We present an efficient and highly optimized implementation of destructuring-case in Common Lisp. This macro allows the selection of the most appropriate destructuring lambda list of several given based on structure and types of data at run-time and thereafter dispatches to the corresponding code branch. We examine an optimization technique, based on finite automata theory applied to conditional variable binding and execution, and type-based pattern matching on Common Lisp sequences. A risk of inefficiency associated with a naive implementation of destructuring-case is that the candidate expression being examined may be traversed multiple times, once for each clause whose format fails to match, and finally once for the successful match. We have implemented destructuring-case in such a way to avoid multiple traversals of the candidate expression. This article explains how this optimization has been implemented.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; *Type theory*;

ACM Reference Format:
Jim E. Newton and Didier Verna. 2019. Finite Automata Theory Based Optimization of Conditional Variable Binding. In *Proceedings of the 12th European Lisp Symposium (ELS'19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.2635412

1 INTRODUCTION

The Common Lisp macro destructuring-bind [1] binds the variables specified in a given lambda list to the corresponding values in the tree structure resulting from the evaluation of a given expression. However, in the case that the tree structure of the expression does not coincide with the given lambda list, a run-time error is signaled. This error may pose a challenge to the programmer. The problem, simply stated, is that the destructuring lambda list [1, Section 3.4.5] is specified at compile time, and the expression is evaluated at run-time. Thus, it may not be possible to know until run-time that the input data is problematic. In certain cases the programmer would like to specify the run-time behavior to take if the match fails, rather than having an error signaled. This behavior cannot be specified portably using the condition system [1, Chapter 9], because the condition signaled is simply of type error with no additional information about exactly what failed. Furthermore, the programmer may not wish to signal an error at all, but rather detect the actual run-time pattern of the input data and proceed differently depending on which format of data is discovered.

```
(destructuring-case expression
  ((X Y)
   (declare (type fixnum X Y))
   :clause-1)
  ((X Y)
   (declare (type fixnum X)
            (type integer Y))
   :clause-2)
  ((X Y)
   (declare (type (or string fixnum) X)
            (type number Y))
   :clause-3))
```

Figure 1: Example of destructuring-case usage.

We presented destructuring-case in [11] as a mechanism to test run-time adherence of the destructuring lambda list to the value of a candidate expression. An example usage of this macro can be seen in Figure 1. This example shows three clauses, each with the same lambda list, (X Y), but with different type declarations. In general, a usage of destructuring-case may use radically different lambda lists, which differ in number of variables, having different &optional and &key sections, and also using different hierarchical structure of the variables.

The semantics of destructuring-case are that the value of the given expression is tested in turn against each of the given destructuring lambda lists, until a match is found, *i.e.* a match in both hierarchical structure and type of values. Only at such time are the indicated consequent expressions or any default values evaluated. This restriction is especially important if there are side-effects in the default values of optional arguments in the lambda lists such as (... &optional (x (incf *global-var*))).

ELS'19, April 01–02 2019, Genova, Italy

ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2635412

```
(rte-case expression
  ((:cat fixnum fixnum)
   (destructuring-bind (X Y) expression
     :clause-1))
  ((:cat fixnum integer)
   (destructuring-bind (X Y) expression
     :clause-2))
  ((:cat (or string fixnum) number)
   (destructuring-bind (X Y) expression
     :clause-3)))
```

Figure 2: Expansion of `destructuring-case` from Figure 1 into `rte-case`.

```
(rte-case expression
  ((:cat fixnum fixnum)
    :clause-1)
  ((:cat fixnum integer)
    :clause-2)
  ((:cat (or string fixnum) number)
    :clause-3))
```

Figure 3: Simple example of `rte-case` from Figure 2.

The implementations of the macros discussed in this article, including `destructuring-case`, `rte-case`, `rte-ecase`, and `bdd-typecase`, are available in Quicklisp[1] via the package `:rte`.

2 FROM DESTRUCTURING-CASE TO RTE-CASE

Our implementation of `destructuring-case` converts its input of destructuring lambda lists to rte (regular type expression) and then outputs an invocation of `rte-case`. The essential part of such an expansion is shown in Figure 2. An rte, introduced in [11], is Common Lisp syntax to specify a set of sequences, *i.e.* a subtype of the `sequence` type. We explain in Section 2.2 how a destructuring lambda list is converted to an rte.

As can be seen in Figure 2, each destructuring lambda list has been converted to an rte such as `(:cat fixnum fixnum)` in the first clause, followed by a call to `destructuring-bind`. As is implied by the syntax, the `destructuring-bind` will only be executed at run-time if the value of the candidate expression matches the pattern designated by the rte.

We further notice in the simplistic example shown in Figure 2, that no `destructuring-bind` in the `rte-case` expansion plays any role. The variables bound by the `destructuring-bind` are not used in the expressions which follow. Therefore, in our further discussion we will refer to the even simpler, semantically equivalent code in Figure 3.

A straightforward expansion of `rte-case` might include successive type checks of `expression` such as suggested in Figure 4. Such an expansion would be semantically correct, but inefficient because the sequence `expression` would be traversed three times in the worst case, to determine which consequent clause to evaluate. As will be seen, our technique eliminates these redundant traversals, allowing one single traversal of the sequence to be executed and thereby determining which consequent expressions to evaluate.[2]

```
(typecase expression
  ((rte (:cat fixnum fixnum))
   :clause-1)
  ((rte (:cat fixnum integer))
   :clause-2)
  ((rte (:cat (or string fixnum) number))
   :clause-3))
```

Figure 4: Naive expansion of `rte-case` from Figure 2

2.1 Examples of rte Syntax

The grammar an rte is explicitly detailed in [9]. Nevertheless, the basic grammar can be understood intuitively, assuming the reader has a basic understanding of string-based regular expression syntax. The concatenation operator, `:cat` specifies a sequences successive elements: *e.g.*, `(:cat fixnum string)` denotes a sequence of exactly two elements, the first of type `fixnum` and the second of type `string`. To make the `string` optional use the syntax `(:cat fixnum (:? string))`. To specify the occurrence, zero or more times, of `fixnum` followed by an optional string, use `(:cat (:* fixnum) (:? string))`. Substitute `:+` for `:*` to express an occurrence of one or more times. Finally, expressions may be combined logically using `:and`, `:or`, and `:not`, *e.g.*, `(:or (:cat fixnum string) (:+ (:not number)))`.

2.2 From Destructuring Lambda List to rte

In this section we summarize how a destructuring lambda list and associated type declarations may be converted into an rte. The conversion procedure is explained in more detail in [11].

The set of lists which are valid argument lists for a given invocation of `destructuring-bind` with an optional set of type declarations can be characterized by an rte. A destructuring lambda list, such as used in `destructuring-bind`, specifies a required portion, denoted by a leading sequence of variables; an optional portion, delimited by `&optional`; and a repeating portion of keyword value pairs, delimited by `&key`. To construct the rte corresponding to a given destructuring lambda list, we construct the *required-rte*, the *optional-rte*, and the *repeating-rte*, and concatenate them using the `:cat` operator.

`(:cat` *required-rte optional-rte repeating-rte* `)`

As an example, consider the lambda list shown in Figure 5. The required portion and optional portions are easy.

required-rte = `(:cat string string)`
optional-rte = `(:? list)`

[1] Quicklisp, `https://www.quicklisp.org/`, is a public repository, maintained by Zach Beane, consisting of user contributed Common Lisp libraries.

[2] The reader may well notice that a fourth traversal is also necessary to evaluate the `destructuring-bind` which is present in each of the consequent clauses. In this paper we do not address the elimination of this fourth traversal.

```
(destructuring-bind (A B &optional Q &key X Y)
    expression
  (declare (type string A B)
           (type list Q)
           (type real X)
           (type integer Y))
  ...)
```

Figure 5: Example `destructuring-bind` with declarations

The repeating portion deserves careful attention; we consider two restrictions.

(1) If `&allow-other-keys` *is not given*, such as is the case in Figure 5, then the only allowed keywords are those explicitly specified. In our case the only allowed keywords are `:X` and `:Y`, meaning the repeating portion is also of the form

```
(:* (:cat (member :X :Y) t)).
```

(2) Type declarations such as `(declare real X)` only restrict the value associated with the *first* occurrence of each keyword in an argument list, because only the first such occurrence is bound the the associated variable [1, Section 3.3.4]. A keyword portion of the argument list such as `(:X 1.2 :X 'not-real)` is perfectly valid, whereas `(:X 'not-real :X 1.2)` is not. Thus, we iterate over all specified keywords, generating one pattern for each. The pattern handling `&key X` requires that either there is either no `:X` given, or that the first `:X` is followed by a `real`. See the note `restriction 2` in Figure 6.

Putting all these restrictions together, we have the rte in Figure 6 representing the `destructuring-bind` with type declarations in Figure 5.

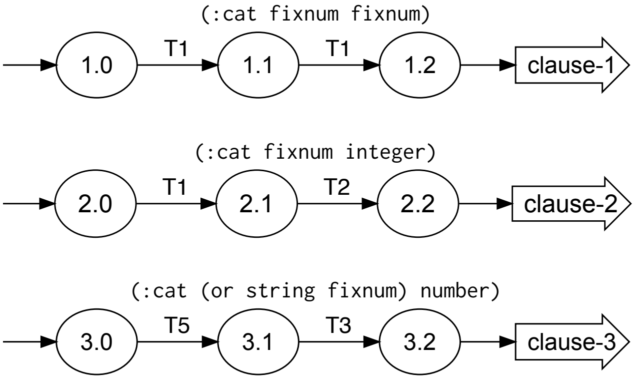
```
(:cat
 ;; required-rte
 (:cat string string)

 ;; optional-rte
 (:? list)

 ;; repeating-rte
 (:and
  ;; restriction 1
  (:* (:cat (member :X :Y) t))

  ;; restriction 2 for :X real
  (:or (:* (:cat (:not (eql :X)) t))
       (:cat (:* (:cat (:not (eql :X)) t))
             (eql :X) real
             (:* t)))

  ;; restriction 2 for :Y integer
  (:or (:* (:cat (:not (eql :Y)) t))
       (:cat (:* (:cat (:not (eql :Y)) t))
             (eql :Y) integer
             (:* t)))))
```

Figure 6: The rte representing the `destructuring-bind` and type declarations from Figure 5.

There are several other features of `destructuring-bind` which are supported by `destructuring-case`, but whose details we omit in this discussion, including tree structure variables/data, default values, *supplied-p-parameter*, `&allow-other-keys`, and others.

3 FROM RTE-CASE TO INDIVIDUAL DFAS

Each rte shown in Figure 3 can be converted to efficient type checking Common Lisp code, as explained in [11]. Such conversion involves first converting each rte to a deterministic finite automaton (DFA), where the transition labels represent type checks for successive elements of the candidate expression. Figure 7 shows the three DFAs corresponding to the `rte-case` in Figure 3.



Label	Type specifier
T_1	fixnum
T_2	integer
T_3	number
T_5	(or string fixnum)

Figure 7: Automata for clauses of `rte-case` in Figure 2

We now summarize how a deterministic finite automata (DFA) is constructed, given an rte. Some approaches to such generation, such as [6, 15], involve constructing a non-deterministic finite automaton and thereafter determinizing it. We use the technique presented by Brzozowski [2] and clarified by Owens [13]. The Brzozowski algorithm uses a technique called the rational derivative, to construct a DFA, and thereby obviating the necessity to determinize the result. In [9, 11], we explain how the rational derivative can be extended to accommodate Common Lisp types, in particular rather than calculating the rational derivative (as Owens suggests) with respect to each letter of the alphabet, instead we calculate the derivative with respect each type calculated in the maximal disjoint type decomposition as explained in [12].

3.1 Constructing States and Transitions

The algorithm can be summarized as follows. Each state in the DFA represents all the possible futures which are accepting. Moreover, there is a (not necessarily unique) rte which expresses that set of futures. For example, let:

$$P_1 = \texttt{(:or (:cat number string) (:cat fixnum float))}$$

be the rte representing all the sequences of either a `number` followed by a `string` or a `fixnum` followed by a `float`. Suppose there is a state in the DFA associated with this rte. Now we consider all the possible types of the first element of such a sequence. And for each such first element type, we calculate what the remaining future would be given that the first element of that type. If the first element is a `fixnum`, then the future is a sequence containing either a `string` or a `float`. Such a sequence is denoted by the rte `(:or string float)`. In terms of the rational derivative we say:

$$P_2 = \partial_{\texttt{fixnum}} P_1 = \texttt{(:or string float)}\,.$$

If, on the other hand, the first element is not a `fixnum` but is a `number`, then the remaining sequence whose only element is a `string`. That is to say:

$$P_3 = \partial_{\texttt{(and number (not fixnum))}} P_1 = \texttt{string}\,.$$

Since there is no other possible first element of P_1, we construct two additional states, P_2 and P_3 and construct two transitions $P_1 \longrightarrow P_2$ labeled `fixnum`, and $P_1 \longrightarrow P_3$ labeled `(and number (not fixnum))`.

We continue this process until all the futures of each state have been calculated, generating all the possible states, and all the possible transitions between the states.

3.2 Associating Code with Accepting States

DFAs used for matching pattern languages such as regular expressions, normally represent Boolean functions; returning TRUE if the sequence matches the expression, and FALSE otherwise. In our case each accepting state of the DFAs in Figure 7 indicate which code paths to take in the originating `rte-case`, Figure 3. This problem is easily addressed. We have simply extended our `state` object (CLOS class [5, 7]) to contain a slot indicating a piece of continuation code to be serialized in the final macro expansion.

3.3 Overlapping Clauses

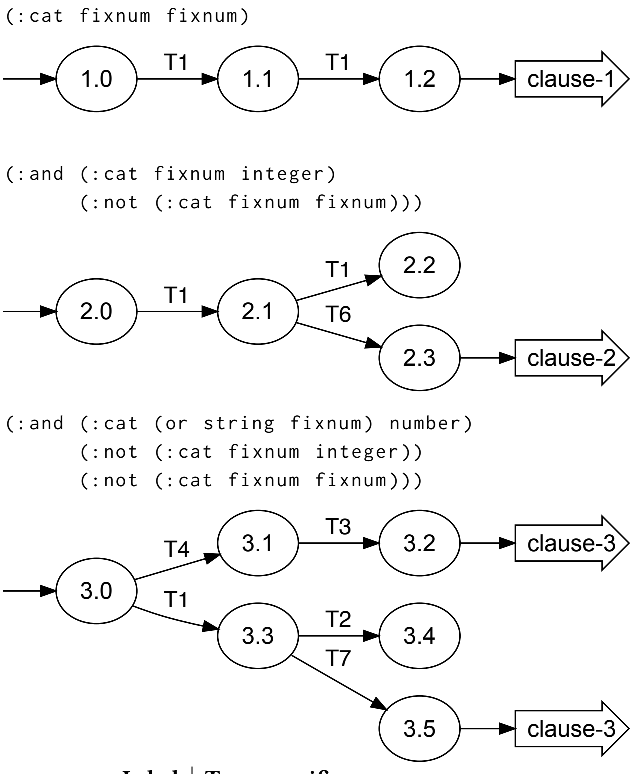
The *synchronized cross-product* (SXP) of two or more given DFAs is a single DFA whose behavior simultaneously emulates the behavior of the given DFAs. Typically such a cross-product implements the intersection or union languages of the input DFAs; however the semantics of such a cross-product can be taken to be any Boolean combination of the input.

For example, to implement the symmetric difference language we apply the Boolean XOR function; a state, X, in the SXP, corresponding to states A and B from two given DFAs, is marked as an *accepting* state if A XOR B are accepting (if either but not both are accepting). In our case we would like to select the code for evaluation corresponding to the code appearing first in the original `destructuring-case`; so we need priority based selection, rather than simply a Boolean function.

An important property of the behavior of `rte-case` is that if more than one pattern matches the expression in question, then the clause appearing first has priority over the others. For example, in the code in Figure 3, if the value of `expression` is the list `(1 2)`, then all three rtes match; nevertheless `:clause-1` must be the return value.

```
(rte-case expression
  ((:cat fixnum fixnum)
   :clause-1)
  ((:and (:cat fixnum integer)
         (:not (:cat fixnum fixnum)))
   :clause-2)
  ((:and (:cat (or string fixnum) number)
         (:not (:cat fixnum fixnum))
         (:not (:cat fixnum fixnum)))
   :clause-3))
```

Figure 8: Example of `rte-case` with pairwise disjoint patterns



```
(:cat fixnum fixnum)
```

```
(:and (:cat fixnum integer)
      (:not (:cat fixnum fixnum)))
```

```
(:and (:cat (or string fixnum) number)
      (:not (:cat fixnum integer))
      (:not (:cat fixnum fixnum)))
```

Label	Type specifier
T_1	`fixnum`
T_2	`integer`
T_3	`number`
T_4	`string`
T_6	`(and (not fixnum) integer)`
T_7	`(and (not integer) number)`

Figure 9: DFAs for disjoined clause-1, clause-2, and clause-3

An approach of addressing this ambiguity is to extend or augment the patterns so that they are mutually exclusive; *i.e.* assure that no two patterns simultaneously match any candidate expression. The code shown in Figure 8 is equivalent to that in Figure 3 but any input expression, `(1 2)`, for example, matches at most one pattern. This pattern augmentation can be accomplished as a code transformation. The pattern corresponding to `:clause-1` is unchanged, but the subsequent clauses have been augmented to emphasize that those clauses are never reached if any prior pattern matches.

These rtes correspond to the DFAs shown in Figure 9. The first DFA is exactly the same as before, but we notice in the second DFA that the state labeled 2.2 is non-coäccessible; *i.e.*, there is no path from state 2.2 to any accepting state. This non-useful state corresponds to `(:not (:cat fixnum fixnum))` in the input pattern, and it enforces that a sequence consisting of two objects of type `fixnum`, is a rejected sequence rather than a matching sequence. The third DFA in the figure contains a similar state, 3.4, but in addition, contains two states 3.2 and 3.5 which are equivalent to each other.

The disjoining process described here produces DFAs which have redundant or non-coäccessible states. Despite this fact, these slightly more complex DFAs play an important role in the SXP construction, because the process guarantees that the SXP construction will never encounter a situation where it must choose between two different pieces of code to execute on reaching an acceptance condition. If attempting to calculate the union of the three DFAs shown in Figure 7, the algorithm would have to deal with the fact that a sequence of `(1 2)` at run time should return `:clause-1` rather than `:clause-2`. However, if calculating the union of the DFAs from Figure 9, such ambiguity is averted. The union can be performed purely algebraically, with no consideration or order of priority.

4 MERGING DFAS INTO SYNCHRONIZED CROSS-PRODUCT DFA

We explain in detail in [9] how the type check associated with an rte is compiled to efficient Common Lisp code by first converting it to a deterministic finite automaton. It is further pointed out in the perspectives of [9] that it is desirable to *merge* these automata into a single automaton in order to share states between the various automata which serve the same function, and also to eliminate redundant traversals of the candidate expression. Having a single automaton which implements the union of the mutually exclusive patterns enables the candidate list to be traversed once and thereby matching any one of the expressions specified in the various clauses of the `rte-case`.

One advantage of the conversion from destructuring lambda list to rte is that rtes support an algebra sufficient for expressing sets of non-overlapping types, resulting in mutually exclusive patterns in the expansion to `rte-case`. As an additional feature of the implementation of `rte-case`, we have arranged so that it treats the code in Figure 3 and Figure 8 exactly the same, internally disjoining patterns which are not already disjoint.

The following is an explanation of how several automata are merged into such a single automaton.

We would like to merge the three DFAs shown in Figure 9 into a single DFA. There are well known techniques for merging multiple DFAs [6, 15] into the SXP DFA. These techniques are not general enough for several reasons which we address in our approach.

dfa_2	dfa_3	**intersection**	**Target State**
T_1	T_1	T_1	(2.1, 3.3)
T_1	T_4	$\emptyset$	
T_1	$\top \setminus (T_1 \cup T_4)$	$\emptyset$	
$\top \setminus T_1$	T_1	$\emptyset$	
$\top \setminus T_1$	T_4	T_4	$(\bot,\ 3.1)$
$\top \setminus T_1$	$\top \setminus (T_1 \cup T_4)$	$\top \setminus (T_1 \cup T_4)$	$(\bot,\ \bot)$

Figure 10: Transition Computation for $\mathrm{dfa}_2 \times \mathrm{dfa}_3$

It is not necessary to explicitly consider the SXP of more than two DFAs, because the operation is associative. Therefore, given the Common Lisp function `synchronized-product`, we may compute the SXP of one or more DFAs as a call to `cl:reduce`.

```
(reduce #'synchronized-product dfas)
```

4.1 Calculating States and Transitions

We consider constructing the SXP of two DFAs, `dfa-1` (with n states) and `dfa-2` (with m states). We construct a DFA, `dfa-3`, having $m \times n$ states, worst case; one state for each pair (x, y) with $x \in \mathrm{dfa}_1$ and $y \in \mathrm{dfa}_2$. Fortunately, this worst case does not often occur in practice as many of the states are not accessible. For example, if computing the SXP of the first two DFAs of Figure 9, there is no possible input sequence which would put dfa_1 into state 1.1 while putting dfa_2 into state 2.2. Thus there will be no state in the product DFA corresponding to $(1.1, 2.2)$.

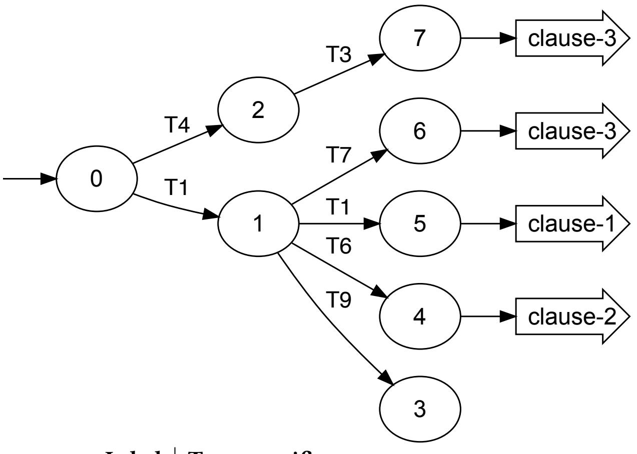
An efficient algorithm is described in [15]. We seed a work list with the one initial state. Next, we traverse the work list, growing it by adding new states as we construct them. All possible input types are considered for each state, and all possible transitions are generated.

An example will make this clearer. First start with dfa_2 and dfa_3, the second and third DFAs illustrated in Figure 9. The states list is initialized to $S = \{(2.0,\ 3.0)\}$.

We examine the behavior of states 2.0 and 3.0. We must characterize the behavior for every possible input. This infinite set of potential input values is partitioned into several disjoint types: those annotated on transitions exiting state 2.0 and 3.0, and the complement of their union. This complement type represents the set of all values for which an implicit transition leads to the virtual so-called *sync* state, denoted $\bot$. The sync state is a state which has exactly one exiting, all encompassing, transition: $\bot \xrightarrow{\top} \bot$.

State 2.0 has one explicit transition, namely $2.0 \xrightarrow{T_1} 2.1$. Thus, there is an implicit complement transition $2.0 \xrightarrow{\top \setminus T_1} \bot$, where $\top$ represents the universal type. State 3.0 has two explicit transitions: namely $3.0 \xrightarrow{T_1} 3.3$ and $3.0 \xrightarrow{T_4} 3.1$. Thus, there is an implicit complement transition $3.0 \xrightarrow{\top \setminus (T_1 \cup T_4)} \bot$.

To compute the transitions from (2.0, 3.0), we must consider all six pairwise intersections between the transition types of the two states (2.0 and 3.0). These intersections are shown in Figure 10, which also indicates the target states in the three non-empty cases.



Label	Type specifier
T_1	`fixnum`
T_3	`number`
T_4	`string`
T_6	`(and (not fixnum) integer)`
T_7	`(and (not integer) number)`

T_9	`(and integer`
	`    (or (not integer) fixnum)`
	`    (not fixnum))`

Figure 11: DFA for `rte-case` not yet reduced

Given an input of type `fixnum`, dfa$_2$ transitions from state 2.0 to state 2.1; and given the same input dfa$_3$ transitions from state 3.0 to state 3.3. So we add (2.1, 3.3) to S; $S = S = \{(2.0,\ 3.0),\ (2.1,\ 3.3)\}$, and add transition $(2.0,\ 3.0) \xrightarrow{T_1} (2.1,\ 3.3)$. Likewise, given an input of type `string`, dfa$_2$ transitions from state 2.0 to state $\perp$; and given the same input dfa$_3$ transitions from state 3.0 to state 3.1. So we add $(\perp,\ 3.1)$ to S; $S = S = \{(2.0,\ 3.0),\ (2.1,\ 3.3),\ (\perp,\ 3.1)\}$, and add transition $(2.0,\ 3.0) \xrightarrow{T_4} (\perp,\ 3.1)$. Finally, given an input of type `(and (not fixnum) (not string))`, dfa$_2$ transitions from state 2.0 to state $\perp$, and dfa$_3$ transitions from state 3.0 to state $\perp$. The state $(\perp, \perp)$ is the sync state of the cross product DFA so we need generate no additional transition from (2.0, 3.0).

Next, we to apply the same procedure to calculate any new states and transitions of any newly added elements of S. We continue the procedure until all elements of S have been visited, and no new states were generated.

After dfa$_2$ × dfa$_3$ has been computed, we can repeat the process via the `reduce` operation mentioned above to compute dfa$_1$ × dfa$_2$ × dfa$_3$. This procedure constructs a DFA isomorphic to that shown in Figure 11. We say *isomorphic* because the choice of state names is arbitrary. Figure 11 has states named 0 through 7 rather name names such as (1.0, 2.0, 3.0), (1.1, 2.1, 3.3) as suggested in the procedure description in Section 4.1.

The DFA shown in Figure 11 is not in minimal form. It has a non-coäccessible state, 3, from which there is no path to an accepting state. It also has indistinguishable states; *e.g.*, states 6 and 7 have the exact same future, albeit a trivial one of just returning the symbol `clause-3`. Since each of the states in the computed DFA and each of the transitions contribute to the number of lines of Common Lisp code which will be generated when the DFA is serialized in Section 5, we should simplify this DFA to reduce the lines of redundant code in the final macro expansion.

$s \in \mathbb{S}$	$v \in \Upsilon$	$\delta(s, v)$
0	T_1	1
0	T_4	2
1	T_1	5
1	T_6	4
1	T_7	6
2	T_3	7

$s \in \mathbb{S}$	$v \in \Upsilon$	$\psi_1(s, v) \in \Pi_0$
0	T_1	$\{0, 1, 2\}$
0	T_4	$\{0, 1, 2\}$
1	T_1	$\{5\}$
1	T_6	$\{4\}$
1	T_7	$\{6, 7\}$
2	T_3	$\{6, 7\}$

$s \in \mathbb{S}$	$\Phi_1(s)$
0	$\{\ (T_1, \{0, 1, 2\}),\ (T_4, \{0, 1, 2\})\ \}$
1	$\{\ (T_1, \{5\}),\ (T_6, \{4\}),\ (T_7, \{6, 7\})\ \}$
2	$\{\ (T_3, \{6, 7\})\ \}$
4	$\emptyset$
6	$\emptyset$
7	$\emptyset$

Figure 12: All values of the δ, ψ_1, and Φ_1 functions.

We eliminate non-coäccessible states by a simply *trimming* procedure based on graph traversal, finding states which lack a path to an accessible state. However, the procedure to coalesce indistinguishable states is more subtle, and we discuss it in Section 4.2.

4.2 DFA Simplification

The goal of simplification is to coalesce indistinguishable states such as states 6 and 7 in Figure 11, to result in the DFA in Figure 13.

In order to give a good explanation of the simplification algorithm we need some notation. Let $\mathbb{S}$ denote the set of states of the DFA, $\mathbb{S} = \{0, 1, 2, 4, 5, 6, 7\}$. Let Υ denote the set of all Common Lisp types annotated in the DFA: $\Upsilon = \{T_1, T_3, T_4, T_6, T_7\}$. Denote the state transfer function, δ, which given a state, $s_i \in \mathbb{S}$, and a type $v \in \Upsilon$, returns the target state, $s_j \in \mathbb{S}$ of the transition $s_i \xrightarrow{v} s_j$. The values of δ are given in Figure 12 (top left).

We will construct a sequence $\{\Pi_1, \Pi_2, ...\Pi_n, ...\}$ of partitions of $\mathbb{S}$. A *partition* of $\mathbb{S}$ is a set of mutually disjoint subsets of $\mathbb{S}$ for which the union of the subsets is $\mathbb{S}$ itself. Each element $\kappa \in \Pi_k$ is called a *k-equivalence class*. If $s_i, s_j \in \kappa$, then s_i and s_j are said to be *k-equivalent* to each other.

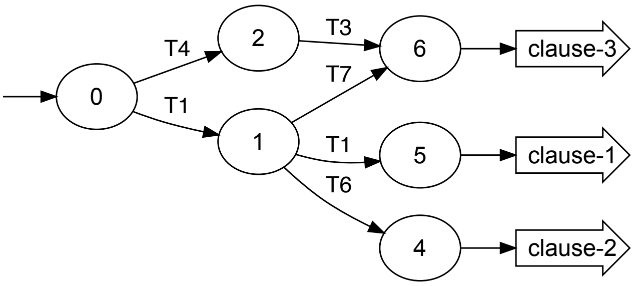
To construct the initial partition, Π_0, we group the set of all non-accepting states into one 0-equivalence class: $\{0, 1, 2\}$; thereafter, there is one 0-equivalence class per unique return value: `:clause-1`, `:clause-2`, and `:clause-3`: $\{5\}$, $\{4\}$, and $\{6, 7\}$ respectively.

$$\Pi_0 = \{\{0, 1, 2\}, \{4\}, \{5\}, \{6, 7\}\}$$

Next, we wish to construct Π_1, Π_2, ... Π_n, Π_{n+1} in turn, continuing the iteration until $\Pi_n = \Pi_{n+1}$. Each Π_k is derived from Π_{k-1} as we will explain.

For each integer $k > 0$, to determine the k-equivalence classes we define two functions ψ_k and Φ_k.[3] In each case, we will construct

[3] ψ is referred to as the partition transformation function. Φ is referred to as the partition image function.



Label	Type specifier
T_1	`fixnum`
T_3	`number`
T_4	`string`
T_6	`(and (not fixnum) integer)`
T_7	`(and (not integer) number)`

Figure 13: DFA for `rte-case` simplified

ψ_{k+1} and Φ_{k+1} by examining Π_k. These two functions may be difficult to understand intuitively from their mathematical definitions. Nevertheless, the mathematical definitions help when coding the simplification function in Common Lisp.

ψ_{k+1} is a function which takes two arguments, $s \in \mathbb{S}$ and $v \in \Upsilon$, and returns a k-equivalence class $\kappa \in \Pi_k$. (*I.e.*, $\psi_{k+1} : \mathbb{S} \times \Upsilon \to \Pi_k$) To compute the value of $\psi_{k+1}(s, v)$, we select and return the unique $\kappa \in \Pi_k$ for which $\delta(s, v) \in \kappa$. Figure 12 (top right) shows all the values of ψ_1.

Φ_{k+1} takes an element $s \in \mathbb{S}$ and returns a set of order pairs, each of the form (v, κ) where $v \in \Upsilon$ and $\kappa \in \Pi_k$. $\Phi_{k+1}(s)$ is defined as the set of all pairs $(v, \psi_{k+1}(s, v))$, such that $v \in \Upsilon$, and such that $\psi_{k+1}(s, v)$ exists. Figure 12 (bottom) shows all the values of Φ_1.

Now we construct the (k+1)-equivalence classes by splitting the k-equivalence classes; *i.e.* we refine Π_k to construct Π_{k+1}, so that every $\kappa \in \Pi_{k+1}$ contains those elements which have the same value of Φ_{k+1}. This rule implies that if κ has is a singleton set (*e.g.* $\{4\} \in \Pi_0$, and $\{5\} \in \Pi_0$), then $\kappa \in \Pi_{k+1}$ (*i.e.* $\{4\} \in \Pi_1$, and $\{5\} \in \Pi_1$).

Consider the 0-equivalence class $\{0, 1, 2\} \in \Pi_0$. Since $\Phi_1(0)$, $\Phi_1(1)$, and $\Phi_1(2)$ have three different values, then we must further partition $\{0, 1, 2\}$ into three distinct 1-equivalence classes $\{0\}$, $\{1\}$, and $\{2\}$.

Consider the 0-equivalence $\{6, 7\}$. Since $\Phi_1(6) = \Phi_1(7)$, then $\{6, 7\}$ is a 1-equivalence class, and $\{6, 7\} \in \Pi_1$.

$$\Pi_1 = \{\{0\},\ \{1\},\ \{2\},\ \{4\},\ \{5\},\ \{6, 7\}\}$$

If we repeat this process, generating the functions ψ_2 and Φ_2, and use Φ_2 to construct Π_2, we would find that $\Pi_2 = \Pi_1$, which means Π_1 is a fixed point of the procedure.

$$\Pi_2 = \{\{0\},\ \{1\},\ \{2\},\ \{4\},\ \{5\},\ \{6, 7\}\}$$

We can use Π_1, directly, to construct the minimum DFA shown in Figure 13. We simply merge the states which are 1-equivalent. We have determined that states 6 and 7 are 1-equivalent, and no others. We can thus construct the DFA in Figure 13 by merging states 6 and 7 from Figure 11.

```
(let* ((g1 expression)
       (g2 g1))
 (block check
  (tagbody
   s.0
     (unless g1 (return-from check nil))
     (typecase (pop g1)
       (fixnum (go s.2))
       (string (go s.1))
       (t (return-from check nil)))
   s.1
     (unless g1 (return-from check nil))
     (typecase (pop g1)
       (number (go s.3))
       (t (return-from check nil)))
   s.2
     (unless g1 (return-from check nil))
     (typecase (pop g1)
       (fixnum
        (go s.4))
       ((and (not integer) number)
        (go s.3))
       ((and (not fixnum) integer)
        (go s.5))
       (t (return-from check nil)))
   s.3
     (unless g1 (return-from check
       (destructuring-bind (X Y) g2
        (declare (type (or string fixnum) X)
                 (type number Y))
        :clause-3)))
     (case (pop g1)
      (t (return-from check nil)))
   s.4
     (unless g1 (return-from check
       (destructuring-bind (X Y) g2
        (declare (type fixnum X Y))
        :clause-1)))
     (case (pop g1)
      (t (return-from check nil)))
   s.5
     (unless g1 (return-from check
       (destructuring-bind (X Y) g2
        (declare (type fixnum X)
                 (type integer Y))
        :clause-2)))
     (case (pop g1)
      (t (return-from check nil))))))
```

Figure 14: Macro expansion of `rte-case` from Figure 2 and consequently of `destructuring-case` from Figure 1.

5 OPTIMIZED CODE GENERATION

Figure 14 shows the essential part of the final macro expansion of the `rte-case` shown in Figure 2. Each state in the DFA corresponds to a label within a `tagbody`, a conditional `unless` checking for end of sequence, and a `typecase` with one branch per transition in the DFA, including the implicit transition to ⊥. We have used `typecase` in this example output, but the reader may well notice that there are several occurrences of redundant type checks in the output. For example, the `typecase` at label `s.2` in Figure 14 contains multiple checks for `fixnum` and `integer`. We showed in [10] how these redundant type checks might be eliminated simply by replacing `typecase` with `bdd-typecase`.

6 PREVIOUS WORK

Attempts to implement `destructuring-case` are numerous. We mention three here. R7RS Scheme provides `case-lambda` [14, Section 4.2.9], allowing fixed length argument lists, but lacking any sort of destructuring; the implementation of `destructuring-case` provided in [3] is missing tree-structure-based clause selection; the implementation provided in [4], provides tree-structure-based clause selection, but not within the `&optional` nor `&key` portion. In none of these cases does the clause selection consider the types of the objects within the list being destructured.

Manuel and Ramanujam [8] introduce automata over infinite alphabets, which seems to be an interesting theoretical approach of viewing DFA whose transitions are Common Lisp types. Manuel and Ramanujam do not investigate questions of construction and simplification as we have investigated in our approach.

6.1 Conclusion and Perspectives

The simplification algorithm described in Section 4.2 may not guarantee a minimum result. For example, reconsider Φ_1 in Figure 12 (bottom). Suppose $T_3 = T' \cup T''$, and suppose there exists $s \in \mathbb{S}$ such that $\Phi(s) = \{(T', \{6,7\}),\ T'', \{6,7\}\}$. In such a case, states 2 and s would be indistinguishable, but not mergable with the simplfication algorithm we have described. More research is needed to determine whether such a case can occur, and what the most general form is. Such analysis is necessary to accomplish our goal of generalizing finite automata theory on finite alphabets to handle infinite alphabets representable as disjoinable types.

In the procedure described in Section 4, we constructed the SXP starting with DFAs which were sub-optimal. The DFAs shown in Figure 9 have states which are not coäccessible: states 2.2 and 3.4. Furthermore, one of the DFAs has states 3.2 and 3.5 which are indistinguishable. If we choose to trim and simplify the input DFAs before constructing the SXP there seem to be cases where we reduce the number of state pairs which need to be visited.

A natural question is whether it is better to simplify the input DFAs before computing the SXP, simplify after, or both. One might be tempted to claim that we should always simplify DFAs before computing the SXP. However, we do not currently have enough data to confidently support this claim.

We also discussed in Section 3.3 a technique for making the DFAs match non-overlapping languages before attempting to calculate the SXP. This technique avoids having to make priority based decisions when the languages overlap. We thereafter saw that this technique produces DFAs with non-coäccessible states. It may well be worth investigation whether robustly implementing the priority based SXP procedure is more efficient, as the input DFAs would themselves be smaller in many cases, and be absent the non-coäccessible states.

The `rte-case` macro we discuss in this paper does not attempt to answer questions about exhaustiveness. It is possible however, to enhance the `rte-case` macro with `rte-ecase` (exhaustive `rte-case`) which would append a final *otherwise* clause, `(:* t)`. This clause would serve at compile time to detect whether the leading clauses are exhaustive; for if no state in the DFA corresponds to this `:otherwise-clause`, then the given rte patterns are exhaustive. However, if there is a path in the DFA from an initial state to the `:otherwise-clause`, then the type labels on such a path form a type signature for such a counter example. The types of the elements of such a counter-example sequence could easily be generated by finding any transit through the DFA, and clipping away any loops it contains. The macro might also produce a compiler warning, as well as insert a call to error in the code in case the code path is taken at run-time.

REFERENCES

[1] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[2] Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, October 1964. ISSN 0004-5411. doi: 10.1145/321239.321249. URL http://doi.acm.org/10.1145/321239.321249.

[3] Public Domain. Alexandria implementation of destructuring-case. URL https://common-lisp.net/project/alexandria/draft/alexandria.html.

[4] Nobuhiko Funato. Public domain implementation of destructuring-bind, 2013. URL https://gist.github.com/nfunato/6247751. accessed 14 October 2018, 12h36 +0200.

[5] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991. ISSN 0001-0782.

[6] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

[7] Sonja E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley, 1989. ISBN 0-20117-589-4.

[8] Amaldev Manuel and Ramaswamy Ramanujam. Automata over infinite alphabets. In *Modern Applications of Automata Theory*, pages 529–554. 2012. doi: 10.1142/9789814271059_0017. URL https://doi.org/10.1142/9789814271059_0017.

[9] Jim Newton. *Representing and Computing with Types in Dynamically Typed Languages*. PhD thesis, Sorbonne University, November 2018.

[10] Jim Newton and Didier Verna. Strategies for typecase optimization. In *European Lisp Symposium*, Marbella, Spain, April 2018.

[11] Jim Newton, Akim Demaille, and Didier Verna. Type-Checking of Heterogeneous Sequences in Common Lisp. In *European Lisp Symposium*, Kraków, Poland, May 2016.

[12] Jim Newton, Didier Verna, and Maximilien Colange. Programmatic Manipulation of Common Lisp Type Specifiers. In *European Lisp Symposium*, Brussels, Belgium, April 2017.

[13] Scott Owens, John Reppy, and Aaron Turon. Regular-expression Derivatives Re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009. ISSN 0956-7968. doi: 10.1017/S0956796808007090. URL http://dx.doi.org/10.1017/S0956796808007090.

[14] Alex Shinn, John Cowan, and Arthur A. Gleckler. Revised 7 report on the algorithmic language Scheme. Technical report, 2013.

[15] Francois Yvon and Akim Demaille. *Théorie des Langages Rationnels*. EPITA LRDE, 2014. URL http://www.lrde.epita.fr/~akim/thlr/lecture-notes/theorie-des-langages-rationnels.pdf. Lecture notes.

Lazy, parallel multiple value reductions in Common Lisp

Marco Heisig
FAU Erlangen-Nürnberg
Cauerstraße 11
Erlangen 91058, Germany
marco.heisig@fau.de

ABSTRACT

Reductions, folds, or catamorphisms are an important component of every functional programmer's toolbox. However, common manifestations of these operators can only operate on a single sequence at once and don't have any potential for parallel execution.

We present a new, parallelizable reduction operator that can simultaneously reduce k arrays at once, using a function with $2k$ arguments and k return values. We then discuss an efficient implementation of this new reduction operator as part of the Petalisp project.

CCS CONCEPTS

•Software and its engineering →Functional languages; Data flow languages; Parallel programming languages; *Just-in-time compilers;*

KEYWORDS

Common Lisp, Reductions, Lazy Evaluation, Parallelism

ACM Reference format:
Marco Heisig. 2019. Lazy, parallel multiple value reductions in Common Lisp. In *Proceedings of the 12th European Lisp Symposium, Genova, Italy, April 01–02 2019 (ELS'19),* 6 pages.
DOI: 10.5281/zenodo.2642164

1 INTRODUCTION

The reduction is one of the most versatile tools of the functional programmer. Figure 1 defines the exemplary reduction operator `fold`. Despite its simplicity, it captures the essence of what we find in the standard libraries of Scheme, Haskell, Common Lisp and many other programming languages: recursive processing of a data structure and combination of values with a binary function.

```
(defun fold (f z l)
  (if (null l)
      z
      (fold f (funcall f (first l) z) (rest l))))
```

Figure 1: A simple reduction operator: `fold`.

The simple four line function in figure 1 is quite powerful, as long as we confine ourselves to the domain of lists. Depending on the supplied binary function and initial value, we can express a variety of concepts:

- **sum**
  `(fold #'+ 0 numbers)`
- **product**
  `(fold #'* 1 numbers)`
- **maximum**
  `(fold #'max 0 non-negative-numbers)`
- **reversal**
  `(fold #'cons '() list)`
- **filtering**
  ```
  (fold (lambda (i j) (if (oddp i) (cons i j) j))
        '() list)
  ```

The good news is that due to its tail-recursive structure, our `fold` function can be run very efficiently on a serial processor. The bad news, however, is that this function is completely unsuited for execution on parallel hardware. This gets apparent if we visualize the data flow of a particular call, as in figure 2.

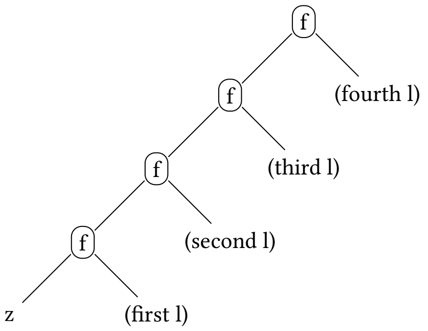


Figure 2: Data-flow graph of a call to fold.

A call to `fold` results in a dependency chain that has the same length as the list being worked on and, consequentially, there is zero potential for parallel execution. In a world of ubiquitous multicore processors, this is a huge and growing problem. Or, as Guy Steele put it, "foldl and foldr Considered Slightly Harmful"[9].

In this paper, we present a reduction operator that has inherent parallelism, addresses simultaneous reduction of multiple sequences and reductions of multi-dimensional arrays.

ELS'19, Genova, Italy

DOI: 10.5281/zenodo.2642164

2 PETALISP

This work has been developed as part of the Petalisp project[5, 6]. The goal of Petalisp is to provide a model for data parallel programming that fits nicely into the existing general purpose programming language Common Lisp. Petalisp programs are composed of only a tiny number of core operators — parallel map, parallel reduce, affine-linear data motion and data fusion — and only a single data structure — the lazy array. Lazy arrays can be evaluated, i.e., turned into Lisp arrays with the function `compute`.

As a convenience feature, Petalisp functions implicitly convert arguments that are regular Lisp arrays to lazy arrays, and converts all other arguments to lazy arrays of rank zero. Because of this implicit conversion, most of the discussion in this paper can be carried out without worrying about Petalisp at all, apart from inserting an occasional call to `compute` for explicit evaluation.

The minimalist set of features in Petalisp unlocks a lot of powerful optimizations, but means that its programs are fundamentally limited by the semantics of its core operators. That is one of the reasons why we put so much effort into the definition of parallel reduction.

3 RELATED WORK

We do not explicitly list each and every kind of reduction throughout the computer science landscape, but report only operators that have either inherent parallelism, or special support for handling multiple values.

- The Scheme languages includes a variety of operators for folding and reducing list elements in SRFI-1[8]. Many of them can operate on multiple lists at once, in which case the combining function will receive one argument for the accumulating value and one argument per list. However, no special provisions exist for reducing multiple values at once, and none of the functions is inherently parallel.
- The parallel programming language NESL[1] permits hierarchical reduction similar to ours, since its language core supports nested parallel constructs. However, the advantage of our technique is that it is embedded into the powerful general purpose language Common Lisp instead of being a standalone tool. Furthermore, we are not aware that NESL has capabilities for handling multiple values.
- The MPI standard[4] for distributed programming includes primitives for reducing potentially distributed data in parallel. It also permits reductions with user-defined functions and, with the right annotations, changes to the order of evaluation. But all MPI reductions are limited to reducing a single array with a binary function.
- The programming model of MapReduce[2] is similar to the model provided by Petalisp. Here, processes are split into a *Mapper* function to express embarrassingly parallel tasks and a *Reducer* function describe how data is to be accumulated. While MapReduce has excellent support for parallelism, each *Reducer* function takes only a single key-value pair and therefore suffers from the same limitations as most other reductions.
- Connection Machine Lisp[10] features a syntactic construct for parallel, unordered reduction, named β. This construct has not only influenced the design of our reduction operator, it is also the origin of our operator's name.

4 OUR TECHNIQUE

What follows is the definition of our reduction operator β.

function **β** *f* *array* &rest *more-arrays* → *result**

The supplied function f must accept $2k$ arguments and return k values, where k is the number of supplied arrays in *array* and *more-arrays*. All supplied arrays must have the same shape S, which is the cartesian product of some ranges, i.e., $S = r_1 \times \ldots \times r_n$, where each range r_k is a set of integers, e.g., $\{0, 1, \ldots, m\}$. Then β returns k arrays of shape $s = r_2 \times \ldots \times r_n$, whose elements are a combination of the elements along the first axis of each array according to the following rules:

(1) If the given arrays are empty, signal an error.
(2) If the first axis of each given array contains exactly one element, drop that axis and return arrays with the same content, but with shape s.
(3) If the first axis of each given array contains more than one element, partition the indices of this axis into a lower half l and an upper half u, such that $r_1 = l \cup u$ and such that either $|l| = |u|$, or $|l| = |u| + 1$. Then split each given array into a part with shape $l \times s$ and a part with shape $u \times s$. Recursively process the lower and the upper halves of each array independently to obtain $2k$ new arrays of shape s. Finally, combine these $2k$ arrays element-wise with f to obtain k new arrays with all values returned by f. Return these arrays.

4.1 A Simple Example

Let us illustrate this definition with a simple example. If we apply β to a binary function f and a vector with four elements, we start out with $k = 1$ and the shape $S = \{(0), (1), (2), (3)\}$. We are dealing with a rank one array, so the result will be a rank zero array.

Since the given array is neither empty, nor has just a single element, we start out with an application of rule 3. That means we split the given array into a lower half with indices $\{(0), (1)\}$ and an upper half with indices $\{(2), (3)\}$ and process each half recursively. The lower half is again subject to rule 3 and split into a part with the sole index $\{(0)\}$ and one with the sole index $\{(1)\}$. Further recursive processing of each of these one-element arrays results in an application of rule 2, where the given rank one array are turned into equivalent rank zero array. These rank zero arrays are returned to the previous application of rule 3, where their sole elements are combined with the function f to form the content of the rank zero array that is the value of the lower half. The upper half is processed analogously. Finally, these arrays are combined by yet another application of the function f to obtain the final result. Figure 3 illustrates this process.

4.2 Discussion

We will now motivate and justify the individual design decisions we made when defining β.

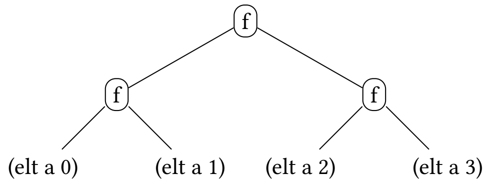


Figure 3: Data-flow graph of a call to β on a four element vector.

Handling of Empty Arrays. It is common for reduction operators to take an explicit initial element that is returned when processing an empty sequence. The function β, however, simply signals an error for this case. This behavior is best explained by looking at the two cases how initial elements are typically used:

In the first case, the initial element has the same type as the elements of the sequence and both arguments of the combining function therefore have the same type. In this case, it is sometimes possible to find a suitable initial element, e.g., zero for addition. But in general there is no such element. A prominent example for this general case is computing the maximum of a sequence of integers. The only sane initial element would be $-\infty$, which is not an integer.

In the second case, the initial element has a different type than the elements of the sequence and, consequently, the arguments of the combining function must be heterogeneous. In essence, the initial element acts as an accumulator that is threaded through all sequence elements sequentially. This would directly conflict with our goal to allow parallel execution. Luckily, there is a suitable equivalent for accumulation in our β operator, which is to convert the array of values to an array of accumulators first, and to define the combining function such that it merges given accumulators[1]. The difference can be seen in figure 4, where we use lists as accumulators and the function append as combining function. The function α that we use in this figure is the lazy, parallel mapping operator of Petalisp. Its semantics is similar to that of `cl:map`.

```
(defparameter *a* #(1 2 3 4 5 6))

;; inherently serial
(reduce #'cons *a* :initial-value '() :from-end t)

;; parallel alternative
(compute (β #'append  (α #'list *a*)))
```

Figure 4: Two ways to convert an array to a list.

The Name β. Some readers might frown at the decision to use a greek letter as a function name. One frequent complaint is that this renders any code using this function non-portable. The other frequent complaint is that modern keyboards offer no easy way to insert Greek letters. To the first complaint, we reply that parallel programming is already outside of the scope of the Common Lisp specification, so the concern would only apply to the hypothetical implementations that support concurrency, but not Unicode. To the second complaint, we reply that there is plenty of IDE support for inserting nonstandard characters, and that parallel programming is so hard that the time spent typing should be negligible in contrast to the time spent thinking.

Rank Zero Arrays Instead of Scalars. By definition, or function β can never return scalar values, only arrays of rank zero. This could be an annoyance for the simple, frequent case of reducing vectors. The solution we developed here is that Petalisp treats scalars and arrays of rank zero interchangeably. And, most importantly, when calling `compute` to trigger explicit evaluation, all arrays of rank zero are automatically converted to scalars.

Choice of Axis. According to our definition, reductions apply only to the first axis of a given array. Another option would have been to reduce along a specified axis and to reduce the entire array if no axis is explicitly specified. While this would add some convenience for the user, it would make β less orthogonal to the existing set of Petalisp primitives. The choice of axis can already be achieved by permuting an array's indices, which is already supported by another Petalisp primitive. And the reduction of an entire array with rank n can be emulated by n successive reductions. Figure 5 illustrates these techniques.

Subdivision Strategy. The current subdivision strategy is to split the first axis of an array into two equal halves, until the axis has been reduced to a single index. As it can be seen in figure 3, the effect is that all array elements are effectively combined along the nodes of a binary tree. One might wonder whether a different or more flexible subdivision strategy could be more efficient, but we decided not to pursue this thought further until we have an excellent implementation of reduction on a binary tree.

One argument in favor of this reduction order is that it produces deterministic results. Since all Petalisp programs are just a combination of core operators, and all other core operators are already deterministic, we gain the property that all Petalisp programs are fully deterministic — a very desirable property for a parallel programming language.

Arrays Only. Our definition of β requires that all its arguments but the combining function are arrays (or, lazy arrays) and not, e.g., arbitrary sequences. The reason for this is that efficient execution requires that both the shape and the elements of each argument must be accessible in $O(1)$ time. However, it is conceivable that future versions of Petalisp will also support lazy arrays whose backing storage is a user defined sequence with fast random access, e.g., as proposed by Rhodes[7].

Multiple Values. Coming from statically typed functional languages, one might wonder why we bother with functions returning multiple values, when we just could have used tuples and let the compiler optimize them away. One reason is that this way, even code without sufficient static type information can be run efficiently and without additional consing. The other reason is that we would have had to introduce static, immutable tuples into Common Lisp

[1] Some readers might worry about the cost of creating an array of accumulators first. But thanks to lazy evaluation, Petalisp can eliminate this temporary array and move the creation of the accumulator directly into the reduction.

and force every user of our reduction operator to use them, whereas multiple values are already part of the language.

5 EXAMPLES

In the following section, we show how our reduction operator can be used in practice.

5.1 Numeric Accumulation

In this first example, in figure 5, we show how β can be used to express the sum and product of some numbers. This illustrates also how some seeming deficiencies like lack of an initial value can be overcome with a suitable abstraction. In this case, we call this abstraction β*. It correctly handles the case of receiving an empty array, and, for further convenience, reduces the whole array if no explicit axis is supplied.

```
(defun β* (f z x &optional axis)
  (cond ((empty-array-p x)
         z)
        ((integerp axis)
         (β f (exchange-axes x 0 axis)))
        ((loop until (zerop (rank x))
               do (setf x (β f x))
               finally (return x)))))

(defun sum (x &optional axis)
  (β* #'+ 0 x axis))

(defun product (x &optional axis)
  (β* #'* 1 x axis))
```

Figure 5: Using β for numeric accumulation.

5.2 Computing the Maximum and its Index

In this second example, in figure 6, we show how to perform a reduction on multiple values. Our goal is to efficiently obtain both the maximum of a vector, and the corresponding index. To do so, the function `max*`, supplies two arrays to β — the array itself, and an array of the same shape containing the indices of the axis zero corresponding to each array element. These two arrays are then reduced with a four argument function that forwards either the two left arguments or the two right arguments, depending on which side yields the larger value.

As discussed later in section 6, the function `max*` is probably more efficient than a programmer would assume by looking at its definition. Not only can large parts of it be run in parallel, it is also possible to completely eliminate the lazy array that is given as the last argument to β, by computing its elements into a function of the currently processed index.

```
(defun max* (x)
  (β (lambda (lv li rv ri)
       (if (> lv rv)
           (values lv li)
           (values rv ri)))
     x (indices x 0)))
```

Figure 6: Computing the maximum element and its index.

6 IMPLEMENTATION

In figure 7, we show a naïve implementation of β. The only *user-visible* difference between this naïve code and the one Petalisp actually uses is that there is no implicit broadcasting of argument arrays, no error handling, and no support for lazy arrays. But what this code shows is that a naïve implementation will always be prohibitively slow. Neither the dimensions of the given arrays, nor their rank, nor their element type are known at compile time. Not even the number of arrays k is known statically. To tackle this problem regardless, we have to make use of higher-order functions and relatively expensive constructs like `multiple-value-list` and `(apply #'aref)`.

We see that the crucial question regarding an efficient implementation of our proposed β operator is how to deal with the large amount of compile time uncertainty. One possible approach would be to write or generate multiple versions of the code for each possible invocation, e.g., using the technique of Strandh[3]. The problem is that in our case, the space of possible arguments and types is extremely large. Assuming we wanted to create special versions just for the case of reducing up to three arrays with a rank below 3 and for every specialized array element type. Then, assuming an implementation with 20 specialized array types[2], we would end up with $2(20^1 + 20^2 + 20^3) = 16840$ different versions. Such an amount of specialization is unreasonable, even on a modern computer with plenty of memory.

What we do instead is that we generate specialized reduction functions on demand, using the existing framework of Petalisp. This has multiple advantages:

- Data flow analysis prevents unnecessary evaluation. If parts of the result of a reduction are not used, the corresponding inputs will also not be computed.
- If the inputs of a reduction are lazy arrays, the code that computes the contents of these arrays can often be inlined directly into the reduction, thus avoiding an unnecessary intermediate array.
- Specialized code is cached efficiently, such that future invocations with a similar signature can reuse the previously compiled code.
- The programmable type inference engine of Petalisp can often statically deduce the element type of the results of a reduction and allocate them in a suitable specialized array.
- The size of the argument arrays is known during code generation. This makes it possible to generate and use different variants, e.g., to avoid thread parallelization when the workload is known to be small.

[2] Having 20 specialized array types is a conservative estimate. SBCL on a 64bit architecture recognizes 34 subtypes of `array`, CCL even 38.

```
(defun β (f array &rest more-arrays)
 (let* ((arrays (list* array more-arrays))
        (k (length arrays))
        (r (array-dimension array 0))
        (dims (rest (array-dimensions array)))
        (results
          (loop repeat k
                collect (make-array dims))))
  (map-indices
   (lambda (indices)
    (mapcar
     (lambda (o v)
      (setf (apply #'aref o indices) v))
     results
     (multiple-value-list
      (labels
         ((divide-and-conquer (start end)
            (if (= start end)
                (values-list
                 (mapcar
                  (lambda (a)
                    (apply #'aref a
                           (list* end indices)))
                  arrays))
               (multiple-value-bind (ls le us ue)
                    (split-range start end)
                  (values-list
                   (subseq
                    (multiple-value-list
                     (multiple-value-call f
                      (divide-and-conquer ls le)
                    (divide-and-conquer us ue)))
                    0 k))))))
        (divide-and-conquer 0 (1- r))))))
   dims)
  (values-list results)))

(defun split-range (start end)
  (let ((mid (floor (+ start end) 2)))
    (values start mid (1+ mid) end)))

(defun map-indices (fn dims)
  (if (null dims)
      (funcall fn '())
      (apply #'alexandria:map-product
             (alexandria:compose fn #'list)
             (mapcar #'alexandria:iota dims))))
```

Figure 7: A possible implementation of β.

In order to support the new reductions, we also had to make several changes to the Petalisp internals. The most challenging task was to add support for functions returning multiple values. This change has profound implications, as it turns what used to be data-flow trees into directed acyclic data-flow graphs, and because it touches many optimization passes, such as common subexpression elimination and hoisting of loop invariant code.

We are happy to report that this transition is now complete, and that Petalisp now has full support for functions returning multiple values. These changes affect not only reductions, but also parallel mapping. It is now also possible to, e.g., map the function `floor` over a single array to obtain one array with all the quotients and one array with all the remainders.

To give a glimpse into the workings of our code generator, we show in figure 8 a part of the code generated during the first evaluation of a call to the function `max*` from figure 6 on a vector. This snipped of generated code shows how the number of values has been turned into a compile time constant, how the reference to the second array has been reduced to `(identity index)` and how the reference to the first array has been lowered to a call to `row-major-aref` with a simple offset.

```
...
(labels
  ((divide-and-conquer (min max)
     (declare (type fixnum min max))
     (if (= min max)
         (let ((index (+ min (* #:g3 #:g4))))
           (let* ((v (row-major-aref a0 index))
                  (i (identity index)))
             (values v i)))
       (let ((mid (+ min (floor (- max min) 2))))
           (multiple-value-call
               (lambda (l0 l1 r0 r1)
                 (multiple-value-bind (r0 r1)
                     (funcall f l0 l1 r0 r1)
                   (values r0 r1)))
             (divide-and-conquer min mid)
           (divide-and-conquer (1+ mid) max))))))
 (divide-and-conquer 0 (/ (- #:g5 #:g3) #:g4)))
...
```

Figure 8: An excerpt from the code generated for a call to `max*`.

7 PERFORMANCE

We have not yet implemented all optimizations that we envision, so it is too early for a detailed performance analysis. But we can already outline the most important performance characteristics of our technique. Our measurements show that for large vectors, our operator is about half as fast as SBCL'S built-in function `cl:reduce`, despite being vastly more general. The downside of on-demand code selection or generation is that it incurs a constant overhead of several microseconds, making it unsuitable for reductions of small arrays. However, we expect that we can lower this constant overhead in the future by switching to more efficient data structures and by caching some expensive intermediary steps.

8 CONCLUSIONS AND FUTURE WORK

We have presented a reduction operator β that is simple, powerful and has plenty of inherent parallelism. Most importantly, our reduction operator supports accumulation of multiple values at once. Thus, it greatly extends the range of programs that can be expressed as a single reduction. e.g., for finding both the minimum and maximum of a sequence, or for accurate summation of floating point numbers, using a second value to accumulate errors.

We have carefully presented our design considerations, especially with respect to inherent parallelism and simplicity. The value of simplicity is still underappreciated in modern parallel programming. We think that in order to obtain both correctness and speed, it is sometimes better to go for clean, robust approaches — such as reducing along a binary tree only — instead of chasing after the last few percent of performance.

As a second contribution, we have presented an implementation technique — on-demand compilation of specialized code at run time — that allows us to turn this operator into efficient, highly specialized code. To do so, we use the existing data-flow analysis and compiler infrastructure of Petalisp. All our code is freely available under a copyleft license[3].

This paper marks the end of the design process of the parallel programming library Petalisp. This doesn't mean we are finished with Petalisp development, but we will now focus exclusively on under-the-hood improvements, such as better thread-level parallelization, faster dispatch, SIMD vectorization and, ultimately, distributed parallelization.

9 ACKNOWLEDGMENTS

We would like to thank all the people in the `#petalisp` IRC channel.

REFERENCES

[1] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[3] Irène Durand and Robert Strandh. Fast, maintainable, and portable sequence functions. In *Proceedings of the 10th European Lisp Symposium*, ELS2017. European Lisp Scientific Activities Association, 2017.

[4] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1.* High Performance Computing Center Stuttgart (HLRS), 2015. URL https://www.mpi-forum.org/docs/.

[5] Marco Heisig. Petalisp: A common lisp library for data parallel programming. In *Proceedings of the 11th European Lisp Symposium on European Lisp Symposium*, ELS2018, pages 1:4–1:11. European Lisp Scientific Activities Association, 2018. ISBN 978-2-9557474-2-1.

[6] Marco Heisig and Harald Köstler. Petalisp: Run time code generation for operations on strided arrays. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2018, pages 11–17, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5852-1. doi: 10.1145/3219753.3219755. URL http://doi.acm.org/10.1145/3219753.3219755.

[7] Christophe Rhodes. User-extensible sequences in common lisp. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, pages 13:1–13:14, New York, NY, USA, 2009. ACM. ISBN 978-1-59593-618-9. doi: 10.1145/1622123.1622138. URL http://doi.acm.org/10.1145/1622123.1622138.

[8] Olin Shivers. Srfi-1: List library, 1998. URL https://srfi.schemers.org/srfi-1/srfi-1.html.

[9] Guy L. Steele, Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. *SIGPLAN Not.*, 44(9):1–2, August 2009. ISSN 0362-1340. doi: 10.1145/1631687.1596551. URL http://doi.acm.org/10.1145/1631687.1596551.

[10] Guy L. Steele, Jr. and W. Daniel Hillis. Connection machine lisp: Fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 279–297, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319870. URL http://doi.acm.org/10.1145/319838.319870.

[3] https://github.com/marcoheisig/Petalisp

Session IV: Code as Data

Monday, 1.4.2019

16:00–16:30	Mikhail Raskin and Christoph Welzel: Working with first-order proofs and provers
16:30–17:00	António Leitão: Plagiarism Detection for Lisp
17:00–17:30	Lightning talks
19:00	**Reception at Aula Magna**

Working with first-order proofs and provers

Michael Raskin
Christoph Welzel*
Technical University of Munich
Garching bei München
raskin@mccme.ru,welzel@in.tum.de

ABSTRACT

Verifying software correctness has always been an important and complicated task. Recently, formal proofs of critical properties of algorithms and even implementations are becoming practical. Currently, the most powerful automated proof search tools use first-order logic while popular interactive proof assistants use higher-order logic.

We present our work-in-progress set of tools that aim to eventually provide a usable first-order logic computer-assisted proof environment.

CCS CONCEPTS

• **Software and its engineering → Constraint and logic languages**; • **Theory of computation** → *Interactive proof systems*; *Automated reasoning*;

KEYWORDS

automated reasoning, computer-aided proofs, first-order logic, verification

ACM Reference Format:
Michael Raskin and Christoph Welzel. 2019. Working with first-order proofs and provers. In *Proceedings of the 12th European Lisp Symposium (ELS'19)*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.2633990

1 INTRODUCTION

Complete formal verification of algorithms and their implementations is becoming more widely applicable. Probably the most general approach is construction of formal proofs in a chosen theory. Interactively constructed formal proofs often use one of the popular higher order logics, such as Calculus of Coinductive Constructions in case of Coq[10] or the chosen higher-order logic of Isabelle/HOL[12]. At the same time, a lot of progress in automated reasoning is achieved in the field of first-order logic. For example, the Conference on Automated Deduction (CADE) has an automated theorem prover (ATP) competition, called CADE ATP System Competition (CASC)[4]. Satallax[13], the leading higher-order ATP according to the CASC results[5], uses E prover[6] (one of the leading first-order provers) for some tasks. On the other hand, CoqHammer[11], a tool that aims to partially automate interactive construction of proofs with Coq, also uses translation into first-order logic and multiple first-order automated provers.

We want to be able to verify statements about distributed algorithms where direct application of generic ATP systems might still be impractical. To that aim, we create first-order specifications, and use domain knowledge to write or generate proofs as sequences of lemmas, while automated theorem provers verify implications. As CASC competitions have popularized the unified input format of the Thousands of Problems for Theorem Provers (TPTP) collection[1], using multiple ATP systems does not require any changes in the proof format. To support this kind of exploration, we develop supporting tooling for managing the specification, preparing the list of lemmas, and interacting with the proof system.

While it is too early to draw any conclusions from our ongoing experiments with representing properties of distributed systems in the first-order logic, we want to present the supporting tooling used in this research.

2 OUR TOOLING

2.1 Data formats

All of our tooling uses TPTP for all the output and most of the proof input. We use the SyntaxBNF file from the TPTP distribution (Backus–Naur form of the TPTP format definition) and translate it into Esrap[7] rules to parse the format. It turns out that unambiguity of the official TPTP BNF specification allows us to order the parsing rules in a way compatible with packrat parsing[8]. More specifically, in every alternation rule the first (after reordering) successful option can be taken.

However, the formal specifications of the systems in question contain large amounts of similar statements. These specifications are generated programmatically. Currently we do not want to make lasting decisions about the structure of the specifications we will work with, so the generating code is written in Common Lisp and refactored according to the current specification in question.

The proof itself contains additional definitions and lemmas, and various instructions such as advice to prove some lemma by case analysis (with a list of cases provided). We use TPTP Process Instructions (TPI) extension[2] of the TPTP syntax

*The project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 787367

ELS'19, April 01–02 2019, Genova, Italy

ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2633990

to encode the additional imperative instructions related to lemma list processing.

2.2 Global workflow

First of all, we need to generate the axioms describing our formal specification.

To validate that the axioms describe the intended model, we generate a test run by evaluating the transition rules. We have code that can evaluate a first-order formula on an incomplete model, if the fixed part of the model is enough to determine the formula value easily. The generated runs are validated in two ways: by manual inspection, and by verifying that an automated theorem prover given this run and the full specification does not find a contradiction in reasonable time.

At our current stage of exploration, the next step involves writing a list of lemmas (and instructions for their preprocessing) that should be sufficient to prove the desired condition.

The last step is verifying that a list of lemmas constitutes a correct proof. Of course, in practice this step is performed in parallel with the previous one. A part of verification is performed, then the proof is updated to avoid the problems observed during verification attempt. The verification attempts are usually started inside the part of the proof currently of interest, and stop when some lemma cannot be proved.

Even after the last step of proof verification our tooling can offer some further support. We have some utilities for analysing and visualising the output of an ATP system.

The system currently does not provide any dedicated user interface. It can be used either from a Common Lisp REPL, or via wrapper shell scripts invoking necessary operations.

2.3 Structure of an example model

The examples will be related to one possible encoding of the Dijkstra's mutual exclusion protocol[9] executed on a single CPU with multiple time-sharing processes (as illustrated in Algorithm 1). In general, this model has a set of agents switching between states, and local variables. We automatise a linearly ordered discrete time model which uses an initial value *initial* and a function $next\_moment(T)$ which "advances" the time one step. The state of agent A at moment T is represented by the function value $active\_state(T, A)$. There are also some other per-agent variables (and a global *turn* variable), modelled in the same way, e.g. $counter(T, A)$ which gives the value of the variable *counter* of agent A at time T. For a single-CPU multi-process execution we can assume that only one agent at a time can change its state or variables, and denote this agent as $active\_agent(T)$. We want to avoid a situation where two agents execute the critical section (i.e. have the active state equal to *criticalSection* which represents line 17 in Algorithm 1) at the same time.

The safety-critical part of the Dijkstra's mutual exclusion protocol consists of an agent declaring its intent to enter the critical section, and checking that no other agent has also declared the same intention.

```
1  begin
2      $Stealable_i \longleftarrow false$;
3      if $turn \neq i$ then
4          $Outside_i \longleftarrow true$;
5          if $Stealable_{turn} = true$ then
6              $turn \longleftarrow i$;
7          end
8          go to 3;
9      end
10     else
11         $Outside_i \longleftarrow false$;
12         for $counter_i \longleftarrow 1$ to $n$ do
13             if $counter_i = i$ then continue;
14             if $Outside_{counter_i} = false$ then go to 3;
15         end
16     end
17     <critical section>;
18     $Outside_i \longleftarrow true$;
19     $Stealable_i \longleftarrow true$;
20     <remainder of cycle>;
21     go to 2;
22 end
```

Algorithm 1: Dijkstra's algorithm for process i with n parallel processes.

To avoid encoding a full theory with induction, one can start with proving just the inductive step: define an invariant then prove that this invariant at some moment implies the same invariant at the next moment of time, and that the invariant implies safety. The reason to delay encoding the full proof by induction is that the most natural ways to encode induction axiomatically require an infinite number of axioms. This is often called "induction axiom schema" — for every formula expressing a predicate, there is an axiom. This axiom claims that proving the base case and the induction step for the property in question is enough to verify the property for all natural numbers.

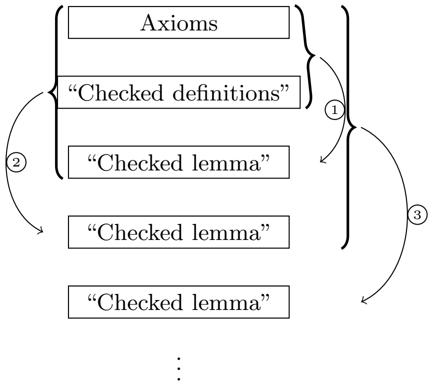
2.4 Representation of the proof

In the TPTP format each statement is given a role; we parse the list of statements and look at their roles. Axioms are introduced in the specification, and can be used directly.

"Checked definitions" can be introduced in the proof; they are axioms that introduce simple abbreviations, extending the theory in a conservative way. We verify that they define a single name in terms of previously seen names, and use these definitions as axioms. For example, we can define the safety condition.

```
fof(define_safety_for, checked_definition,
     ![T,A1,A2]: (safe_for(T,A1,A2)<=>(
        (active_state(T,A1)=criticalSection
          & active_state(T,A2)=criticalSection)
        => A1=A2))).
```

Figure 1: Structure and proof-steps of our tool.



This formula, which can be also rewritten in the usual notation as $\forall T, A_1, A_2 : (safe\_for(T, A_1, A_2) \Leftrightarrow ((state(T, A_1) = state(T, A_2) = \mathit{criticalSection}) \Rightarrow A_1 = A_2))$, says that a moment T is safe for a pair of agents A_1 and A_2 if either at least one of the agents is out of the critical section or they actually are the same agent. This reflects a part of the desired property that two different agents should not execute the critical section simultaneously. The full safety condition is defined by requiring this property to hold for each pair of agents.

"Checked lemmas" constitute the main part of the proof. Every such lemma is first given to an automated prover as a conjecture to prove, using the axioms and previous lemmas. If the prover reports a success, the lemma can be used as an axiom during the following steps. These steps are illustrated in Figure 1.

For example, the following lemma is proved as a part of a case by case analysis.

```
fof(safety_conditions_local_cases, checked_lemma,
  ![T,A,B]:( ~passed(T,A,B)
    | (passed(T,A,B) & ~passed(T,B,A))
    | (passed(T,A,B) & passed(T,B,A)))).
```

This lemma uses the predicate *passed*, that is defined to mean that agent A has declared its intent to enter the critical section and has already checked that agent B had not declared such at intent at the time of the check. The lemma itself is trivial, claiming only that at any given moment either A has not yet passed B, or A has passed B but B has not passed A, or both agents have passed each other.

We also support defining a limited set of axioms (and/or previously proved lemmas) to use when proving a specific lemma. This reduces the proof search space and therefore drastically improves the performance. There are cases where specifying the proof dependencies manually is easy; in addition, our lemma generation strategies include generation of such dependency hints where appropriate.

2.5 Additional proof-handling capabilities

We use TPI (TPTP Process Instructions) to specify operations on lemmas inside the proof. To improve interactive usability, we allow a special declaration that declares valid all the checked lemmas earlier in the proof. This can be convenient to skip a part of the proof that has already been verified earlier, or just to focus on a step in the middle of the proof before spending time on a possible unsuitable beginning.

In many cases, lemmas needed to achieve good performance of the proof search are predictable. Some of the techniques described in [3] are broadly applicable, especially proving all the components of each conjunction separately. Another important source of lemmas is case-by-case analysis, which requires choosing the cases but becomes a purely mechanical task afterwards.

For example, consider the following case. The lemma under consideration claims that if it is impossible for two agents to have passed each other, and two agents are distinct, and reaching the critical section requires passing all the other agents, then the two agents cannot both be in the critical section. It is easier to prove the conclusion if we know whether some agent hasn't passed the other one (in which case we can say it has not reached the critical section), or both agents have passed each other (in which case we obtain a contradiction with impossibility of mutual passing). So we prove exhaustiveness of a list of possible situations, and prove the lemma in each of them before proving it in the general case.

```
fof(safety_conditions_local_cases, checked_lemma,
  ![T,A,B]:( ~passed(T,A,B)
    | (passed(T,A,B) & ~passed(T,B,A))
    | (passed(T,A,B) & passed(T,B,A)))).

fof(safety_conditions_local_simplified,
  checked_lemma,
  ![T,A,B]:
  ((passed_exclusive_for(T,A,B) & A!=B
      & passed_in_critical_for(T,A)
      & passed_in_critical_for(T,B)) =>
      (active_state(T,A)!=criticalSection
      | active_state(T,B)!=criticalSection))).

tpi(ca_safety_conditions_local, add_cases,
  safety_conditions_local_cases =>
    safety_conditions_local_simplified).
```

It will be checked that the case enumeration is exhaustive, and then the main lemma will be checked with each of the cases added as an additional assumptions, e.g.

```
fof(ca_safety_conditions_local_simplified_case_...,
  checked_lemma,
  ![T,A,B]:
  ((passed_exclusive_for(T,A,B) & A!=B
      & passed_in_critical_for(T,A)
      & passed_in_critical_for(T,B)
      & ~passed(T,A,B)) =>
      (active_state(T,A)!=criticalSection
    | active_state(T,B)!=criticalSection))).
```

One more tool which sometimes unexpectedly turns out to be useful is definition expansion. We have an instruction that expands specified definitions in a given formula. It turns out that there are formulas that are simpler for existing provers if some definitions are expanded. Implementation of this functionality translates the definitions to expand into code performing the expansion and uses the run-time code evaluation and compilation capabilities provided by Common Lisp to run this code.

2.6 Processing the prover output

We have some tools for processing the output of automated theorem provers. If a prover has produced a proof in the TPTP format, it can be translated into Graphviz (an automated graph layout tool) or VUE (Visual Understanding Environment, a GUI tool which includes functionality convenient for working with some types of graphs) format for visualization, Visualization is supported both for the details of an individual lemma proof, and for an overview of the global lemma dependence.

We also have a tool for detection of unused lemmas. Unfortunately, sometimes removing unused lemmas from a proof makes the task much harder for some of the provers. The most likely reason for that is that even eventually unused axioms affect the prioritization of possible directions of proof search.

2.7 Parallel processing considerations

Following the naive interpretation of upstream TPI semantics, the operations on proofs are defined in imperative terms and operate on the entire proof. This currently limits the opportunities for parallel execution of the lemma-preprocessing code. On the other hand, invoking external ATP systems provides an isolated task to each prover instance, and to aggregate the results we just need to check that every instance has printed a line signalling successful proof. We currently work with proofs in an interactive mode, observing the proof verification progress step by step. To verify non-interactively a complete generated proof our tooling allows to export all the prover tasks, so that the prover invocations can be scheduled in any desired way (possibly with multiple computers involved in processing). We do not use this mode of operation yet.

3 CONCLUSION

We present a set of proof-manipulation tools that already implements quite a few useful operations and will further grow in parallel with the research they support.

We currently use the tool set described in the present paper to explore the performance implications of using different representations and using different provers for distributed algorithms. For example, we have encoded the inductive step proof for safety of Dijkstra's mutual exclusion algorithm. We plan to develop the capabilities further, supporting both computer-assisted proof construction and providing an intermediate representation for automated verification via proof generation. We hope that the approach and some parts of our code might be of use to others. A mirror of the code is available at https://gitlab.common-lisp.net/mraskin/gen-fof-proof/.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their comments and suggestions about presentation.

REFERENCES

[1] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0 Journal of Automated Reasoning, 2017, vol. 59, № 4, pp. 483–502.
[2] Geoff Sutcliffe. The TPTP Process Instruction (TPI) Language. Retrieved on 18 March 2019. http://tptp.cs.miami.edu/~tptp/TPTP/Proposals/TPILanguage.html
[3] Ewen Denney, Bernd Fischer, Johann Schumann. Using Automated Theorem Provers to Certify Auto-generated Aerospace Software. International Joint Conference on Automated Reasoning 2004.
[4] Geoff Sutcliffe. The CADE ATP System Competition - CASC. AI Magazine, 2016, vol. 37, № 2, pp. 99–101.
[5] The CADE ATP System Competition — The World Championship for Automated Theorem Proving, homepage. Retrieved on 18 March 2019. http://tptp.cs.miami.edu/~tptp/CASC/
[6] Stephan Schulz. System Description: E 1.8. Proc. of the 19th LPAR, Stellenbosch, 2013, pp. 735-743.
[7] Esrap project homepage. Retrieved on 28 January 2019. http://nikodemus.github.com/esrap/
[8] Bryan Ford, Packrat Parsing: a Practical Linear Time Algorithm with Backtracking. 2002. Retrieved on 28 January 2019. http://pdos.csail.mit.edu/~baford/packrat/thesis/
[9] E. W. Dijkstra. Solution of a problem in concurrent programming control. Commun. ACM 8, 9 (September 1965).
[10] Coq Development Team. The Coq Proof Assistant Reference Manual, version 8.7. Retrieved on 27 January 2019. http://coq.inria.fr
[11] L. Czajka and C. Kaliszyk. Hammer for Coq: Automation for Dependent Type Theory. Journal of Automated Reasoning, 2018, vol. 61, issue 1–4, pp. 423–453. http://cl-informatik.uibk.ac.at/cek/coqhammer/coqhammer.pdf
[12] Tobias Nipkow, Lawrence C Paulson, Markus Wenzel. Isabelle/HOL - A Proof Assistant for Higher-Order Logic Lecture Notes in Computer Science (2002).
[13] Chad E. Brown. Satallax: An Automatic Higher-Order Prover
[14] Graphviz project homepage. Retrieved on 18 March 2019. https://graphviz.org/
[15] Visual Understanding Environment project homepage. Retrieved on 18 March 2019. https://vue.tufts.edu/

Plagiarism Detection for Lisp

António Menezes Leitão
antonio.menezes.leitao@tecnico.ulisboa.pt
INESC-ID/Instituto Superior Técnico, University of Lisbon
Lisboa, Portugal

ABSTRACT

Computers made it very easy to copy someone else's work. This makes grading a difficult task, as the teacher that wants to prevent plagiarism needs to compare each student's assignment against every other student's assignment, a quadratic process that is impractical when the number of assignments gets large. Students know this and some take advantage of it. To be able to detect plagiarism among students' programming assignment we created a software tool that checks all assignments against each other, searching for copied fragments. Unlike many other tools, this search is not based on textual comparisons or hashing functions but, instead, on collecting pieces of evidence for and against a plagiarism verdict. This collection is determined by specialized procedures, invoked in a data-driven fashion, that incorporate expert knowledge regarding what is plagiarism and what is not plagiarism. The tool has been successfully used since 1995 in the evaluation of assignments programmed in Lisp dialects, particularly, Common Lisp, Scheme, Racket, and AutoLisp, and its mere existence became a deterrent for plagiarism.

ACM Reference Format:
António Menezes Leitão. 2019. Plagiarism Detection for Lisp. In *Proceedings of the 12th European Lisp Symposium (ELS'19).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

Nowadays, every student uses a computer to accomplish his assignments. Unfortunately, the computer also makes it very easy to copy and share students' work. The computer even helps in camouflaging the copied parts so that the teacher does not easily detect them. This situation is called plagiarism [15] and it is based on making a series of systematic changes to a program to create a derivative work that is syntactically different but semantically identical. The problem is especially serious in Computer Science courses, where evaluation typically includes the development of software projects. In this case, it is very easy to make copies that, at first sight, look very different from the original. Besides, when the assignments to be graded are divided among teachers, it is almost impossible to detect the copies.

In this paper, we discuss a set of techniques for the detection of copies in students' projects and we present a software tool that implements those techniques for Lisp-based projects. The tool was evaluated in real cases of projects developed using Common Lisp, Scheme, Racket, and AutoLisp and the results confirm its ability to detect plagiarism, even when the projects suffered considerable changes in order to hide their non-originality.

In the next section, we discuss the detection of plagiarism. In the following sections, we present an approach for its automation and specific details of its implementation in a plagiarism-detection tool. In the Results section, we evaluate its use in a real case. The final section summarizes the work and compares it with other approaches.

2 DETECTING PLAGIARISM

The intent of a plagiarized work is to obtain a grade without being associated with the original work from which it was derived. This implies that a carefully plagiarized work presents sufficient differences from the original that the teacher cannot easily correlate them. Unfortunately, seldom there is time to make a really good plagiarized work and some traces of the original remain in the copy. This is particularly true in software, where the copy needs to preserve the semantics of the original algorithms. In this case, the copier can change some particular parts of the original but many other parts such as the global structure, the control structures, the recursive or iterative nature of some functions, or the primitive operations used, remain as traces of the original. It is these traces that help identify plagiarism.

We now describe the principles behind the proposed plagiarism detection tool. The tool is intended to detect plagiarism between programs written in Lisp dialects, namely, Common Lisp, Scheme, Racket, and AutoLisp. In all of these languages, a program can be subdivided into smaller and relatively independent parts. Depending on the particular programming language, these parts are named *classes*, *functions*, *procedures*, *structures*, *methods*, etc. In most high-level languages, and even more so, in Lisp dialects, there are few limitations upon the textual form of the code and so it is possible to interchange parts of the program as well as change its textual indentation without affecting its behavior. Additionally, the program is, to a large extent, independent of the particular names used to describe it. This means that two programs may be textually different but semantically identical.

Usually, the visual detection of copies in programs is based exclusively on its textual form. Students know this, and take advantage of it. By using all the possibilities described above, they can modify the textual form of a program is such convoluted ways that it no longer resembles the original. If they have enough care not to change the semantics of the original, the copy will accomplish the same task. This suggests that the detection of copies should not be based on the textual form of the programs but on its semantics.

Given that programs are composed by subprograms, and there is no order between the subprograms, in order to identify plagiarism

ELS'19, April 01–02 2019, Genova, Italy

ACM ISBN 978-2-9557474-3-8.

in two programs we have to compare all the subprograms in one program against all the subprograms in the other program. If these subprograms are basic units, i.e., they do not contain any subprograms, then they can be compared in terms of the control structures they use, the primitive operations upon which they depend, etc. If both subprograms start by using an `if` control structure, most people would agree that this, by itself, is not a verdict of plagiarism, but it can be considered as a very small piece of evidence. In order to collect more pieces of evidence, we check the test form of the control structure. If in both cases this form is the application of the very same primitive function then we increase our confidence that we are facing plagiarism, otherwise, we decrease it. Then, we look at the consequent of the control structure and we apply similar reasoning. If, for example, in both consequents the program execute a loop until some criterion is fulfilled, this is another piece of evidence. We proceed by comparing the termination criteria of both loops, as well as the body of the loops. In case the forms in both subprograms do not match, we search for some of the common tricks students use to transform one into the other, such as swapping arguments in commutative operations. This process proceeds until we exhaust both subprograms. In the end, we decide whether we collected enough evidence to signal the subprograms as suspicious.

The process just described is commonly done by teachers that grade students' projects. However, they tend to do it only if they already suspect that there was some cheating, which makes the process unfair, as some are detected and others are not. Moreover, the process does not scale, either in the amount of information that teachers can collect, or in the number of comparisons between projects that they can do. In this paper, we address these two problems, presenting a tool built specifically to detect plagiarism between Lisp-based student projects.

3 AUTOMATIC PLAGIARISM DETECTION

We now describe an approach and a tool for automatic detection of plagiarism in Lisp-based projects. We will use projects written in the Common Lisp language [22] as an example but the tool is configurable to operate with other Lisp dialects and, with some extra effort, with other non-Lisp languages.

The tool is, essentially, a function which accepts two programs as arguments and returns a similarity value. This value abstracts away the particular similarities between the compared programs and is related to the level of confidence that we have in a verdict of plagiarism.

The tool can be decomposed into four main components:

- A parser that builds a syntax tree describing the programs to compare. This allows us to forget the textual form of the programs.
- A collection of specialized compare functions, each one designed to analyze particular features of the code that is being compared and to compute its similarity value.
- A data-driven matcher that identifies particular pieces of code within the programs to compare and selects the specialized compare functions appropriate to those particular pieces of code. This way, the tool can be easily extended or adapted to work with other languages.
- A combination function that combines all the similarity values found by the previous modules into a global value.

The parser is the easiest part of the tool because there are already some available for most high-level languages. In fact, for the Common Lisp language we just use the plain `read` function.

After parsing the two programs to compare, we feed the syntax trees found into the compare function. Since we may have one specialized compare function for comparing `if` forms, and another specialized compare function for comparing iteration forms, and so on, the top-level compare function will then check which one of the specialized compare functions is appropriate to analyze the two programs. To this end, each specialized compare function includes two patterns, each one matching one particular form on each of the compared programs. When the patterns match, the specialized compare function uses pattern variables to access the sub-forms of the programs and recursively compares those sub-forms using the top-level compare function until we reach atomic pieces of code.

In each recursive step, we collect pieces of evidence suggesting that we are facing copied programs. These pieces of evidence have a numeric value which may be larger or smaller depending on their importance. As an example, if both compared programs use a rarely used primitive, this is higher evidence of plagiarism than if both use a more frequent operation.

Since the number of forms to compare is quite large and there may be a large number of specialized compare functions to choose from, it is important that each of the operations involved—selecting the compare function, matching its patterns, and combining the amount of evidence found—executes quickly. We now analyze each of these operations.

3.1 Evidence

During the compare process we collect pieces of evidence *for* and *against* a verdict of plagiarism. For instance, it is unlikely that a Lisp function that starts with an `if` special form can be considered a copy of another that starts with the primitive function `car`. It is the combination of all collected evidence—for and against—that leads to the final verdict.

Expressing and combining evidence is a problem that has already been treated by [20], in the context of diagnosing medical problems. In this area, almost nothing is certain and doctors can only gather evidence for diseases based on the patient's symptoms. For instance, if I have a headache, I may have got a cold, but it is also possible that I have one or more of many other diseases such as cancer, meningitis, etc. To identify the correct disease, doctors collect and combine other pieces of evidence, like my body temperature, or the fact that I sneeze or not.

In our case, we are only interested in determining if the plagiarism "disease" is present and there are pieces of positive evidence and pieces of negative evidence for this possibility. For instance, when comparing two functions, an equal number of parameters constitutes a (small but) positive piece of evidence that they are a copy of each other, while a different number of parameters constitutes a negative one. Following the work of [20], we define evidence as a number in the interval $[-1, 1]$. A value in this interval represents the amount of evidence that we have about something and is called *certainty factor*. A certainty factor of 1 means absolute certainty that it is true, −1 means absolute certainty that it is false, and 0 means that we have nothing for or against—we simply do not know.

To combine evidence, we cannot simply add them, as we need to maintain the combined evidence within the interval $[-1, 1]$. Again, we refer to [20] to justify the following combination function C:

$$C(x, y) = \begin{cases} x + y - xy & \text{if } x > 0 \text{ and } y > 0, \\ x + y + xy & \text{if } x < 0 \text{ and } y < 0, \\ \frac{x+y}{1-\min(|x|,|y|)} & \text{otherwise.} \end{cases}$$

The parameters of the function are two certainty factors that we want to combine. The function combines then in such a way that combining true with something else except false is true, combining false with something else except true is false, combining unknown with something else does not change the result and combining true with false is an error as it would be equivalent to a contradiction. When two pieces of evidence are *for* something, that is, they are both positive, its combination is stronger evidence for that thing. When two pieces of evidence are *against* something, that is, they are both negative, its combination is stronger evidence against that thing. When one of them is positive and the other is negative, its combination will be something in between.

Although certainty factors have well-known problems regarding the dependency of evidence, the distinction between conflict and ignorance, and the undefined semantics of the certainty factor itself, they are simpler and computationally more tractable than other approaches such as probability theory [16], Dempster-Schafer theory [6], or fuzzy set theory [4].

As an example of the use of certainty factors in our tool, we now present the evidence defined for the specific situation of a Lisp conditional: the cond macro without a default clause.

```
(defevidence both-conds-miss-default 0.5 -0.3)
```

In the previous example, associated with the name of the evidence there are two values, one for the evidence in favor of plagiarism, and the other for the evidence against plagiarism. In this particular example, these evidences are relatively large because a cond without a default clause is a rare situation. When it occurs in the same place in two programs, it is a strong indicator of plagiarism. When it occurs in one program and not in the other, it is a medium indicator against plagiarism.

3.2 Defining Compare Functions

To be able to accurately compare Lisp forms, we must specify different comparing functions for different combinations of Lisp forms. In this section, we deal with the problem of specifying the *form* of Lisp forms.

Our idea is to use an association between patterns describing Lisp forms and expressions computing the evidence of a copy between those Lisp forms. The patterns will be matched against some form and, when successful, pattern variables will be bound to selected elements in the form. Then, the sub-forms designated by the pattern variables are compared and the pieces of evidence found are combined.

As an example of a specialized compare function, consider the situation where we have one `if` special form on each compared program:

```
(defcompare ((if ?test1 ?conseq1 ?altern1)
             (if ?test2 ?conseq2 ?altern2))
  (combine-evidence
    (get-evidence both-ifs t)
    (compare ?test1 ?test2))
    (compare ?conseq1 ?conseq2)
    (compare ?altern1 ?altern2))
```

The macro `defcompare` uses the common practice of tagging pattern variables with a leading question mark to distinguish them from other literal symbols. Note also that the body of the specialized compare function calls the generic compare function to compute the evidence for copy of the test, consequent, and alternative of both `if`s and combine the returned values with the specific evidence of having two `if`s on both programs.

Unfortunately, the previous comparison is too strict and does not search for additional signs of plagiarism. To that end, the actual comparison function is the following:

```
(defcompare ((if ?test1 ?conseq1 . ?altern1)
             (if ?test2 ?conseq2 . ?altern2))
  (combine-evidence
    (get-evidence both-ifs t)
    (compare-if-args ?test1 ?test2
                     ?conseq1 ?conseq2
                     (if ?altern1 (first ?altern1) nil)
                     (if ?altern2 (first ?altern2) nil))
    (get-evidence both-ifs-as-whens
      (and (null ?altern1) (null ?altern2)))))


(defun compare-if-args
    (test1 test2 conseq1 conseq2 altern1 altern2)
  (max (combine-evidence
        (compare test1 test2)
        (compare conseq1 conseq2)
        (compare altern1 altern2))
       (combine-evidence
        (compare test1 `(not ,test2))
        (compare conseq1 altern2)
        (compare altern1 conseq2))))
```

Note that, besides recognizing the case where the `if` is being used as a when, this improved comparison also verifies if the consequent and alternative might have been swapped.

The comparison function is opportunistically called with two forms and it will then try to match its patterns against those forms and, in case of success, bind the variables to the correspondent matched sub-forms. In case one of the patterns does not match, the function immediately returns with a null result.

Usually, the result of a successful match is a substitution list where pattern variables get their bindings. Unfortunately, there are two problems with this solution:

- The match is expensive. The result of the match is a structure that consumes time and space (that becomes garbage very soon). As the match is a fundamental operation there should be a minimum of garbage involved.
- There isn't a direct connection between the bindings found by the match process and the free variables in the compare function body.

Both problems can be solved if we take into account that the patterns are known at compile time. Thus, there is no need to depend upon a generic match algorithm. We can optimize every match process because we know in advance what it must do.

3.3 Partial Evaluation

The kind of optimization that we are talking about is named *partial evaluation* and is a technique that aims at speeding up a program by specializing it on some of its inputs.

In generic terms, if we have a program F that expects an input i, we define the execution of that program as the result of $F(i)$. Now, let us consider that the input i can be split into two parts, one *static*—known in advance—and the other *dynamic*. Let us write the execution of our program on these inputs as $F(s, d)$ where s and d are the static and the dynamic parts of the input i, respectively. Since we know the static part of the input, it may be possible to rewrite the program F so that all computations that depend upon the static part of the input are already done, that is, we want to write a new program F_s such that $F(s, d) = F_s(d)$.

The program $F_s(d)$ is named the *residual program* for F with respect to s or, equivalently, a version of F specialized to s.

A *partial evaluator* is a program that specializes other programs with respect to some static input. Historically, a partial evaluator is known as *mix* (after [9]). A partial evaluator is, then, a program mix such that for every program F and static input s, $\text{mix}(F, s) = F_s$.

In our case, the program F is the match algorithm, and the static input s which we want to use to specialize the program is the known pattern. The result of the partial evaluation is a specialized function that no longer receives the pattern but is capable of implicitly matching it. To do this, we write a partial evaluator $\text{mix}_{\text{match}}$ already specialized on the match program. We then use $\text{mix}_{\text{match}}$ on some pattern to produce a specialized matcher for that pattern: $\text{mix}_{\text{match}}(s) = \text{match}_s$

Although there are already many partial evaluators available (for example [10], [17], and [19]), we decided to build our own. This decision was made because our patterns are quite simple, allowing a specialized partial evaluator to generate very fast code. Besides, our partial evaluator generates code which is integrated with the evaluation of evidence. The result of our partial evaluation of a compare function is another function that interleaves the matching and binding processes until all patterns are matched, and then evaluates the compare function body in the lexical environment established by the binding process.

3.4 A Data-Driven Approach

Since there are many kinds of Lisp forms, we also have many specialized compare functions. These functions must be invoked by a generic compare function when and only when they are needed. Since we want to be able to add more specialized compare functions in the future, we need some way of doing it without disturbing the rest of the compare functions. The *data-driven* programming methodology [13] is an appropriate solution.

In a data-driven approach to our compare problem, we keep all specific compare functions on a database. This database is dynamic in the sense that new specific compare functions can be added at any time, even during the compare process. The generic compare function will use this database to identify the specific compare function appropriate for each situation. In many cases, there will be more than one specific compare function matching a particular situation. For example, if we consider the Lisp expression `(+ 1 2)`, we can easily build patterns that match it, such as `(+ 1 ?arg)`, `(+ ?arg1 ?arg2)`, `(?f ?arg1 ?arg2)`, `(?f . ?args)`, `(?car . ?cdr)`, or simply `?expr`. All these patterns may coexist in our system because they have different purposes: `(+ 1 ?arg)` matches an increment operation, `(+ ?arg1 ?arg2)` matches a sum operation, `(?f ?arg1 ?arg2)` matches a two-argument function application, `(?f . ?args)` matches a general function application, and so on. To decide which compare function should be used, there must be an order between all the patterns. This order is needed to ensure that more specific compare functions are tested before more generic compare functions.

There are two perspectives regarding this order:

- Automatic-based: If we can compute the specificity of a pattern, we can use it to sort the compare functions so that more specific patterns are tried before less specific patterns.
- Manual-based: Since text files are sequential, there is a sequential order in the compare function definitions. We can arrange our definitions so that their relative position reflects our intended matching order.

The first approach is similar to what is used in *CLOS*—The Common Lisp Object System [11, 22], where multi-methods can be specialized on the type of all arguments. These types have a subtype relation between them that must be taken into account when dispatching a generic function. In our case, we have patterns with different degrees of specificity and we also have multi-patterns, thus being comparable to CLOS. The problem with this approach is that it is not always obvious which patterns are more specific and thus, we might end up with the wrong order.

Depending exclusively on the second approach is also somewhat problematic. Whenever we want to add some new patterns, we need to carefully choose where to define them. If we define them after more generic patterns, the new patterns will never be checked. If we define them before more specific patterns, the older patterns will never be checked.

Our solution to the problem is a mixed approach. We will retain the flexibility of the CLOS approach while depending on the definition order in situations where a generic dispatch would be ambiguous. The idea is to define a subsumption relation between patterns, and use that relation to order the patterns. Whenever the subsumption relation cannot decide which pattern is more specific, we use the definition order. This approach follows the programmer's intention except when his intention leads to wrong results.

3.5 Indexing

When the number of compare functions is large, checking the patterns of all compare functions until one of them succeeds is very time-consuming. Besides, this time grows as we add more compare functions. To solve this problem, when patterns contain literals, they are used as indexes in an hash-table of compare functions. Since each compare function specifies two patterns, we have a doubly-indexed hash-table. On each entry of the hash-table, we keep all the compare functions whose patterns begin with the correspondent keys, sorted by the specificity of the rest of both patterns. This allows a very fast search of the appropriate compare function in most cases. In other cases, the indexing mechanism restricts the subset of applicable compare functions to just two or three, which is still quite good.

It is important to note that the indexing mechanism is itself implemented by a specialized compare function. This function has a parameter list which matches all patterns that can be indexed. When the match succeeds, the function uses the literals found to get the appropriate compare functions. This allows the indexing mechanism to be included in the system without modifying it. This is useful because different programming languages may require different indexing mechanisms and we can have several indexing mechanisms at the same time.

With this technique, our generic compare function needs only to check each specialized compare function in turn, according to the subsumption relation between the patterns of those functions. Some of these specialized compare functions are in fact indexing functions which speed up the selection of the appropriate compare functions. In case they are not applicable, the process continues, checking another function, being it a real compare function or just another indexing function. Obviously, it is possible to have indexing functions within indexing functions without limit. An indexing function can thus be seen as an abstraction of several compare functions.

4 COMPARING PROJECTS

The previous sections described the techniques that we developed for a generic compare process. We now describe some of the specific enhancements that specialize the process for the Lisp language.

4.1 Comparing Project Structure

Let us suppose we are comparing the following Lisp project:

```
(defun a (x y)
  (b x)
  (c y))
(defun b (z)
  (c z))
(defun c (z)
  z)
```

and the following copy:

```
(defun b1 (z1)
  (c1 z1))
(defun a1 (x1 y1)
  (b1 x1)
  (c1 y1))
(defun c1 (z1)
  z1)
```

The functions a, a1, b, b1, c, and c1 are so small that they can hardly be considered copies. Nevertheless, there are obvious resemblances between the two projects, as they share the same structure. In the previous example, a calls b and c while a1 calls b1 and c1.

In order to compare the projects' structures, a possible solution would be to generate the callers/callees graph of both projects and compare them with a graph comparing algorithm. However, there is a simpler solution: whenever we find functions applications, we check their definitions just as we are checking the current ones.

Using the previous example, while comparing a and a1 we find that they call, respectively b and b1. Then, we compare b and b1 just to find that they call c and c1. Again, we compare c and c1. The result of the comparison is then returned and combined with the comparison of b and b1, and the result is returned and combined with the comparison of a and a1. This way, functions near the root of the graph will see its comparison getting more and more precise as each called function is compared.

This process also takes into account self-recursive and mutually-recursive functions. When in presence of such cases, we stop the check (avoiding infinite regression) and we return with a value reflecting that additional piece of evidence.

4.2 Comparing Lisp Forms

As we said before, when we compare two Lisp forms we first try to use specialized compare functions for those forms. The following list is just a short extract of some cases that the specialized compare functions deal with:

- To compare two functions, we check their names, their arguments lists, their documentation strings, and their bodies.
- Common transformations involving `let` forms consist of exchanging some bindings, and replace `let` with `let*` and vice-versa. This forces us to compare `let` forms, `let*` forms, and combinations of them.
- Another common transformation consists of cascading `lets`. Instead of using a single `let` to establish bindings, it is possible to use a large number of `lets`, each one establishing some of the bindings. Although the result is semantically similar, syntactically it looks very different. To avoid this trick, we collect all subsequent lets. Although we may be changing the semantics of the form, the change is not relevant to the compare process.
- While comparing bindings, we must be careful about unusual bindings. Although the usual `let` form is similar to `(let ((var val)) body)`, it is possible to have uninitialized bindings, such as `(let ((var)) body)`, or unique uninitialized binding such as `(let (var) body)`.
- When comparing `ifs`, we give special attention to the exceptional cases of `ifs` without alternative because they are rare. In this case, the `ifs` look like whens.
- When an `if` is copied, most students are smart enough to modify it by negating the test and swapping the consequent with the alternative. To deal with this case, we must also compare the possible transformation. We return the highest value found.
- Since the macro cond is very common, it is difficult to depend on it to detect copies. For this reason, we just dispatch on the tests and actions. However, we do analyze the default clause because its absence is a rare situation. If two cond forms both miss the default clause, that is a strong sign that they may have been copied.
- One of the common copying practices is replacing a cond by an `if` and vice-versa. To handle this case, we compare the first cond clause with the condition and consequent of the `if` and we compare a new cond containing the remaining clauses with the alternative of the `if`. This approach allows the comparison of a multi-clause cond with a cascade of `if` forms.

- Transforming between an `if` and `when` is another common plagiarism technique. The only tricky situations are (1) that a `when` with multiple consequents implies an `if` with a `progn` consequent and (2) that the `if` should not have an alternative or it should be `nil`.
- Another common transformation is between `if` and `unless`. As with the `if` and `when` comparison, the only tricky situations are (1) that an `unless` with multiple consequents implies an `if` with a `progn` consequent and (2) that the `if` should not have an alternative or it should be `nil`. Besides, the `if` must have a negated test or have the consequent swapped with the alternative.
- Substituting a `dotimes` with a `do` and vice-versa are also good approaches to hide plagiarism. To detect these situations, we compare the corresponding parts, also including the `dotimes` macro-generated ones (namely, stopping condition for the loop and increment expression).
- Comparing `setq`s and `setf`s needs to take into account that each variable must have its value, but the order between variable-value pairs can be exchanged (to a certain extent).

4.3 Comparing Transformations

We have already exposed some of the common copying techniques explored by students. Some of these techniques are sufficiently general to be easily formalized. On this set, we include changing parameter names, permuting function arguments, and cascading `let` bindings.

However, there is another set of techniques that are much more elaborated and involve more knowledge about Lisp functions. Transforming `(1+ expr)` into `(+ expr 1)` is an example. They are semantically equivalent expressions and either can be used. Since they are syntactically different, they make copy detection a harder task. There are much more examples, for instance, `(null expr)` and `(not expr)`.

Different Lisp functions that perform very similar operations in most contexts are another source of problems. The functions `car` and `first` are equivalent and either can be used. The functions `endp` and `null` have similar semantics and in most situations either can be used.

We extended our tool with some syntactic sugar to make it easy to define new compare functions to recognize such equivalent forms. Below we present some of the defined transformations.

```
(1+ ?x) ≡ (+ ?x 1)
(1- ?x) ≡ (- ?x 1)
(< ?x ?y) ≡ (>= ?y ?x)
(> ?x ?y) ≡ (<= ?y ?x)
(null ?x) ≡ (eql ?x nil)
(zerop ?x) ≡ (= ?x 0)
(car ?x) ≡ (first ?x)
(cdr ?x) ≡ (rest ?x)
(null ?x) ≡ (endp ?x)
(= (length ?list) 1) ≡ (endp (rest ?list))
(not (null ?list)) ≡ ?list
(cons ?elem nil) ≡ (list ?elem)
```

As is visible, we describe the equivalences using repeated pattern variables in both patterns. Usually, when a pattern variable is repeated on two patterns, it means that for them to match, the variables' values must be equal. Here, it just means that we should compare the values on each matched form. When we enter the equivalent forms into the system, all pattern variables are renamed and then two new compare functions with permuted arguments are defined so that both transformations can be tried.

In order to define these transformations, the macro `defequivs` takes a list of equivalent forms and, optionally, the corresponding evidence. As an example, the form

```
(defequivs ((cadr ?x) (car (cdr ?x)) (second ?x) (nth 1 ?x)))
```

creates 12 comparing functions, each dealing with one of the two-element combinations of the given patterns.

5 RESULTS

Validation is an important part of the plagiarism detection process. To this end, we also developed a mode for the Emacs editor [21] that simplifies the verification of the results. The mode presents, side by side, the fragments of code that were considered sufficiently similar to merit a careful observation.

We tested our tool on a course where students had to create a moderately complex project in Common Lisp. One of the requirements was that students had to implement a small number of functions with a pre-established signature.

The students submitted 112 projects with an average of 27 implemented functions per project. We conducted tests where we compared just a selected function and tests where we compared all the implemented functions. On the first case, there were 6216 comparisons between projects. On the second case, the number raises to 4.4 million.

5.1 Analyzing a Selected Function

In this test, we analyzed a selected function that all students had to implement. The results are in Figure 1, where each axis represents the project's number and each square has a shade that is proportional to the amount of evidence found. In this figure, we note the selectivity of the compare process. Although all projects defined the selected function, only a few pairs (16/6216 = 0.26%) were found suspicious. This means that originality was high, that is, there were many different implementations of the selected function. On the other hand, if we count the number of suspicious projects (not *pairs* of projects), the ratio grows to 21/112 = 18.8%. Note that the first percentage represents the number of comparisons that were found positive, while the second percentage represents the affected projects. The first number will only be 100% when all students use the very same function, while the second can reach that value if there is one copy for each original function.

To evaluate the tool's accuracy, all suspicious projects were manually checked. We also checked projects with small negative evidence to be sure that the tool did not miss anything. This was important to determine an evidence level which justifies further inspection. This value depends on the average size of the functions, the difficulty of the assignment, etc. In our case, we found that evidence below 0.6 does not represent plagiarized functions but, instead, functions that happen to be similar.

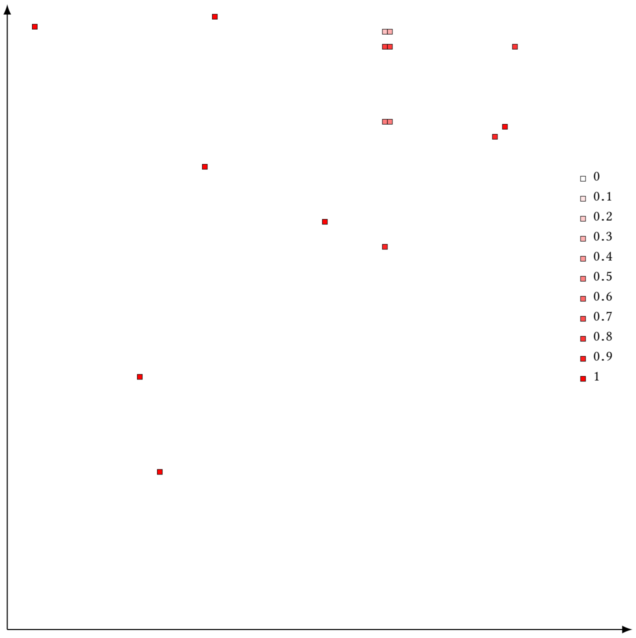


Figure 1: Evidence of copy of a selected function between all pairs of projects. Both axis represent the set of compared functions.

5.2 Analyzing Projects

An unrestricted test involving all projects' definitions can produce a very large number of suspicious pairs of definitions. To avoid cluttering the results with irrelevant information, we only searched for very strong evidence of copy (larger than 0.8). Note that we excluded from this test the critical functions tested by the previous analysis.

Figure 2 represents the results of the analysis on pairs of projects. A combined analysis of the previous two figures shows that there are data points in Figure 2 that are not present in Figure 1. This means that there are students who are careful enough to develop the critical parts of the project but careless in what regards less relevant parts.

In the end, the final number of copied projects detected was 28, which represents 25% of all projects, the highest ever recorded on that course.

5.3 Pedagogical Effects

Besides helping teachers detecting plagiarism, the tool is invaluable as a pedagogical measure. The mere fact of knowing that assignments will be scrutinized regarding possible plagiarism entails in the student a completely different attitude.

To verify the change in students' habits, we run the tool again one year later, on the very same course. This time, the project was given to 132 groups of students and 109 of them delivered a solution.

We present in Figure 3 the results of the plagiarism detection process. As before, only pairs of projects with more than 2 copied forms are shown. As we can see, there are much less copied projects now.

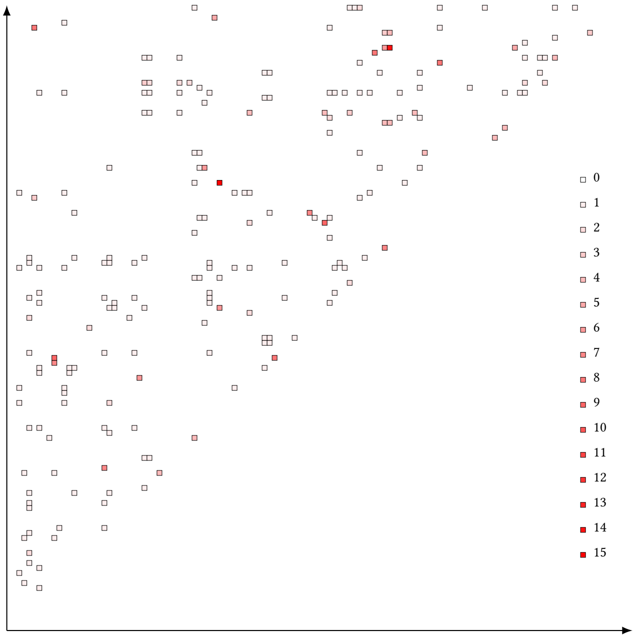


Figure 2: Number of copies among pairs of projects. Both axis represent the set of compared projects.

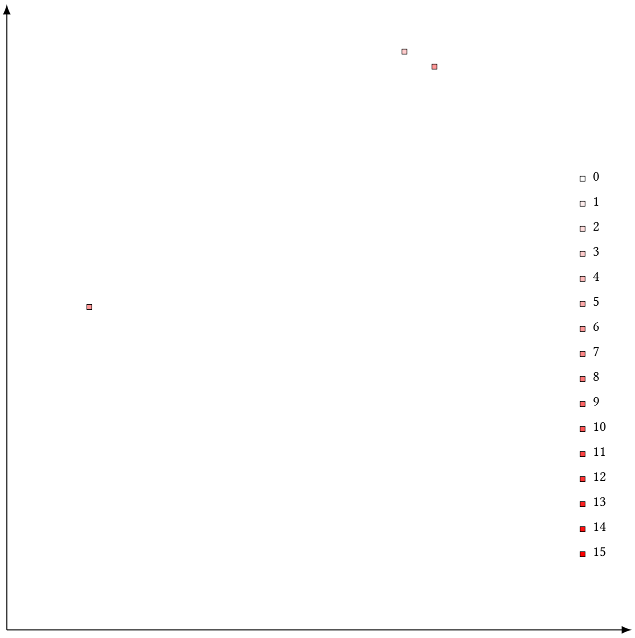


Figure 3: Number of copies among pairs of projects in the following year. Both axis represent the set of compared projects.

In our opinion, the main reason for these results is the psychological effect of the perceived increase in the plagiarism detection effort. Independently of the students' suspicions regarding the real cause of that increase, they quickly adapt to this perception by either avoiding plagiarism or by increasing the effort to modify the plagiarized projects so that cheating would not be detected.

6 RELATED WORK

The plagiarism detection tool here presented was invented in 1995. Despite being frequently used in the following years, it remained a closely guarded secret because we were afraid that it would not be well-received by the student's association. Nowadays, when automatic grading is already a given in universities, plagiarism detection would not surprise anyone but, at that time, that was not the case. In fact, the existence of the tool was acknowledged only after several other similar tools were announced.

Plagiarism detection was studied by several authors, including [3, 5, 7, 12, 14]. The proposed approaches include textual comparison [1], that compares source code modulo name changes, abstract syntax tree (AST) comparison [2, 23, 24], which is immune to textual changes but does not detect code transformations, metric analysis [5, 7], that detect code framents that have a similar number of unique operators, operands, declared variables, etc., and fingerprinting [8, 18], based on the use of hashing functions that are immune to the typical code transformations done to hide plagiarism, so that two plagiarised fragments produce the same hash value.

The work here presented can be considered as an extended AST-based comparison approach that is immune to the code transformations generally employed to hide plagiarism, and where the similarities and differences between AST nodes are evaluated using a metric-based approach. The metrics were derived from our experience regarding the code that is typically written by our students. A final difference is the adaptability and extensibility of our approach, which was designed to easily support user-defined comparisons and metrics.

7 CONCLUSION

We described a tool to identify plagiarism in students' projects. The tool compares program fragments and gives a certainty factor to the question "is plagiarism present in these fragments?". Based on this certainty factor, the teacher can quickly identify students' projects that deserve more careful attention.

We have used the tool on a course and we found a surprisingly high number of copied projects. However, the results obtained in the following years show that students quickly adapt to the increased scrutiny by dramatically reducing plagiarism.

There are other important uses for this tool. In particular, it can be used to detect redundancies in code. By comparing the code against itself one can detect which code fragments are sufficiently similar to deserve being substituted by a conveniently parameterized function.

8 ACKNOWLEDGMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2019.

REFERENCES

[1] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In Linda M. Wills, Philip Newcomb, and Elliot J. Chikofsky, editors, *Proceedings: Second Working Conference on Reverse Engineering*, pages 86–95. IEEE Computer Society Press, 1995. ISBN 0-8186-7111-4.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In T. M. Koshgoftaar and K. Bennett, editors, *Proceedings; International Conference on Software Maintenance*, pages 368–378. IEEE Computer Society Press, 1998. ISBN 0-7803-5255-6, 0-8186-8779-7, 0-8186-8795-9.

[3] H. L. Berghel and D. L. Sallach. Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices*, 19(8):65–76, August 1984. ISSN 0362-1340.

[4] Didier Dubois and Henri Prade. An introduction to possibilistic and fuzzy logics. In et al Smets, editor, *Non-Standard Logics for Automated Reasoning.* Academic Press, 1988. Reprinted in Readings in Uncertain Reasoning.

[5] J. A. W. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity within a university programming environment. *Computers and Education*, 11(1):11–19, 1987.

[6] Jean Gordon and Edward H. Shortliffe. The dempster-shafer theory fo evidence. In Bruce G. Buchanan and Edward H. Shortliffe, editors, *Rule-Based Expert Systems*, pages 272–292. Addison Wesley Publishing Company, Reading, Massachusetts, 1984.

[7] S. Grier. A Tool that Detects Plagiarism in PASCAL Programs. *SIGSCE Bulletin*, 13(1), 1981.

[8] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of CASCON '92,* (Toronto, Ontario; November 9-11, 1992), pages 171–183, November, 1992.

[9] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Springer-Verlag, 1985.

[10] J. Jørgensen. Generating a pattern matching compiler by partial evaluation. In S. L. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 177–195. Berlin: Springer-Verlag, 1991.

[11] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A programmer's guide to CLOS.* Addison-Wesley Publishing Company, Cambridge, MA, 1989.

[12] I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. In *Proc. 18th NIST-NCSC National Information Systems Security Conference*, pages 514–524, 1995.

[13] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann, San Mateo, California, 1992.

[14] K. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGSCE Bulletin*, 8(4):30–41, 1976.

[15] A. Parker and J. Hamblen. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, 32(2):94–99, May 1989.

[16] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, California, 1988.

[17] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to symbolic pattern matching with intelligent backtrack. In M Billaud, P Castéran, MM Corsini, K Musumbu, and A Rauzy, editors, *WSA '92—Workshop on Static Analysis*, number 81-82 in bigre, pages 109–117, Bordeaux (France), September 1992.

[18] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[19] P. Sestoft. ML pattern match compilation and partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 446–464. Berlin: Springer-Verlag, 1996.

[20] E. H. Shortliffe. *MYCIN: A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection.* PhD thesis, Stanford Artificial Intelligence Laboratory, Stanford, CA, October 1974.

[21] Richard M. Stallman. EMACS: The extensible, customizable, self-documenting display editor. Technical Report AIM-519A, Massachusetts Institute of Technology, June 1979. URL ftp://publications.ai.mit.edu/ai-publications/500-999/AIM-519A.ps.

[22] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, second edition edition, 89. URL ftp://cambridge.apple.com/pub/CLTL/CLTL.tar.gz.

[23] Wise. Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 24, 1992.

[24] Wise. YAP3: Improved detection of similarities in computer program and other texts. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 28, 1996.

Session V: Bootstrapping

Tuesday, 2.4.2019

09:00–10:00	Christophe Rhodes: 20 more years of bootstrapping (ELS keynote)
10:00–10:30	Irène Anne Durand and Robert Strandh: Bootstrapping Common Lisp using Common Lisp
10:30–11:00	**Coffee Break**

Bootstrapping Common Lisp using Common Lisp

Irène Durand
Robert Strandh
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ABSTRACT

Some Common Lisp implementations evolve through careful modifications to an existing *image*. Most of the remaining implementations are bootstrapped using some lower-level language, typically C. As far as we know, only SBCL is bootstrapped from source code written mainly in Common Lisp. But, in most cases, there is no profound reason for using a language other than Common Lisp for creating a Common Lisp system, though there are some annoying details that have to be dealt with.

We describe the bootstrapping technique used with SICL, a modern implementation of Common Lisp. Though both SICL and the bootstrapping procedure for creating it are still being worked on, they are sufficiently evolved that the big picture outlined in this paper will remain valid. Our technique uses *first-class global environments* to isolate the host environment from the environments required during the bootstrapping procedure. Contrary to SBCL, and implementations written in some other language, in SICL, we build the CLOS MOPclasses and generic functions *first*. This technique allows us to use the CLOS machinery for many other parts of the system, thereby decreasing the amount of special-purpose code, and improving maintainability of the system.

CCS CONCEPTS

• **Software and its engineering → Compilers**; *Multiparadigm languages*;

KEYWORDS

CLOS, Common Lisp, Compilation, Bootstrapping

ACM Reference Format:
Irène Durand and Robert Strandh. 2019. Bootstrapping Common Lisp using Common Lisp. In *Proceedings of the 12th European Lisp Symposium (ELS'19)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.2634314

1 INTRODUCTION

In this paper[1] , by *bootstrapping* a Common Lisp system we mean creating some *target* Common Lisp system by building it from its associated source code, using various *tools* and *language processors* to transform that source code into an *executable file* for some typical operating system such as GNU/Linux. The typical way of making such a target Common Lisp system evolve through maintenance, is to modify its source code and then restart the bootstrapping procedure to build an updated executable file.

Not all Common Lisp systems are created so that they can evolve this way. Some systems evolve by the careful modification of an existing *executing image* which is then saved as an executable file that can be executed as usual.

In this paper we will concentrate on the technique of bootstrapping used with SICL[2].

Before we can start investigating different options for bootstrapping, we must deal with an annoying but crucial detail, namely the definition of *source code*. The Free Software Foundation defines it as "the preferred form of a work for making modifications to it". We agree completely with this definition. It excludes the use of code that was automatically produced. In practice, it also excludes code written directly in machine language and most code written in assembly language, with the exception of (a) small code fragments that can not be expressed easily in some other language, and (b) code fragments that are part of a code generator written in some higher-level language.

However, for it to be possible for the source code of a Common Lisp system to be turned into an executable file, there must be *language processors* (i.e., compilers and/or interpreters) available that can handle the languages that the source code is expressed in. The main debate when it comes to bootstrapping techniques seems to be what is meant by *available* in this context. A common definition seems to be something like *whatever is available on a GNU/Linux system out of the box.*

One of the consequences of such a definition of *available* is that, in order to write a Common Lisp system, one has to use some programming language considered lower level than Common Lisp itself. Typically, C plays this role.

In this paper, we argue that one of the main reasons of the creator(s) of a target Common Lisp system wanting such a system in the first place, is that they are convinced of the virtues of this language for writing programs. Furthermore, Common Lisp is uniquely well adapted to writing language processors. The obvious choice for a language for writing a Common Lisp system is therefore Common Lisp itself. Since there is now a multitude of good Common Lisp implementations available and easily installable on widely-used operating systems, we think that Common Lisp should be considered

[1] In this paper, we assume that the reader is familiar with the metaobject protocol for implementing CLOS, as described in the book [1] that is dedicated to the subject.

[2] https://github.com/robert-strandh/SICL

ELS'19, April 01–02 2019, Genova, Italy

ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2634314

to be a language for which there are language processors *available* for bootstrapping.

2 PREVIOUS WORK

2.1 Overview of existing techniques

In his excellent paper describing how SBCL is bootstrapped [2], Rhodes gives an overview of how different existing Common Lisp systems are made to evolve. Below, we summarize the contents of that paper.

We can divide Common Lisp implementations into those that are mostly written in some other language, and those that are mostly written in Common Lisp.

In the first category, there are implementations that specifically cater to applications written in that other language and that need some scripting capabilities that are supplied by the Common Lisp implementation. Whether it is advantageous or not for these implementations to be written mainly in that other language is outside the scope of this paper.

Of the implementations in the second category that are currently actively used, Rhodes claims[3] that Allegro, LispWorks, CMUCL, Scieneer, and CCL are only possible to build using older versions of the same system, and only using image-based techniques. Only SBCL can be bootstrapped using several other Common Lisp implementations.

Even a Common Lisp implementation that is largely written in Common Lisp such as SBCL has some amount of code written in other languages. In the case of SBCL, Rhodes gives the number 35 000 lines of C and assembly code "for services such as signal handling and garbage collection", of which 8 000 is for the garbage collector. The remaining lines can be summarized as around 2 000 lines per operating system supported. This is a very modest amount of code written in other languages.

2.2 Common Lisp systems in other languages

When a language such as C or C++ is used to implement a Common Lisp system, a small subset of the Common Lisp language is implemented this way. We call that subset the *base* language. The result of the initial bootstrapping procedure is typically an executable file containing the base system. Additional modules are then added to the base system to obtain a complete Common Lisp system. These additional modules must be implemented in the subset of Common Lisp defined by the base language and previously added modules.

There are several issues with this technique. For one thing, some major components that would be more easily expressed in Common Lisp must be written using the implementation language so that new modules can be added to the system, in particular a reader and an evaluator.

Another major issue has to do with maintenance. When one of the additional modules is modified, it is easy to forget exactly what subset of the Common Lisp language is allowed at that point in the bootstrapping procedure. The code for a particular module must often be expressed in some unidiomatic way and it is tempting to make the modified code more idiomatic, but doing so will then break the bootstrapping procedure.

2.3 Common Lisp systems in Common Lisp

Because of the way compilation is defined by the Common Lisp standard, there are some issues that need to be resolved in order for it to be possible for a target Common Lisp system to be bootstrapped on a host Common Lisp system. Since SBCL is very likely the only Common Lisp implementation written mostly in Common Lisp that can be built from an existing Common Lisp implementation, we describe how SBCL solves some of these issues.

2.3.1 Packages and environments. Most existing Common Lisp systems have a single global environment that is used both as the compilation environment and as the run-time environment. Compiling Common Lisp source code requires the existence of definitions of macros, types, etc. in that environment, and when source code for a target Common Lisp system is compiled using a host Common Lisp system, these definitions must be those of the target system. However, with a single global environment there can only be one definition of these entities.

SBCL solves this problem by using different package names for the code of the host system and the target system. In a final step, the packages of the target system are then renamed to conform to the standard.

2.3.2 The compiler and CLOS. Some aspects of CLOS require the presence of the compiler, at least if the resulting code is required to have some reasonable performance. In particular, the compiler is required to create a discriminating function from the effective methods[4] returned by `compute-effective-method`. For that reason, it becomes difficult to use generic functions and standard classes in the code of the compiler itself.

SBCL solves this issue by not using generic functions and standard classes in the code of the compiler. Thus, SBCL can load the compiler into a minimal running target system and then bootstrap CLOS afterwards.

However, not using generic functions and standard classes in the compiler has some of the same problems as Common Lisp systems that are written in some other language, namely that care has to be taken to make sure the proper subset of the language is used when the code of the compiler is being worked on. Furthermore, generic functions and standard classes are great tools for structuring complex code, so not being able to use these tools in such a significant and complex part of a Common Lisp implementation negatively affects the clarity and maintainability of the code.

3 THE SICL SOURCE CODE

SICL is a system that is written entirely in Common Lisp. We decided to use the full language to implement the system so as to avoid having to define and remember what subset of the language is allowed for which modules. Thus, the compiler, called Cleavir[5], makes heavy use of generic functions and classes. By using these two types of objects, we can have a compiler that is adaptable to different Common Lisp implementations. It is currently used as the

[3] For the commercial Common Lisp implementations cited in the paper by Rhodes, he includes a disclaimer that only anecdotal evidence for this information is available.

[4] Recall that the result of a call to `compute-effective-method` is a lambda expression. This lambda expression must be turned into something that is executable, hence the need for an evaluator.

[5] Cleavir resides in the SICL repository on GitHub.

main compiler of Clasp[6], and recently, a Cleavir-based compiler has been written for CLISP[7].

In addition to using the full language for the implementation of SICL, we want the code to be as idiomatic as possible. For example, our definition of the class t, looks like this:

```
(defclass t ()
  ()
  (:metaclass built-in-class))
```

This definition clearly expresses the characteristics of the class t. It has no superclasses because no superclasses are explicitly mentioned, and the metaclass built-in-class does not provide any default superclasses like standard-class and funcallable-standard-class do. While this definition of the class t is clear, it is not operational as is. The metaclass built-in-class is an indirect subclass of the class t, so the class t must exist in order for the class built-in-class to exist.

Our definitions of the classes class and standard-class look like this:[8]

```
(defclass class (specializer)
  ((%name :initform nil :initarg :name ...)
   ...
   (%direct-subclasses :initform '() ...)))
(defclass standard-class (class)
  (...))
```

Again, these definitions are clear. No metaclass option is given, so the metaclass defaults to standard-class. Like the defintion of t, these definitions are not operational as is, because the class standard-class must exist in order to be the metaclass of itself.

In a Common Lisp implementation that must bootstrap CLOS from a subset of the language that does not include CLOS, some other mechanism must be used. As an example of the consequences of the use of such a subset, consider the following definitions from ECL[9]:

```
(defparameter +class-slots+
  `(,@+specializer-slots+
    (name :initarg :name :initform nil ...)
    ...
    (direct-subclasses :initform nil ...)
    ...))
(defparameter +standard-class-slots+
  (append +class-slots+
          '((optimize-slot-access)
            (forward))))
```

Here, two special variables are defined, each one containing the specifications of the direct slots of a class. These two definitions express the exact same information as two defclass forms defining the classes class and standard-class, respectively. However, because the defclass form can not be used at this stage of the bootstrapping procedure, a different mechanism must be used.

In addition to using the CLOS machinery for defining the classes defined by the metaobject protocol, we use the same machinery for defining system classes. For example, our definition of the class symbol looks like this:

```
(defclass symbol (t)
  ((%name :reader symbol-name)
   (%package :reader symbol-package))
  (:metaclass built-in-class))
```

Not only is this definition clear, it is also operational. By using the CLOS machinery for definitions of system classes, we avoid having to use an additional, special, mechanism for this purpose.

In contrast, consider this definition of the system class symbol from SBCL:

```
(define-primitive-object
    (symbol :lowtag other-pointer-lowtag
            :widetag symbol-header-widetag
            :alloc-trans %make-symbol
            :type symbol)
  ...
  (name :ref-trans symbol-name :init :arg)
  (package :ref-trans symbol-package
           :set-trans %set-symbol-package
           :init :null)
  ...)
```

Again, a special mechanism must be used, since CLOS is not available when the type symbol must be defined.

The purpose of the SICL bootstrapping procedure is to make these idiomatic definitions operational in the host environment so as to create a graph of objects isomorphic to that of the target system, and then to create the target graph in an executable file.

By doing it this way, we simplify system maintenance. The bootstrapping procedure is able to work with the definitions of classes, generic functions, and methods using the standard macros defclass, defgeneric, and defmethod, even though these definitions would not be operational in a system that needs to build up functionality from a language subset that does not include CLOS. The SICL maintainer is thus free to alter definitions of core system objects, relying on the bootstrapping procedure to make those definitions operational and ultimately turning them into an executable system.

4 OUR TECHNIQUE

4.1 SICL object representation

A SICL object is represented in one of three different ways:

- As an *immediate* object where the object is stored in the pointer itself, with the appropriate tag bits. Fixnums, characters and single floats are represented this way.
- As a two-word block. This is how cons cells are represented.
- As a two-word block called a *header* where the first word points to a class object, and the second word points to a sequence of words, called the *rack*, that contains the slots of the object. All objects other than immediates and cons cells are represented this way. We call this representation a *general instance*.

The first word of the rack contains a *stamp* which is a unique integer taken from the class when the instance was created. The stamps of the arguments to a generic function are used by the

[6] https://github.com/clasp-developers

[7] https://clisp.sourceforge.io/

[8] In reality, there are intermediate classes between class and standard-class that are not shown here.

[9] https://common-lisp.net/project/ecl/

generic dispatch technique to determine which effective method to execute. The object representation and generic dispatch technique has been described in detail previously [3], but this short summary is sufficient to understand our bootstrapping technique.

In the description of our technique, we use the word *class* in a general way, as an object that can be used as a model for the creation of *instances*. Thus the word *class* does not imply that it is a class in the sense of the host Common Lisp implementation. While this usage of the word *class* may seem odd, recall that a class is just an ordinary Common Lisp object that is passed as an argument to `make-instance` and other functions called by it which then returns a different object. We exploit this idea by supplying our own definition of `make-instance` in different phases of the bootstrapping procedure.

Similarly, we use the word *generic function* in a general way, as an object that can be executed and that can have methods associated with it, providing partial implementations of the generic function. Again, while this usage of the word *generic function* may seem odd, recall that a generic function is simply an ordinary Common Lisp object of type `funcallable-standard-object` for which the ultimate definition (called the *discriminating function*) is computed by combining partial definitions (the *methods*) associated with it. We exploit this fact by providing different representations of generic functions in different phases of the bootstrapping procedure, and by supplying different versions of `compute-discriminating-function` adapted to each phase. Thus, a *generic function* is not a generic function in the sense of the host Common Lisp implementation. However, during the bootstrapping procedure, these objects are executable in the host system, because they are instances of the host class `funcallable-standard-object`.

4.2 Environments for bootstrapping

Our technique uses several first-class global environments [5] to create a graph of objects that is isomorphic to the graph of objects to be written to the executable file instantiating the target Common Lisp implementation. By using first-class global environments, we avoid the problems related to packages and environments cited in Section 2.3. The main feature of our technique, though, is that we create the generic functions and classes of the metaobject protocol *first*.

The environments are filled with definitions mainly as a result of loading files containing production SICL code, though some code specific to bootstrapping is required as discussed at the end of this section. This loading procedure uses the Eclector[10] reader and the Cleavir compiler to produce intermediate code in the form of a fairly conventional *flow graph* of instructions. The Cleavir compiler takes a first-class global environment as an argument, and uses this environment to search for definitions of macros, classes, types, etc. The resulting intermediate code is then translated in two different ways:

(1) Native target code is generated from it, and attached to host objects representing executable target objects such as ordinary functions, generic functions, and methods.[11]
(2) It is translated to a simple subset of Common Lisp code that accesses that same environment for definitions of functions and other objects. This Common Lisp code is then compiled using the host compiler in order to make it executable in the host.

The remainder of this section is concerned with how the host-executable code is used in order to determine the graph of target objects represented as an isomorphic graph of host objects.

4.3 Definitions

In preparation for the bootstrapping procedure, several first-class global environments are created and filled with definitions of SICL macros. The definitions of those macros reside in production SICL files. Little or no special code is required for those definitions.

A number of host object types are used during bootstrapping, in particular symbols, packages, `cons` cells, and integers. However, when such an object is used as an argument to a SICL generic function, a special version of `class-of` assigns a SICL class object as its type. Some of the host functions operating on these kinds of objects are *imported* to our environments in preparation for the bootstrapping procedure.

To facilitate the description of our technique, we need some definitions:

Definition 4.1. A *host class* is a class in the host system. If it is an instance of the host class `standard-class`, then it is typically created by the host macro `defclass`.

Definition 4.2. A *host instance* is an instance of a host class. If it is an instance of the host class `standard-object`, then it is typically created by a call to the host function `make-instance` using a host class or the name of a host class.

Definition 4.3. A *host generic function* is a generic function created by the host macro `defgeneric`, so it is a host instance of the host class `generic-function`. Arguments to the discriminating function of such a generic function are host instances. The host function `class-of` is called on some required arguments in order to determine what methods to call.

Definition 4.4. A *host method* is a method created by the host macro `defmethod`, so it is a host instance of the host class `method`. The class specializers of such a method are host classes.

Definition 4.5. A *simple host instance* is a host instance that is neither a host class nor a host generic function.

Definition 4.6. An *ersatz instance* is a target general instance (as defined in Section 4.1) represented as a host data structure, using a host standard object to represent the *header* and a host simple vector to represent the *rack*. In fact, in order for the ersatz instance to be callable as a function in the host system, the header is an instance of the host class `funcallable-standard-object`.

Definition 4.7. An ersatz instance is said to be *pure* if the class slot of the header is also an ersatz instance. An ersatz instance is said to be *impure* if it is not pure. See below for more information on impure ersatz instances.

Definition 4.8. An *ersatz class* is an ersatz instance that can be instantiated to obtain another ersatz instance.

[10] https://github.com/robert-strandh/Eclector
[11] We do not yet have a code generator for native executable code, so currently this part of the bootstrapping procedure is omitted.

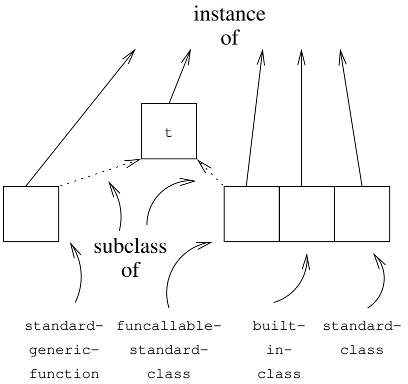


Figure 1: Simplified diagram of MOP classes.

Definition 4.9. An *ersatz generic function* is an ersatz instance that is also a generic function. It is possible for an ersatz generic function to be executed in the host system because the header object is an instance of the host class `funcallable-standard-object`. The methods on an ersatz generic function are ersatz methods.

Definition 4.10. An *ersatz method* is an ersatz instance that is also a method.

Definition 4.11. A *bridge class* is a representation of a target class as a simple host instance. An impure ersatz instance has a bridge class in the class slot of its header. A bridge class can be instantiated to obtain an impure ersatz instance.

Definition 4.12. A *bridge generic function* is a representation of a target generic function as a simple host instance, though in order for it to be executed by the host, it is an instance of the host function `funcallable-standard-object`.

Arguments to a bridge generic function are ersatz instances. The bridge generic function dispatches on the *stamp* (See Section 4.1.) of its required arguments.

The methods on a bridge generic function are bridge methods.

Definition 4.13. A *bridge method* is a target method represented as a simple host instance. The class specializers of such a method are bridge classes. The *method function* of a bridge method is an ordinary host function.

4.4 Bootstrapping phases

The essence of our technique consists of four phases (1 to 4), using six first-class global environments. An initial phase 0 imports host classes to environment E_0. Only classes that are required in phase 1 are imported. Classes `standard-method`, `standard-generic-function`, and the class used to represent slots `standard-direct-slot-definition` are imported with the same. Classes `standard-class`, `built-in-class`, and `funcallable-standard-class` in environment E_0 all refer to one and the same host class, namely a subclass of the host class `funcallable-standard-class`.

In each phase $i > 0$, three first-class global environments are involved, E_{i-1}, E_i, and E_{i+1}. Before phase i starts, E_{i-1} contains classes to be instantiated during phase i, and E_i contains generic functions that are not involved in phase i, but that will be used in phase $i+1$ to operate on the instances of the classes in E_{i-1}. Some of the generic functions in E_i are accessor functions containing methods that were automatically added as a result of the classes in E_{i-1} being defined. Others are higher-level functions that call those accessors to accomplish tasks such as initialization of various metaobjects, class finalization, creation of effective methods, and creation of discriminating functions.

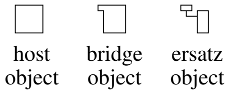


Figure 2: Objects in different phases.

A phase i has two main steps:

(1) Accessor generic functions are created in E_{i+1} by loading SICL production code containing `defgeneric` forms. These generic functions are accessor functions for MOP classes and MOP generic functions. These functions are created in E_{i+1} rather than in E_i so as to protect the existing functions in E_i that are needed later.
(2) Classes are created in E_i by loading SICL production code containing `defclass` forms. As a result of the creation of these classes, methods are automatically added to the corresponding accessor generic functions in E_{i+1}.

Depending on the phase, SICL production code might be loaded before the first step, between the two steps, or after the last step.

Four phases accomplish the creation of a number of objects, ending with a complete set of ersatz objects. The result of each phase is illustrated by a separate figure. In these figures, the shape of each object illustrates its type as shown in Figure 2.

The four phases accomplish this following results:

(1) Host classes and host class metaclasses in E_0 are used to create host generic functions in E_2 and host classes in E_1. The result of this phase is illustrated in Figure 3.
(2) Host classes in E_1 are used to create bridge generic functions in E_3 and bridge classes in E_2. The result of this phase is illustrated in Figure 4.
(3) Bridge classes in E_2 are used to create impure ersatz generic functions in E_4 and impure ersatz classes in E_3. The result of this phase is illustrated in Figure 5.
(4) Impure ersatz classes in E_3 are used to create pure ersatz generic functions in E_5 and pure ersatz classes in E_4. The result of this phase is illustrated in Figure 6.

The result of these phases is that the impure ersatz generic functions in environment E_4 can operate on the pure ersatz generic function in environment E_5 and on the pure ersatz classes in E_4. But they can also operate on impure ersatz objects, provided their call caches contain entries for the corresponding stamps. Filling the call caches is the purpose of our *satiation* technique [4].

4.5 Tying the knot

At the end of these four phases, we have fully functional impure ersatz generic functions in environment E_4, and fully functional impure classes in environment E_3. But we still do not have the cyclic graph of metaobjects that a functioning CLOS system requires.

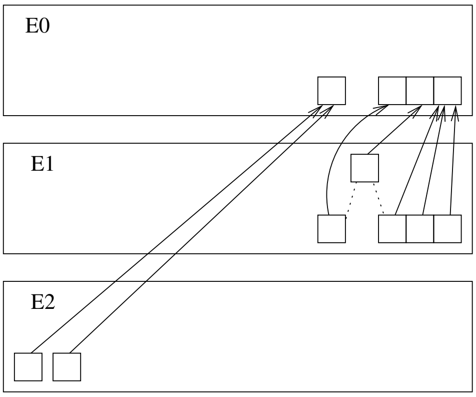


Figure 3: Phase 1.

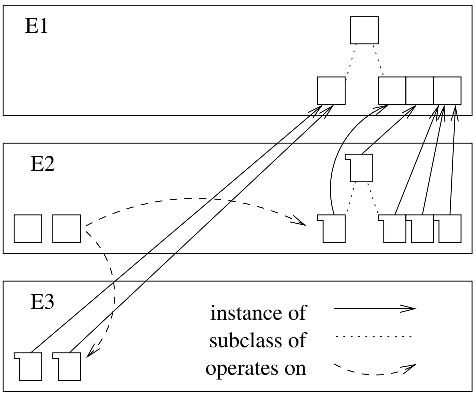


Figure 4: Phase 2.

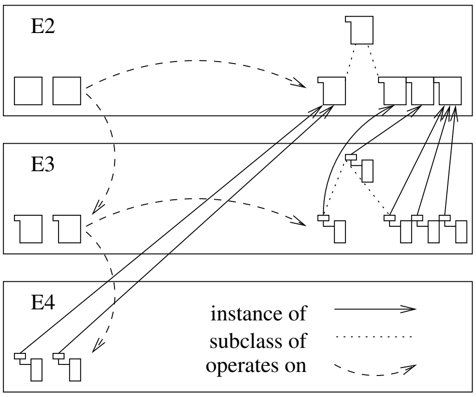


Figure 5: Phase 3.

Furthermore, there are still bridge generic functions that might be called in order to operate on our impure ersatz metaobjects.

To accomplish the conversion of this hierarchy of objects to a cyclic graph, we need to modify the `class` slot of the headers of each impure metaobject so that instead of referring to a bridge class, it refers to an impure ersatz class. This operation will transform every impure ersatz metaobject into a pure ersatz metaobject. However, there are a few more operations required to completely remove all references to bridge metaobjects:

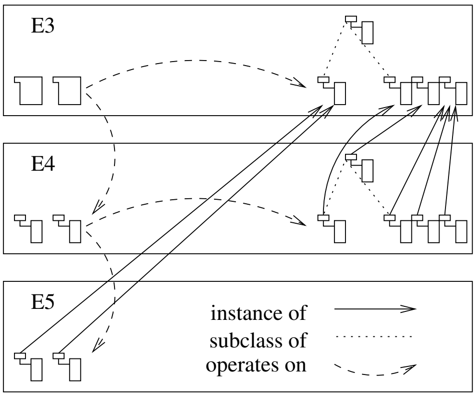


Figure 6: Phase 4.

- Each ersatz metaobject contains a list of the effective slot definition metaobjects of its class as the second word of the rack. In an impure ersatz metaobject, those effective slot definitions are bridge objects. Once the `class` field of the impure ersatz metaobject has been updated, this list must be updated to contain a reference to the list of the ersatz effective slot definitions from the new ersatz class.
- Each ersatz generic function contains a slot containing the method class of the methods on this generic function. In an impure ersatz generic function, this slot refers to a bridge class, so it must also be updated.

We must still find and update all impure ersatz metaobjects in the system. For classes and generic functions, this is trivial, as they are all reachable from the first-class global environment they are defined in. For other object types such as methods, slot-definitions, and method combinations, this is not the case. They must be found by a traversal of the class or generic function metaobject that they are part of. Such a traversal is straightforward.

Before the cyclic graph can be traversed and an isomorphic graph be generated in a native executable file, additional definitions must be loaded:

- Standard classes that are needed in order for the resulting native executable to be viable must be loaded. In particular, definitions of classes such as `symbol`, `package`, `cons`, `sequence`, `list`, `null`, `number`, `rational`, `integer`, and `fixnum` are needed in order for it to be possible to load compiled code into the executing image.
- Many standard functions are also needed, such as functions on packages, lists, hash tables, etc. Functions that operate on first-class global environments are needed as well.
- A simple version of the compiler must be loaded so that the resulting executable image can construct discriminating functions when definitions of generic functions and methods are loaded.

On the other hand, the garbage collector may not be needed in the initial executable image, though the data structures that the garbage collector works with must of course be present so that objects can be laid out in memory.

5 BENEFITS OF OUR TECHNIQUE

Appendix C of "The Art of the Metaobject Protocol" [1] (Living with Circularity) cites a number of ways in which their system handles circularity and avoids bootstrapping and metastability issues.

5.1 Bootstrapping benefits

The first bootstrapping problem that is mentioned is the fact that `standard-class` must exist before it can be created. Their solution is to create this class using some special-case mechanism. Our technique uses the version of `standard-class` in the preceding environment, so this problem is avoided altogether. As a result, we can freely modify the definition of `standard-class` and rerun the bootstrapping procedure. No special case has to be considered.

The second bootstrapping problem mentioned is that generic functions are used for method lookup, but these generic functions can not exist until a significant part of the protocol has been implemented. As an example, take the call to `ensure-class` made as a result of executing the expansion of a `defclass` form. By having `ensure-class` check for the special case when the argument is `standard-class` and by supplying a special function for creating instances of `standard-class` they avoid bootstrapping issues, simply because during bootstrapping, all classes created will be instances of `standard-class`. They also supply a special version of `finalize-inheritance` that checks for the metaclass `standard-class` and calls special-purpose code in this case. With our technique, no such special case is needed. All classes that are instantiated are fully operational in the preceding environment, as is the `finalize-inheritance` generic function.

5.2 Metastability benefits

The first example of a metastability problem mentioned in the book is that `slot-value` calls `slot-value-using-class` which then calls `slot-location` which in turn recursively calls `slot-value` on the class metaobject to access the slot metaobjects of the class. The authors propose to solve this problem by arranging for the function `slot-location` to check for the special argument `effective-slots` and return a predefined location. Our technique does not need this kind of special case, because the function `class-slots` does not call `slot-value` at all. It accesses the `effective-slots` slot directly, using its location. This location has been compiled in during the creation of the effective method and discriminating function for `class-slots`.

The final issue discussed in the book arises because the function `compute-discriminating-function` is also a generic function that can not be called with itself as an argument when a method has been added or removed from it. Again they solve the issue by a special case whereby a test is made to see whether the argument is a standard generic function (i.e. an instance of `standard-generic-function`) and if so, a special version of `compute-discriminating-function` which is not a generic function is called instead. With our technique, every generic function, `compute-discriminating-function` included, has a *call cache* that includes an effective method that is able to handle arguments that are direct instances of `standard-generic-function`. That call cache entry is not invalidated when `compute-discriminating-function` has new methods added to it, at least not when the methods added respect the restrictions of the metaobject protocol, i.e. that user code is not allowed to add methods that are applicable when given only standard objects as arguments.

5.3 Other benefits

In addition to solving the bootstrapping issues and the metastability issues given in the "The Art of the Metaobject Protocol" book, our technique has several additional benefits.

Since we begin the bootstrapping procedure by defining the classes and generic functions specified by the metaobject protocol, we are able to use the CLOS machinery to define system classes. In a system where CLOS is added late, many system classes must be defined using some other mechanism.

Furthermore, as already mentioned, our technique has great advantages to maintenance. There are no dependencies between CLOS code and other code that require duplication of information that must be kept synchronized when some code is modified.

6 CONCLUSIONS AND FUTURE WORK

We have described a technique for bootstrapping a Common Lisp system using an existing conforming Common Lisp system that is also supported by the library `closer-mop`. To our knowledge, no existing Common Lisp system is bootstrapped this way.

There are several advantages to our technique:

- The full Common Lisp language can be used in order to implement the system, including the compiler, thereby making the code more maintainable.
- By bootstrapping the MOP generic functions and the hierarchy of classes first, we eliminate the bootstrapping problems and metastability problems cited by the AMOP book [1].
- Also, by bootstrapping the MOP machinery first, we take advantage of it by using it to define all the standard system classes, thereby eliminating the need for special mechanisms for this purpose.
- The absence of special mechanisms that are needed in existing implementations for defining many aspects of the system itself, further contributes to the maintainability of our code.

Even though the technique outlined in this paper is known to work, many more aspects of the system need further work, including the bootstrapping technique itself, in order for a native executable to be generated:

- We must supply a (simple) code generator that translates intermediate code to native code. The amount of work required is fairly modest, and mainly consists of creating native code for memory operations such as `car` and `standard-instance-access`, for object allocation, and for simple arithmetic on fixnums.
- Interface code to the operating system must be supplied, in particular for input/output operations.

- We have yet to write the code that translates the host representation of the object graph into a native representation. Special care must be taken for object types that are imported from the host during bootstrapping, such as symbols, numbers, and `cons` cells.

However, we are in no hurry to create a native executable system. The moment we do, we lose a fairly good environment (namely the host Common Lisp system) for debugging our code. Instead, we plan to use the host environment for testing as many aspects of SICL as possible, and for creating support for better debugging capabilities, and only later create a native executable.

In terms of future work, there are still several optimization techniques that need to be implemented for the Cleavir compiler framework.

7 ACKNOWLEDGMENTS

We would like to thank David Murray for providing valuable feedback on early versions of this paper.

REFERENCES

[1] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991. ISBN 0262111586.

[2] Christophe Rhodes. Self-sustaining systems. chapter SBCL: A Sanely-Bootstrappable Common Lisp, pages 74–86. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-89274-8. doi: 10.1007/978-3-540-89275-5_5. URL http://dx.doi.org/10.1007/978-3-540-89275-5_5.

[3] Robert Strandh. Fast generic dispatch for common lisp. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, ILC '14, pages 89:89–89:96, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2931-6. doi: 10.1145/2635648.2635654. URL http://doi.acm.org/10.1145/2635648.2635654.

[4] Robert Strandh. Resolving metastability issues during bootstrapping. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, ILC '14, pages 103:103–103:106, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2931-6. doi: 10.1145/2635648.2635656. URL http://doi.acm.org/10.1145/2635648.2635656.

[5] Robert Strandh. First-class global environments in common lisp. In *Proceedings of the 8th European Lisp Symposium*, ELS '15, pages 79 – 86, April 2015. URL http://www.european-lisp-symposium.org/editions/2015/ELS2015.pdf.

Session VI: Lisp in Action

Tuesday, 2.4.2019

11:00–11:45	Nicolas Hafner: Shader Pipeline and Effect Encapsulation using CLOS
11:45–12:30	Robert P. Goldman and Ugur Kuter: Hierarchical Task Network Planning in Common Lisp
12:30–14:30	**Lunch**

Shader Pipeline and Effect Encapsulation using CLOS

Nicolas Hafner
Shirakumo.org
Zürich, Switzerland
shinmera@tymoon.eu

ABSTRACT

Modern real-time graphics make use of a lot of tricks in order to produce stunning visuals. Many of these tricks require separate rendering passes, as well as separate rendering logic for each pass. These passes are then combined in a variety of ways in order to produce a final image. The interaction between such a rendering pass and the objects it draws, as well as the interaction between multiple passes within a pipeline can become quite complex. Often times, in order to handle this complexity, the passes and objects are generalised, and the render logic is controlled almost entirely by either the object or the pass. We present a new method of representing objects, passes, and pipelines, which allows a modular encapsulation of effects and rendering behaviour, as well as object-oriented composition through inheritance. We make use of CLOS' multiple inheritance, multimethods, and standard method combination to form extensible protocols that allow this new method. We also make use of the MOP, in order to introduce additional metadata to classes.

CCS CONCEPTS

• **Software and its engineering → Object oriented architectures**; *Abstraction, modeling and modularity*; *Object oriented frameworks*; Compilers; • **Computing methodologies** → *Computer graphics*;

KEYWORDS

Common Lisp, OpenGL, GPU, CLOS, Object Orientation

ACM Reference Format:
Nicolas Hafner. 2019. Shader Pipeline and Effect Encapsulation using CLOS. In *Proceedings of the 12th European Lisp Symposium (ELS'19).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.5281/zenodo.2636508

1 INTRODUCTION

Modern graphics systems such as OpenGL Core and DirectX offer a lot of customisation to the programmer. Particularly, in order to render an image, they allow the programmer to supply code fragments (shaders) that are run directly on the GPU. These code fragments fill in steps of a fixed rendering pipeline that is executed on the GPU in order to transform vertex data into the pixels of an image. The pipeline for OpenGL is illustrated in Figure 1.

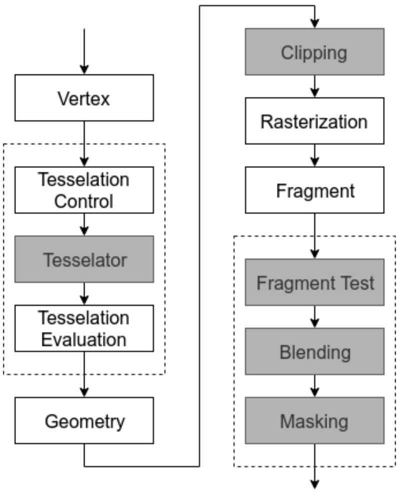


Figure 1: The stages of the OpenGL rendering pipeline. White boxes represent stages that can be customised with shader code.

For consistency, we refer to a step within the hardware rendering pipeline as a "stage", an invocation of the hardware pipeline as a "pass", and all invocations of the hardware pipeline to produce an image as a "frame".

Each of the customisable stages accepts only a single shader for each pass, making it difficult to separate, encapsulate, and ultimately combine behaviour. Furthermore, the steps required in order to change the shaders and shader inputs can be non-trivial and expensive to execute.

Managing this graphics state and the order of rendering can be very complicated for modern requirements. Rendering a frame often requires a multitude of passes, each with their own parameters and shared data. The rendering of each object within a pass can also differ, leading to even more state that needs to be correctly managed.

ELS'19, April 01–02 2019, Genova, Italy

ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2636508

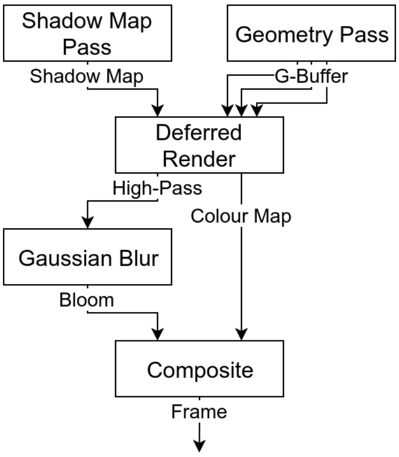


Figure 2: A sample frame pipeline with shadow mapping, deferred rendering, and bloom. Each box represents a pass and each edge a texture buffer.

This complexity results in a difficult challenge for modularity. We attempt to solve this challenge through several systems:

- A protocol for communicating information between a pass and the objects rendered within.
- A protocol to connect the inputs, outputs, and parameters of different passes.
- An algorithm to automatically allocate shared textures used as buffers between passes.

2 RELATED WORK

Courreges[1] presents an in-depth analysis of the rendering procedure employed by the modern, high-production game GTA V. It illustrates the many passes to produce a final image, as well as their data dependencies.

Harada et al.'s work on Forward+[2][3] also clearly illustrates the need for systems that support multi-pass rendering pipelines with complex data interaction schemes.

Gyrling[4] presents an overview of the techniques used to perform parallel rendering in Naughty Dog's private game engine. Individual stages within a pass, render passes of a frame, and multiple frame renderings are divided up into many small jobs that can run in parallel and are synchronised using counters on a shared structure. How rendering logic requirements are communicated is however not explained.

The case study of the Unity game engine by Messaoudi et al[5] shows the availability of a set of fixed rendering pipelines that can be customised to a very limited extent with custom shaders. These shaders must fit into Unity's existing lighting and overall rendering model. While Unity does allow building a custom pipeline via their Scriptable Rendering Pipeline[6], they do not seem to offer any specific encapsulation or modularity features.

The work by He et al.[7] introduces a framework for general encapsulation of shaders and their parameters into structures that minimise the overhead of changing GPU state while retaining the ability to dynamically compose shader parts. They do however not create a distinction between properties for rendering an object and for those for rendering an overall pass.

Foley et al.'s work on Spark[8] presents a high-level graph-based system for defining reusable and composable shader components. Their system shows a much more distanced view of the underlying graphics hardware than we attempt. Similar to He's work, they do not present a separation between object logic and pass logic.

In our previous work[9] we introduce a system to tie shader code to classes and to compose behaviour through inheritance. We make use of this system and extend it to allow further control over rendering behaviour in individual passes.

3 OVERVIEW

The system is composed of three distinct entities: shader objects, shader passes, and pipelines. Both shader objects and shader passes encapsulate rendering logic. The pipeline, on the other hand, represents the assembly of a complete frame, and only influences rendering in the sense that it tracks which passes to run in what order.

Due to the restrictions of the rendering pipeline of OpenGL, the combination of the render logic of objects and of passes is not trivial. We present a protocol to solve this difficulty in section 4.

In section 5 we present a protocol for representing input and output information of a pass and for connecting these inputs and outputs between passes together.

Due to the complex interactions between passes in a pipeline, allocation of intermediary buffer textures is error-prone and tedious. We present algorithms to automate this in section 6.

Finally, in section 7 we show the results of implementing a medium-sized pipeline using these techniques.

4 PASSES

A shader pass should encapsulate the logic for drawing objects in a certain way. In order to accomplish its task, the pass needs to be able to partially or fully control the shaders of objects. For instance, an object should still be able to have control over how its vertices are constructed and what materials and textures are applied to it. The shader pass should be able to make use of this information if it needs to, or discard it completely if unneeded.

In order to permit this amount of control, we make use of the shader composition capabilities that we presented in our previous work[9]. This technique allows us to combine pieces of OpenGL Shader Language (GLSL) code. On top of this shader combination sits a new protocol to control the interaction between shader passes and objects.

```
(defclass shader-pass ()
  ()
  (:metaclass shader-pass-class))

(defgeneric register-object-for-pass (shader-pass object))
(defgeneric shader-program-for-pass (shader-pass object))
(defgeneric make-pass-shader-program (shader-pass object))
(defgeneric coerce-pass-shader (shader-pass object stage))
(defgeneric render-with (shader-pass object))
(defgeneric render (object target))
```

Listing 1: The protocol for shader passes.

For any object that should be rendered using a given pass, first `register-object-for-pass` must be called. This call allows the pass to prepare the shader program that will be used during rendering for this object. A shader program is an OpenGL resource that compiles the shaders for the stages of a pass together. Using `shader-program-for-pass` this program can then later be retrieved. The shader program is important for objects to access, as it allows them to set values for "uniform" variables that are used in the shader code.

`register-object-for-pass` calls `make-pass-shader-program` to compute this program, and registers it internally in the shader pass so that it can be retrieved later. If the shader pass would like to retain complete control over how each object is drawn, it would forego this call and instead generate its own shader program. Doing so is especially useful for post-processing effects that don't render any objects at all, and instead simply operate on textures that are output by previous passes.

`make-pass-shader-program` gathers all the shader sources for a program using `coerce-pass-shader` and then generates a representation for an OpenGL shader program. This representation also includes additional information such as the data buffers used.

`coerce-pass-shader` computes the effective shader source for a particular shader stage or type. Typically this computation simply involves the combination of the shader sources of both the shader pass and of the object, for the given shader type. We perform this combination using the same parsing and code walking strategy as we described in our previous work.[9]

Objects are typically rendered using the `render` function. Users are encouraged to add methods that specialise the behaviour and perform necessary setup, as well as to perform the final draw call for their custom object classes. For instance, a very primitive class could look as shown in Listing 2.

This function is fine for allowing the object control over the behaviour. However, the pass cannot exert the same amount of control, due to the generic function's argument precedence. For example, if an `:around` method specialised on the object exists, a pass would not have any way of preventing it from firing, as its own `:around` method would be executed afterwards. We thus introduce another function with inverted argument order that is called first.

```
(defclass simple-object ()
  ((vertex-array :accessor vertex-array)
  (:metaclass shader-class))

(defmethod render ((oject simple-object) target)
  (let ((vao (vertex-array object)))
    (gl:bind-vertex-array (gl-name vao))
    (%gl:draw-elements :triangles (size vao) :unsigned-int 0)))

(defmethod render :before ((object simple-object)
                           (pass shader-pass))
  (let ((program (shader-program-for-pass pass object)))
    (setf (uniform program "projection_matrix")
          (projection-matrix))))
```

Listing 2: A simple object class and its render methods. The first method tells OpenGL to render a list of vertices. The second method sends the projection matrix to the GPU via a uniform variable.

`render-with` thus exists mostly as an entry point to allow shader passes greater control over the rendering. Typically it will simply defer to `render`. This inversion is very important for shader passes that take all control away from the objects.

This protocol thus allows a great amount of control both over the effective shader code used to render an object, as well as the behaviour leading up towards the actual draw call for an object.

The overall protocol thus only invokes overhead for the dispatch of the `render-with`, `render`, and `shader-program-for-pass` generic functions for each rendered object per frame. All other functions are only invoked during loading, or in exceptional situations. In our use of the system so far we have not found this to impact performance significantly.

5 PIPELINES

Shader passes not only retain information about how objects are drawn, but also about the input and output textures that a pass interacts with. For this reason, each shader pass is also a node in a graph, with distinct input and output ports.

```
(defclass deferred-render (shader-pass)
  ((position-map :port-type input)
   (normal-map :port-type input)
   (albedo-map :port-type input)
   (color :port-type output))
  (:metaclass shader-pass-class)
```

Listing 3: An outline of a deferred rendering pass, taking position-, normal-, and albedo-map textures as input, producing a single color texture as output.

An example of such a pass is illustrated in Listing 3. The ports are modelled as slots of the class that carry additional metadata, requiring a new metaclass. A slot with a port-type will not only hold a texture for its value, but also retain information about how it is connected to other passes. Using slots in this manner also allows us to inherit ports from other classes, and thus combine behaviour alongside the shader source code.

For the sake of brevity and ease of explanation, we have left out the details of texture constraints in these code samples. Nevertheless, constraining the features of textures tied to the ports is important, and we discuss the technique for dealing with that in section 6.

```
(defclass shadow-render (shader-pass)
  ((shadow-map :port-type input)
   (color :port-type output))
  (:metaclass shader-pass-class))
```

Listing 4: An outline of a shadow rendering pass, taking into account the information from a shadow-map to render shadows onto the output color texture.

```
(defclass high-pass-render (shader-pass)
  ((high-pass :port-type output))
  (:metaclass shader-pass-class))
```

Listing 5: An outline of a high-pass renderer, which splices off colours of a high intensity into a high-pass output texture.

For instance, the "Deferred Render" pass shown in Figure 2 is created by combining the passes illustrated in Listing 3, Listing 4, and Listing 5 through inheritance.

When assembling a pipeline, we can then connect these ports together, in order to dictate how the various output textures generated by a pass are used as inputs for other passes. As an example, the pipeline from Figure 2 is created in Listing 6.

```
(let ((pipeline (make-pipeline))
      (shadow (make-instance 'shadow-map-pass))
      (geometry (make-instance 'geometry-pass))
      (deferred (make-instance 'deferred+shadow-pass))
      (blur (make-instance 'gaussian-blur-pass))
      (composite (make-instance 'composite-pass)))
  (connect (port shadow 'shadow-map)
           (port deferred 'shadow-map) pipeline)
  (connect (port geometry 'position-map)
           (port deferred 'position-map) pipeline)
  (connect (port geometry 'normal-map)
           (port deferred 'normal-map) pipeline)
  (connect (port geometry 'albedo-map)
           (port deferred 'albedo-map) pipeline)
  (connect (port deferred 'high-pass)
           (port blur 'previous-pass) pipeline)
  (connect (port deferred 'color)
           (port composite 'color) pipeline)
  (connect (port blur 'color)
           (port composite 'bloom) pipeline)
  (prepare pipeline))
```

Listing 6: An assembly of the pipeline shown in Figure 2.

The pipeline object itself retains information on the order in which the passes should be executed, and keeps track of the textures that are allocated in order to run the passes. How this information is computed is described in section 6.

The procedure to actually render objects with this pipeline technique is then as follows:

1. Create a pipeline object and instances of the desired shader passes.
2. Connect the ports of the shader passes.
3. Prepare the pipeline to allocate the needed resources.
4. Create instances of the desired objects.
5. Register each object with each pass.
6. Call `render` on the pipeline with a collection of all objects to draw.
7. The output texture of the output port of the last shader pass in the pipeline will contain the finished frame.

The completed frame can then be blitted onto the screen, or be used as the input for another computation.

6 ALLOCATION

In order to prepare the shader pipeline, several resources need to be allocated. Each shader stage needs a "framebuffer," an OpenGL resource that allows one to render to off-screen textures. These framebuffers then need to have the required textures to render allocated as well. As each input and output from a pass can specify constraints on the features of the texture, these constraints must be matched up for any connecting edges as well. Finally, in order to minimise memory usage, we would like to re-use textures where possible.

Thus, the allocation proceeds in three phases: reconciling texture constraints on edges between passes, computing how textures are shared between passes, and finally constructing all the necessary resources with the previously gathered information.

6.1 Constraint Merging

OpenGL textures include a massive amount of information[?][?]. When two ports are connected that specify different constraints on the texture properties, a join must be performed. A wide range of the texture property values are fundamentally incompatible, meaning that a lot of the logic can simply error. However, other options require more complicated joining logic. For simplicity and brevity, we will focus on the join operator for a single texture property here: the internal format. The internal format is arguably the most important property. This property specifies how many colour channels the texture has, how many bits of precision each channel has, which format each channel has, as well as whether the texture is compressed or has sRGB gamma normalisation applied.

The list of specified texture formats is quite large[?]. Unfortunately OpenGL does not give us an interface to handle these formats in a way that lets us pick the individual features easily. Instead, each format is represented by a constant whose value has no relation to the features that format includes. This representation means that we first need to destructure each format name into a list of features:

- `R`, `G`, `B`, `A` How many bits to use for each channel, and the format of the channel (normalised, float, integer, unsigned integer).

- `depth` How many bits to use for the depth channel.
- `stencil` How many bits to use for the stencil channel.
- `shared` Whether bits are shared across the channels, and if so how many.
- `features` Whether the format has compression, sRGB, RGTC, BPTC, SNORM, or UNORM features.

Once the texture format specifications are destructured, we can perform a join as follows.

```
1. If the features and sharing are not the same, a join is impossible.
2. The features list of the output spec is set to the same as either of the specs.
3. If both include a depth component:
   3.1. The depth feature is set to the join of both.
   3.2. If either include a stencil feature, the stencil feature is set to the join of both.
4. If both include a stencil component:
   4.1. The stencil feature is set to the join of both.
5. If neither include a depth component:
   5.1. The R feature is set to the join of both.
   5.2. The G feature is set to the join of both.
   5.3. The B feature is set to the join of both.
   5.4. The A feature is set to the join of both.
6. Otherwise a join is impossible.
```

Wherein the "join of both" is computed as the join of two channel formats as follows.

```
1. If both include the format:
   1.1. If the channel format is not the same, a join is impossible.
   1.2. The channel bit depth is set to the maximum of both.
2. If one includes the format, that format is returned.
3. Otherwise, the absence of the channel is indicated.
```

If the join is successful, we then re-encode the texture format specification into OpenGL's constant and use this constant in the real texture specification. Note that this join could produce texture format specifications that are not legal according to the OpenGL specification. However, this can only occur if one of the specifications to join is already illegal. We thus deem it unnecessary to handle such cases.

6.2 Port Allocation

In compilers, allocation of a graph of variables with use-relations is typically handled with a graph colouring algorithm. However, since we represent our graph in a different fashion than usual, with nodes having multiple distinct ports on which edges are connected, we devised a different kind of colouring algorithm to maximise texture sharing.

Given a set of nodes the allocation algorithm proceeds as follows.

```
1. The set of unique texture specifications is computed by joining each port's texture specification with every other and comparing for equality.
2. For each unique texture specification T:
   2.1. The nodes are sorted topologically.
   2.2. The number of colours is set to 0.
   2.3. For each node N:
      2.3.1. For each output port P of N:
         2.3.1.1. If P's texture specification is joinable with T...
         2.3.1.2. The number of colours is increased
   2.4. An array is allocated to fit the number of colours. Each index of the array represents a colour and each value at the index whether the colour is currently available or unavailable.
   2.5. For each node N in reverse order:
      2.5.1. For each input port P of N:
         2.5.1.1. For each neighbour port O of P:
            2.5.1.1.1. If O's texture specification is joinable with T...
            2.5.1.1.2. and O does not yet have a colour...
            2.5.1.1.3. The first available colour is assigned to O.
            2.5.1.1.4. This colour is marked as unavailable.
      2.5.2. For each non-input port P of N:
         2.5.2.1. If P's texture specification is joinable with T...
         2.5.2.2. and P does not yet have a colour...
         2.5.2.3. The first available colour is assigned to P.
         2.5.2.4. This colour is marked as unavailable.
      2.5.3. For each port P of N:
         2.5.3.1. If P's texture specification is joinable with T...
         2.5.3.2. and P has a colour...
         2.5.3.3. P's colour is marked as available.
```

In other words, the algorithm proceeds backwards from the last node in the graph, marking output ports of predecessors with unique colours, then marking unconnected ports with unique colours, and finally marking all colours at the node's own output ports as available again. We repeat this process for each unique texture specification, each time ensuring we only consider ports that share that texture specification.

This algorithm is by no means efficient, but since pipeline allocation only has to happen during loading phases, we currently do not consider this to be a big problem. We also have not performed any analysis as to whether the algorithm produces optimal allocation results in every case.

Unfortunately printed documents cannot yet display animations, so illustrating the algorithm in motion is not possible directly in the document. However, a brief animation of the pipeline illustrated in Figure 2 is available online:

https://raw.githubusercontent.com/Shinmera/talks/master/els2019-shader-pipeline/pipeline-allocation.gif

With this algorithm we reduce the number of necessary textures to allocate down to five. With a primitive allocation of one texture per output, we would instead end up with seven textures. Note that the shadow map output is not assigned, as it requires a different type of texture from the rest – a depth texture. This texture would be allocated in a second pass.

7 PROOF OF CONCEPT

As a proof of concept we have implemented these protocols and mechanisms, as well as the passes shown in Figure 2. As the code required to do so is quite lengthy and involved, and the graphics techniques used are beyond the scope of this paper, we will omit the code. The relevant outline of the passes and the pipeline involved has already been shown in section 5.

Nevertheless, you can find the complete code online at the following pages:
https://github.com/Shirakumo/trial/blob/b889d50/deferred.lisp
https://github.com/Shirakumo/trial/blob/b889d50/hdr.lisp
https://github.com/Shirakumo/trial/blob/b889d50/shadow-map.lisp
https://github.com/Shirakumo/trial/blob/b889d50/workbench.lisp

Figure 3, Figure 4, Figure 5, Figure 6, and Figure 7 show the output textures produced by the various stages for a simple scene.

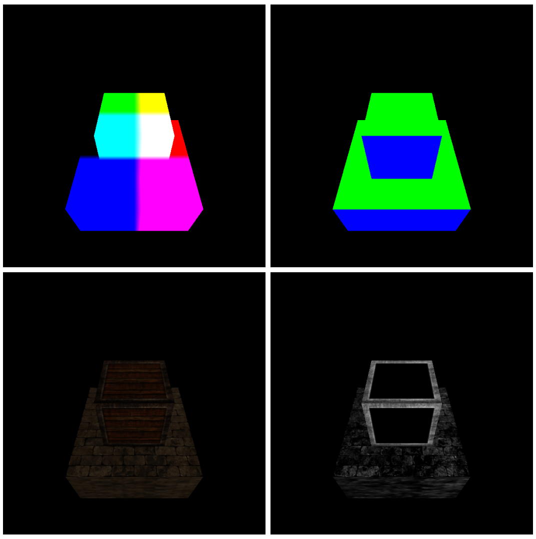


Figure 3: The output of the geometry pass, producing a "G-buffer" composed out of a position- (top left), normal- (top right), albedo- (bottom left), and specular-map (bottom right).

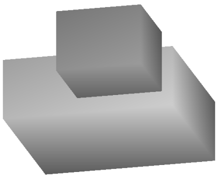


Figure 4: The depth map produced by the shadow map pass. It shows the distance of surfaces from the point of view of the primary light source.

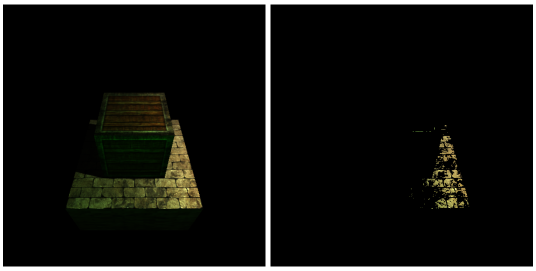


Figure 5: The color map (left) and the high-pass map (right) that is output by the deferred render pass. It combines the information from the G-buffer from Figure 3, the shadow-map from Figure 4, and information for lights to render an image. Note that due to gamma adjustments the image appears quite dark.

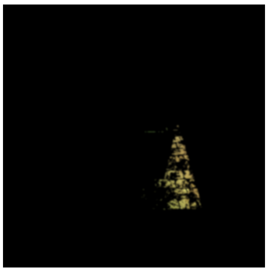


Figure 6: The blurred high-pass from Figure 5 produced by the gaussian blur pass. This is used to produce an effect called bloom in the final composite pass.

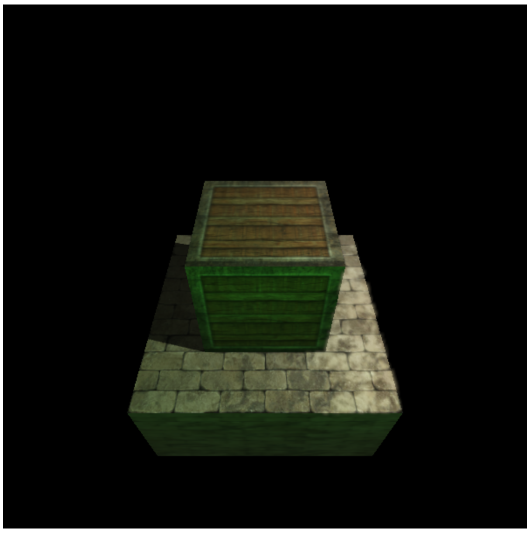


Figure 7: The final output frame that is produced by adding together the bloom texture from Figure 6 and the standard colour texture from Figure 5. Gamma adjustment and High Dynamic Range reduction are used for colour normalisation.

8 CONCLUSION

We have presented a system to handle the separation between objects that should be rendered, and the way in which they should be rendered. By using a protocol for the interaction between such objects and passes, we are capable of controlling and combining a variety of rendering behaviours. With an additional protocol and set of algorithms we also present a high-level view of rendering pipelines, allowing users to conveniently compose effects to produce complicated scenes.

These protocols and algorithms are largely generic enough that they could also be adapted for use in other languages and systems. However, we believe that the Common Lisp Object System gives us a set of very convenient tools to make everything feel more natural and more integrated with the rest of the system than other environments would allow us to do.

Finally, we have demonstrated the capability of this system by constructing a real-time rendering system out of individual, modular pieces of code.

9 FURTHER WORK

Currently there exist several restrictions in our framework.

Most severe is that there is no standardised protocol to communicate capabilities and data between objects and passes. This restriction means that, for instance, an object that does not possess texturing has no way of communicating this to a shader pass that might need to render it. Currently this problem is partially solved by introducing a mandatory superclass that objects need to be a subtype of, if they want to be renderable under a given pass. This superclass then loosely defines the interaction. However, this additional class does not solve issues of compatibility of shader code used by the object and by the shader pass. In order to properly resolve this restriction, we currently see two required features:

Shader source merging must be aware of uses-relations, meaning that the code walker must track where variables, functions, and types are used and reorder the definitions and declarations in such a way that the dependencies are fulfilled. In GLSL, variables, functions, and types cannot be referenced before they are declared, requiring this reordering step. With this problem solved, shader code could be written in such a way to be more amiable towards combination and manipulation by external code.

GLSL's limited expressiveness makes it very difficult to figure out relations between code and how to properly combine pieces. A more high-level language, as is used in other frameworks like Spark[8] or Shader Components[7], should be introduced to allow a more convenient way to declare and write shaders, objects, and passes. With a higher-level language, a compiler could more easily figure out how to combine features and reconcile differences between pieces of code.

Another restriction is that there is currently no way to encapsulate GLSL code other than associating it with a class, which is a rather heavy-weight operation. It would be far more useful to introduce a library mechanism that allows the definition of standalone shader functions, which can then be required by shader classes.

10 ACKNOWLEDGEMENTS

I would like to thank Michal "phoe" Herda, Robert Strandh, Michael Reis, and all the people that will review this paper. Your name could be here!

11 IMPLEMENTATION

An implementation of the proposed system can be found at
https://github.com/Shirakumo/trial/blob/b889d50/shader-pass.lisp
https://github.com/Shirakumo/trial/blob/b889d50/pipeline.lisp
https://github.com/Shinmera/flow

REFERENCES

[1] Adrian Courreges. Gta v-graphics study. http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/, 2015. [Online; accessed 2019.01.24].

[2] Takahiro Harada, Jay McKee, and Jason C Yang. Forward+: Bringing deferred lighting to the next level. 2012.

[3] McKee, Jay Harada, Takahiro. Forward rendering pipeline for modern gpus. https://www.gdcvault.com/play/1016435/Forward-Rendering-Pipeline-for-Modern, 2012. [Online; accessed 2019.01.24].

[4] Christian Gyrling. Parallelizing the naughty dog engine using fibers. https://www.gdcvault.com/play/1022186/Parallelizing-the-Naughty-Dog-Engine, 2015. [Online; accessed 2019.01.24].

[5] Farouk Messaoudi, Gwendal Simon, and Adlen Ksentini. Dissecting games engines: The case of unity3d. In *Proceedings of the 2015 International Workshop on Network and Systems Support for Games*, page 4. IEEE Press, 2015.

[6] Scriptable render pipeline. https://docs.unity3d.com/Manual/ScriptableRenderPipeline.html, 2018. [Online; accessed 2019.01.24].

[7] Yong He, Tim Foley, Teguh Hofstee, Haomin Long, and Kayvon Fatahalian. Shader components: modular and high performance shader development. *ACM Transactions on Graphics (TOG)*, 36(4):100, 2017.

[8] Tim Foley and Pat Hanrahan. *Spark: modular, composable shaders for graphics hardware*, volume 30. ACM, 2011.

[9] Nicolas Hafner. Object oriented shader composition using clos. In *11 th European Lisp Symposium*, page 80, 2018.

Hierarchical Task Network Planning in Common Lisp: the case of SHOP3

Robert P. Goldman and Ugur Kuter
rpgoldman@sift.net
ukuter@sift.net
SIFT, LLC
Minneapolis, MN

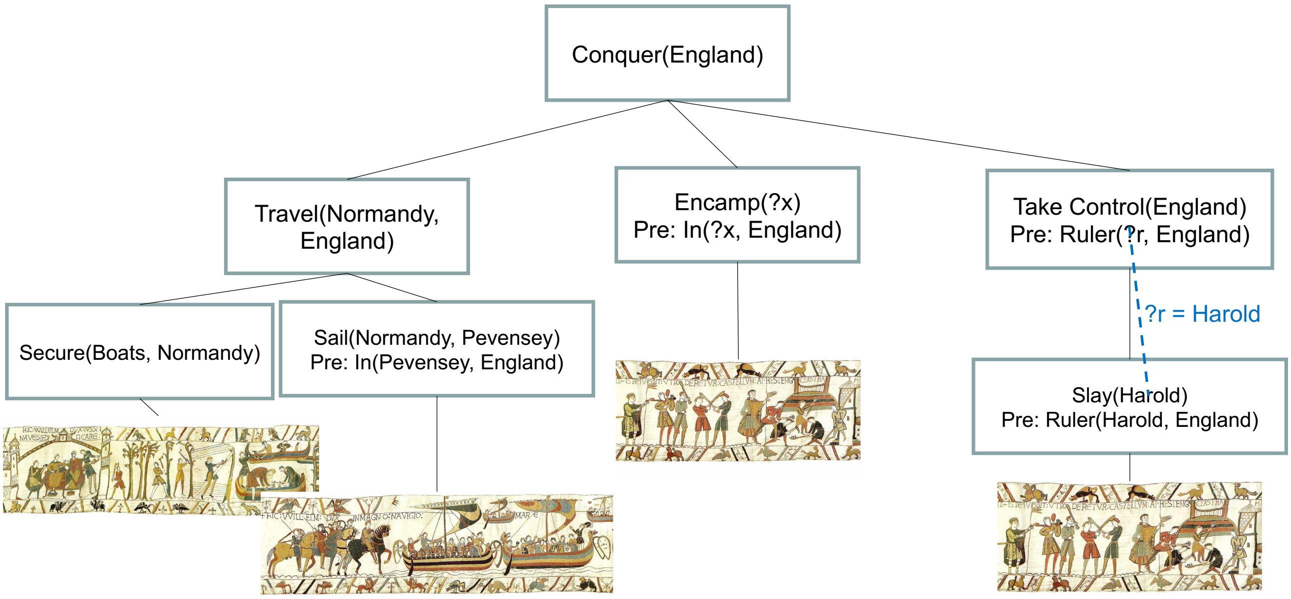


Figure 1: Hierarchical Plan for Conquest of England [11]

ABSTRACT

This paper describes the use of Common Lisp (CL) to develop a new version of the Hierarchical Task Network (HTN) planner, SHOP2, first developed at the University of Maryland (UMD). which we are dubbing SHOP3. We will describe ways in which we have profited from language features offered by CL to build a more solid, efficient, yet flexible planning system, review lessons learned and suggest some best practices. SHOP3 is an open source tool made publicly available by SIFT, hosted on GitHub. It is freely available for use under the terms of the Mozilla Public License. CL provided a good foundation for extensibility and refactoring of the SHOP2 planner to support both more flexibility and extensibility and, at the same time, more usability as a practical tool. By comparison, the Java version of SHOP2 was rigid, and rapidly abandoned after the original developer left UMD. By then it already lagged behind the CL version in terms of features because of Java's rigidity, and poor support for symbolic programming.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; **Planning and scheduling**; **Planning for deterministic actions**; **Planning with abstraction and generalization**; *Logic programming and answer set programming*; • **Information systems** → *Decision support systems.*

KEYWORDS

Common Lisp, HTN planning, AI Planning, symbolic reasoning, software abstraction, software modeling, software flexibility

ACM Reference Format:

Robert P. Goldman and Ugur Kuter. 2019. Hierarchical Task Network Planning in Common Lisp: the case of SHOP3. In *Proceedings of ELS '19: European Lisp Symposium (ELS '19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.2633324

ELS '19, April 01–02, 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2633324

1 INTRODUCTION

AI planning is the subfield of artificial intelligence (AI) that aims at automating processes of *means-ends reasoning*. In general, AI planning is the problem of finding a sequence of actions that, executed in a specified initial state, will reach a goal state. This is a problem with applications to diverse areas including manufacturing, autonomous space and deep sea exploration, medical treatment, and military operations, to name just a few.

We describe our ongoing work, over the past two decades, on the Shop3 planning system, a Hierarchical Task Network (HTN) planner. Shop3 is based on Shop2, which was originally developed at the University of Maryland (UMD). In our work we have made extensive use of Common Lisp (CL) features to extend and harden the Shop2 code base over the years, culminating in the release this month of its successor, Shop3.

Our company, SIFT, (|www.sift.net|) is a for-profit research lab, employing approximately 35 people, with 13 Ph.D.s. We are in our 20th year, and have offices in Minneapolis, and the Boston, Washington, and San Diego metropolitan areas. We do contract research on AI, Computer Security, Formal Methods, and Human-Computer Interaction, primarily for the US Federal Government. Funding sources include DARPA, NASA, Department of Energy, Air Force Research Laboratories, Office of Naval Reserch, etc.

CL provided a good foundation for extensibility and refactoring of the Shop2 planner and, at the same time, more usability as a practical tool. By comparison, the Java version of Shop2 was rigid, and rapidly abandoned after the original developer left UMD. By then it already lagged behind the CL version in terms of features because of Java’s rigidity, and poor support for symbolic programming.

We describe ways in which we have profited from language features offered by Common Lisp to build a more solid, efficient, yet flexible planning system, review lessons learned and suggest some best practices. Shop2, and in turn, Shop3 are based on a powerful Prolog-style logic programming capability for both logical inference and planning. We will discuss how well-suited CL is for this kind of symbolic computation in an AI planning system.

Shop3 is available and regression-tested in several CL platforms, including Allegro CL (https://franz.com/products/allegro-common-lisp/), SBCL (www.sbcl.org), and CCL (https://ccl.clozure.com/). Shop3 is an open source library that is publicly available through SIFT. The final version of Shop2 is currently available for download from Sourceforge. Shop3 is available from GitHub, through the “shop-planner” group: |https://github.com/shop-planner/|. It is available for use according to the Mozilla Public License.

We begin the paper with some background material, first a description of the problem of AI planning, and more specifically, the approach of HTN planning. As part of our discussion of HTN planning, we will review the original Shop2 system, and its history. We will describe a number of challenges we faced in applying Shop2 to problem domains, making it extensible in fundamental ways, and also more efficient. We will describe how we have addressed these problems and specifically how features of Common Lisp have aided us. We will conclude with some best practices for use of Common Lisp for this kind of symbolic computing, and some holes we would like to see filled to improve the utility of the language.

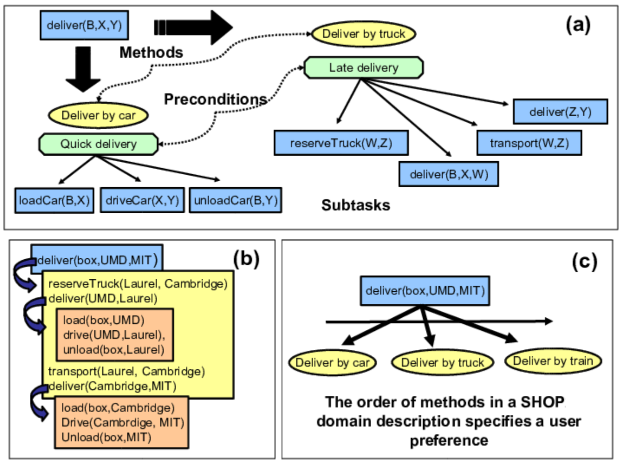


Figure 2: Delivery planning example.

2 PRELIMINARIES: AUTOMATED PLANNING

We provide a brief introduction to AI planning, largely following the discussion of Ghallab, Nau, and Traverso [9]. Formally, a “classical” planning problem involves a *state-transition system*, $\Sigma =< S, A, \gamma >$ where S is a set of states, A is a set of actions, and γ is a state transition function, $\gamma : S \times A \rightarrow S$. A *planning problem* is a triple $P =< s_0, \Sigma, G >$ where s_0 is the initial state, Σ is the state-transition system, and $G \subseteq S$ is the *goal*. A *plan*, or solution, is a sequence of actions $\pi = a_0 \ldots a_n$ such that $\gamma(\gamma(\gamma(s_0, a_0), a_1) \ldots, a_n) \in G$. In classical planning, the state space is factored into a set of propositions, and every state $s \in S$ is a complete truth assignment to the set of propositions. For example, in a planning problem with a robot, `r`, that moves between three workstations, `w1`, `w2` and `w3`, a state would be a truth assignment to the three propositions `(at r w1)`, `(at r w2)`, and `(at r w3)`. In this case, the three are mutually exclusive, so that there are only three states.

Actions are triples, $a = \langle \text{name}(a), \text{prec}(a), \text{eff}(a) \rangle$ where the name is an arbitrary designator, the *preconditions*, $\text{prec}(a)$ specify conditions under which the action can be executed (making γ a partial function), and the *effects*, $\text{eff}(a)$ are a factored representation of the transition function. For example, the action `(move w1 w2)` would have the preconditions `(at r w1)` and the effects `(and (not (at r w1)) (at r w2))`. In the interests of convenience, one describes actions using action *schemas* (macros), for example:

```
(:action (move ?r - robot ?w0 ?w1 - workstation)
   :precondition (at ?r ?w0)
   :effect (and (not (at ?r ?w0)) (at ?r ?w1)))
```

The standard language for describing planning *domains* (a set of action descriptions and ancillary information) and problems is the Planner Domain Definition Language (PDDL)[5, 8, 17]. PDDL was developed to facilitate the International Planning Competition (IPC) [12, 17], held in conjunction with the International Conference on Automated Planning and Scheduling (ICAPS). The `move` action definition above is written in PDDL. There are a wide selection of publicly available benchmark problems that have been used in past

IPCs. PDDL has multiple sub-languages of increasing expressive power. We will return to this issue later.

At its core, planning is a graph search problem, and as such might seem suitable for methods such as Dijkstra's algorithm, no worse than $O(n^2)$. However, for single source shortest path (SSP), n is the size of the graph. For planning the input is extremely compressed, and the state space is exponential in the size of the input, so conventional SSP algorithms are, in practice, exponential. Indeed, the planning problem is very hard: intractable even when the expressive power (domain and problem) are tightly restricted. See, for example, Bäckström, *et al.* [1]. Another way of thinking about AI planning is that it involves synthesizing an open loop controller for goal reachability.

The IPC has spurred a great deal of advancement in classical planning since its inception. Planners can now solve very large classical planning problems, and there are a diversity of different planning methods. The most successful approaches have been based on heuristic search, reduction to propositional satisfiability (SAT), and local search/constraint satisfaction. However, we cannot overemphasize the expressive limitations of these classical planners, which makes them unusable for most practical applications. In the next sections, we will describe hierarchical task network (HTN) planning, sometimes also referred to as "decomposition planning," which has radically more expressive power.

3 HIERARCHICAL TASK NETWORK PLANNING

Hierarchical Task Network (HTN) planning addresses many of the problems of classical, "first principles" planning as described above. Classical planning, for example, is limited to goals of *achievement*. For example, jogging around a track cannot easily be captured as goal achievement, because the goal state is to end up in the starting position. Most classical planners cannot effectively plan for multiple agents, because they only optimize for either plan length or cost minimality for additive, context independent action costs. First principles planners also find plans unconstrained by considerations such as standard operating procedures. Related to this, a first principles planner requires a causal theory (preconditions and effects), whereas an HTN planner can do actions "just because": e.g., part of the protocol for treating stroke victims involves giving aspirin, although we do not have a clear causal theory of its effectiveness.

All of these concerns can be addressed by HTN planning, and the University of Maryland's Shop2 planner was the most mature and complete HTN implementation, so when SIFT was looking for a planner for applications, that is where we started. Unlike a first principles planner, an HTN planner produces a sequence of actions that perform some activity or *task*, instead of finding a path to a goal state. An HTN planning domain includes a set of planning *operators* (actions) and *methods*, each of which is a prescription for how to decompose a task into its *subtasks* (smaller tasks). The description of a planning problem contains an initial state as in classical planning. Instead of a goal formula, however, there is a partially-ordered set of tasks to accomplish.

Planning proceeds by decomposing tasks recursively into subtasks, until *primitive tasks*, which can be performed directly using the planning operators, are reached. For each task, the planner chooses an applicable method, instantiates it to decompose the task into subtasks, and then chooses and instantiates other methods to decompose the subtasks even further. If the constraints on the subtasks or the interactions among them prevent the plan from being feasible, the planner will backtrack and try other methods.

Shop2 is an HTN planner that generates actions in the order they will be executed in the world. Its backtracking search considers the methods applicable to the same task in the order they are specified in the knowledge base given to the planner. This feature of the planner allows for specifying *user preferences* among such methods, and therefore, among the solutions that can be generated using those methods. For example, Figure 2(c) shows a possible user preference among the three methods for the task of delivering a box from the University of Maryland (UMD) to MIT.

Consider a Delivery Domain, in which the task is to deliver a box from one location to another. Figure 2(a) shows two Shop2 methods for this task: *delivering by car*, and *delivering by truck*. Delivering by car involves the subtasks of loading the box to the car, driving the car to the destination location, and unloading the box at the destination. Note that each method's preconditions are used to determine whether or not the method is applicable: thus in Figure 2(a), the *deliver by car* method is only applicable if the delivery is to be a fast one, and the *deliver by truck* method is only applicable if it is to be a slow one. Now, consider the task of delivering a box from the UMD to MIT and suppose we do not care about a fast delivery. Then, the *deliver by car* method is not applicable, and we choose the *deliver by truck* method. As shown in Figure 2(b), this decomposes the task into the following subtasks: *(1)* reserve a truck from the delivery center at Laurel, Maryland to the center at Cambridge, Massachusetts, *(2)* deliver the box from the University of Maryland to Laurel, *(3)* drive the truck from Laurel to Cambridge, and *(4)* deliver the box from Cambridge to MIT. For the two delivery subtasks produced by this decomposition, we must again consider our delivery methods for further decomposing them until we do not have any other task to decompose.

During planning, the planner evaluates the preconditions of the operators and methods with respect to the world state it maintains locally. It is assumed that planner has all the required information in its local state in order to evaluate these preconditions. For example, in the delivery example, it is assumed that the planner knows all the distances between the any initial and final locations so that it can determine how long a truck will be reserved for a delivery task.

4 SIMPLE HIERARCHICAL ORDERED PLANNER (SHOP2)

The original algorithm for Shop2 is based on recursive depth-first search, and has as its key data structures (1) a stack of open tasks, (2) a state object that supports mutation and rollback, (3) a plan list and, (4) a set of variable bindings. A pseudocode version of the algorithm is given in Algorithms 1, 2 and 3. These are taken originally from the Shop2 Manual, but the control flow has been simplified, and the management of variable bindings, which was suppressed in the original, has been included.

Some notes on the algorithm: the **choose** operator in the pseudocode represents nondeterministic choice, implemented as search.

Algorithm 1 Planning algorithm

```
1: procedure PLAN($S$, $t$)                                ▷ state, task
2:     return FIND PLANS($S$, $\{t\}$, $\emptyset$)
3: end procedure
```

In practice, Shop2 uses depth-first search[1] so, for example, in Algorithm 3, line 8, $\boxed{\textbf{choose}\ b \in B}$, what is done is to attempt to proceed with b bound to the first element of B, and if this fails, to try again with the next element of B until a solution is found or B is exhausted, at which time this line of code fails. In practice there will be something like an `OR` in the code, and there will be a non-tail recursive call, leaving a new frame on the stack. As one would expect, difficult search problems – or even degenerate problems involving no search, but requiring long plans, cause stack exhaustion.

Note also that this involves being able to roll back state changes that come from operator application when the system backtracks to an earlier point in the search and consider different alternative plans. This backtracking is not done on the stack: instead the state objects are built out of an original state and set of incremental updates, which are labeled. When the system backtracks, it undoes changes (by removing updates from the state object), using the labels in the update sequence.

The **query()** function in the following is all-solutions Prolog-style database retrieval (backward chaining). For those familiar with Prolog, this is similar to a **bagof** query. The return is a list of binding sets. Each binding set associates some set of variables with values (which may be other variables), by unification. The **apply()** function returns a new expression that results from applying a set of variable bindings to an original expression.

We can see from this that the key operations are (1) **tree search**, (2) **operator application**, (3) **task reduction**, (4) **retrieval** from the state database, and (5) **unification**. All of these operations involve symbolic computation.

4.1 Issues with Shop2

SIFT has been working with Shop for approximately 15 years now; we chose it because it was the only open source HTN planner available, it was relatively efficient and well-tested, and it performed well in the 2002 IPC – the last IPC in which HTN planners competed [16]. Shop2 also has been used in a number of planning applications, including recently at SIFT for Air Operations and UAV planning [14, 15, 18, 19], cyber security [3], cyber-physical systems [10], planning for synthetic biology experiments [26], and software vulnerability analysis [13], to name a few. For an earlier survey of SHOP2 applications, see Nau, *et al.* [21]. Another advantage of Shop is that it provides easy call-out to special purpose solvers through an ability to invoke arbitrary Lisp code. For example, we used this to invoke code in a navigation library that could generate route plans, compute distances on the globe, and retrieve ground elevation information from a GIS to plan safe routes.

Our initial steps working with Shop2 involved modernizing the code base for use in larger systems. After the modernization, we built a number of large systems incorporating our version of Shop, and became increasingly aware of issues in programming and using it. This led us to add features to make it easier to program (develop new domains and programs) *correctly*, and to debug. We rearchitected the Shop2 to Shop3 for two reasons: to make it easier to incorporate/reuse individual components of Shop in external systems and to support extending and adapting Shop's planning model, by incorporating new input languages, inference methods, etc. This rearchitecting has also enabled us to incorporate an entirely new search engine into Shop3, in a way that enables us to better fit search methods to application domains.

Algorithm 2 Simplified planning search algorithm

```
1: procedure FIND PLANS($S$, $T$, $B$)                  ▷ state, tasklist, bindings
2:     if $T = \emptyset$ then
3:         return ()               ▷ No tasks: return empty action sequence.
4:     end if
5:     choose $t \in T$ with no predecessors
6:     if $t$ is primitive then
7:         $o \leftarrow$ operator for $t$
8:         if $o$ is applicable in $S$ then
9:             $S' \leftarrow$ result($o$, $S$)
10:            $T' \leftarrow T - t$
11:            $P \leftarrow$ FIND PLANS($S'$, $T'$, $B$)
12:            return cons($o$, $P$)
13:        else
14:            return $FAIL$
15:        end if
16:    else                                          ▷ $t$ is a complex task
17:        $< b, R' >=$ reduction($t$, $S$)
18:        if $b$ is $FAIL$ then
19:            return $FAIL$
20:        else
21:            $B'$ = apply($b$, $B$)
22:            if $B'$ is $FAIL$ then
23:                ▷ Merge new bindings with incoming.
24:                return $FAIL$
25:            end if          ▷ Replace $t$ with its expansion $R'$ in $T$
26:            $T' \leftarrow$ replace($t$, $R'$, $T$)
27:            return FIND PLANS($S$, $T'$, $B'$)
28:        end if
29:    end if
30: end procedure
```

Algorithm 3 Task reduction procedure

```
1: procedure REDUCTION($t$, $S$)                           ▷ task, State
2:     choose $m$ a method for name($t$)
3:     ▷ List of bindings from precondition query.
4:     $b^*$ = query(pre($m$), $S$)
5:     if $b^* = \emptyset$ then
6:         return $FAIL$
7:     else
8:         choose $b \in b^*$               ▷ bindings from preconditions
9:         $R \leftarrow$ task-net($m$)
10:        $R' \leftarrow$ apply($b$, $R$)
11:        return b, R'
12:    end if
13: end procedure
```

[1] Although variants, including depth-first iterative deepening and branch and bound search are also available.

5 SHOP3 ARCHITECTURE

Like much University research software, Shop2 was originally distributed as a single file, not at all suitable to incorporation in a larger system and also available in many hard-to-track variants – approximately one per graduate student! So for our first application using Shop2, we did a thorough modernization. The first step was namespacing: creating a SHOP package and identifying Shop2's API. Another software engineering chore was to add an ASDF system definition and decompose the source code into multiple files organized topically, and by dependency. We originally moved it to Sourceforge's subversion server for revision control. Finally, as we used Shop more and more, we accumulated a large set of regression tests for the system.

The original monolithic Shop2 system contained at least three different subsystems that were of general usefulness, and that should be usable separately. See Figure 3 for a diagram of the Shop3 system architecture. Two low-level subsystems that could be loaded independently were the `shop3/unifier` and `shop3/common` (state) systems. The unifier, as the name suggests, provides an implementation of the unification algorithm [24] over s-expressions, with supporting data structures for binding, etc. This can be used as a library independent of the rest of Shop. Similarly, `shop3/common` provides an implementation of a logical database supporting change over time. It contains state data structures (of different forms, offering different tradeoffs in cost between update and retrieval), with update and undo operations.

The `shop3/theorem-prover` provides a Prolog-like logic programming framework on top of the state data structures of `shop3/common` and the unification algorithm of `shop3/unifier`. Like conventional Prolog, the Shop theorem-prover provides Horn clause deduction, but unlike it, the theorem-prover allows the programmer to reason about a state that changes over time, forming a state *trajectory*. Note that the theorem-prover does *not* support temporal logic, but it does allow the programmer to provide Prolog-style reasoning as a state evolves through changes. Temporal logic would be an interesting extension, but it remains to be seen how well it would interact with the changeset-style representation Shop uses. The `shop3/theorem-prover-api` provides a programmer-friendly general `query` interface to the theorem-prover, an addition to the more primitive API used by the planner itself.

On top of the theorem-prover and state representation, Shop3 contains a sub-library of "planning operations" that cover the core operations of primitive task insertion and task reduction we see in Algorithms 2 and 3. This layer was separated out in order to support the development of the more general Explicit Stack Search engine described below in Section 5.2. These operations include adding an operator/action to the plan sequence, while updating the state; and replacing a complex task with the task network from a compatible method.

At the top of the pyramid is the Shop3 system itself, marrying search with planning operations, theorem-proving, and state update. The system also provides assorted utilities for visualization, etc., including a plan grapher library that can draw a plan derivation tree using Cl-dot [25] and `graphviz` [4, 6]. This was a door opened to us by the modernization of the system, and the interoperability and supply of libraries enabled by the common adoption of ASDF.

The original Shop2 system had its own idiosyncratic syntax, which evolved more than being designed. This led to parsing code that developed into a hard-to-maintain ball of special cases. Also, the requirement to use that syntax made comparisons with other planners more difficult, and generally closed off access to the large number of publicly-available planning models and problems that have come out of the IPC. Accordingly, we added support for the PDDL language, as we describe below. A substantial part of this involved making the input of domains and problems generic and dispatching on domain type (see below). The theorem-prover's behavior was also made generic, which allows Shop3 to support the different notation for logic used by PDDL and Shop2, and their different scoping disciplines.

5.1 Object model: Domains

As described earlier, a planning *domain* is a repository of action models, and in the case of Shop, also method definitions, and axioms. Domains capture everything that is common among a set of related planning problems. For this reason, when we were designing Shop3 for greater extensibility, subclassing the `DOMAIN` object class was the obvious step to take. Figure 4 gives a simplified diagram of the type lattice around the `DOMAIN` class.

Domains enable customization of domain and problem notation through generic functions for parsing, theorem-proving, and search. Figure 4 gives a sense of how this is done. At the root of the hierarchy is the theorem-prover domain. This class provides the theorem-proving behaviors through methods on `REAL-SEEK-SATISFIERS-FOR` and `LOGICAL-KEYWORDP` that dispatch on the domain and a logical keyword. Definition of such methods is aided by a `DEF-LOGICAL-KEYWORD` macro.

The standard Shop3 `DOMAIN` combines the theorem-prover domain with a mixin that stores action definitions to enable state progression. Examples of customization can be seen in the immediate subclasses of `DOMAIN`, including `TEMPORAL-DOMAIN`, which adds the ability to have operators with durations, and the `LOOPING-DOMAIN`, which adds looping constructs to the Shop3 domain language.

More interesting are the cluster of classes that support PDDL. The PDDL language has a `:requirements` construct that allows features of the language to be enabled separately. The basic language supports only simple action schemas with conjunctive preconditions and effects, and untyped variables (implicitly universally quantified). Various requirement keywords enable the addition of more complex constructs to the language. For each of these we have provided a corresponding `-MIXIN` class (see the left side of Figure 4). Note the addition of the ADL (Action Description Language) [22] class, which assembles a bundle of logical operators, quantifiers, and conditional effects.

5.2 Explicit Stack Search (ESS)

The core of AI planning is *search*; the search for a path of states from the initial state to a state that satisfies the goal. For a number of reasons, we would like to experiment with different search methods, but the original Shop2 implemented its search process using the Lisp process stack. This makes it difficult, if not impossible, to control the search. In general, while using the processor stack as the search stack provides a convenient and rapid way to implement

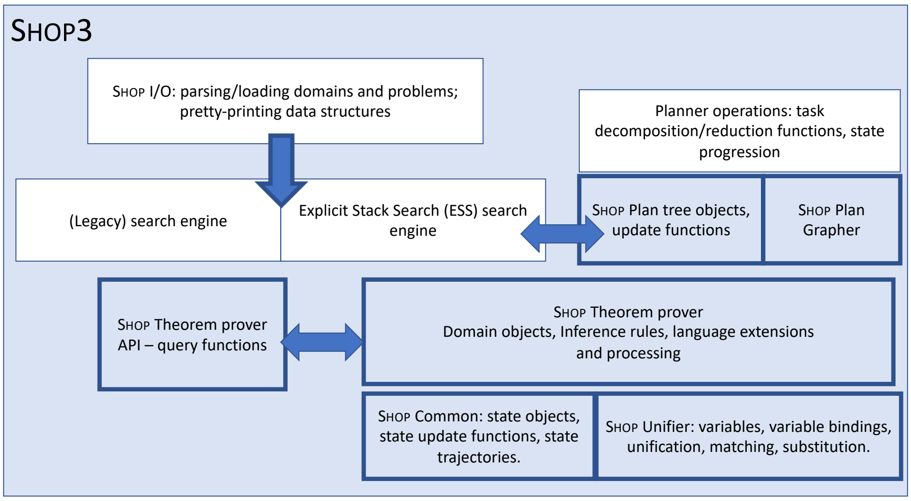

SHOP3
SHOP I/O: parsing/loading domains and problems; pretty-printing data structures
Planner operations: task decomposition/reduction functions, state progression
(Legacy) search engine
Explicit Stack Search (ESS) search engine
SHOP Plan tree objects, update functions
SHOP Plan Grapher
SHOP Theorem prover API – query functions
SHOP Theorem prover Domain objects, Inference rules, language extensions and processing
SHOP Common: state objects, state update functions, state trajectories.
SHOP Unifier: variables, variable bindings, unification, matching, substitution.

Figure 3: High level SHOP3 system architecture. Shaded blocks indicate components directly accessible by programmers. Dark outlines indicate components that can be loaded stand-alone.

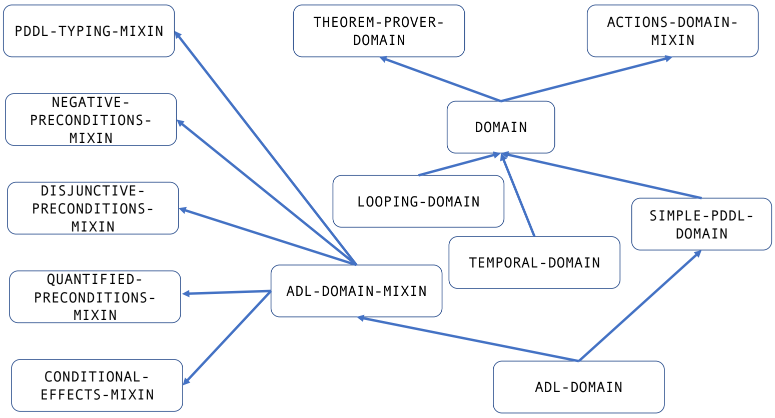


Figure 4: Object model: domains.

search algorithms, tackling complex search problems requires finer control than this implementation strategy provides.

To address this issue, we have introduced Explicit Stack Search (ESS) as an alternative to the standard SHOP2 search algorithm. The first step in doing this was to reify the search engine as a CLOS object, so that we could use generic function method dispatch to tailor individual behaviors. The next step was to introduce the "planning operations" layer (see Figure 3) to tease apart the planning operations from the recursive function invocations of the stock SHOP2 depth-first search strategy.

To implement ESS, we adopted a technique from the CIRCA planner [20], and constructed an abstract finite state machine. Each state in the virtual machine will carry out some computation, update virtual machine data structures, and then jump to a new state. This allows easy incorporation of new behaviors. The current version of SHOP3 supplies standard depth-first search, mirroring the original SHOP2 search implementation, and also *backjumping* [7], which we are using in recent work on plan repair.

The core data structures for ESS are a search state object, and the backtrack stack. The search state groups together all of the information in the state of the search as described in Algorithms 1 through 3 – the current task, the plan so far, the cost, etc. – in addition to the state of the virtual machine – the mode, unexplored alternatives to the current decision, etc. The backtrack stack is made up of two kinds of object. The simplest is a marker, which indicates a choice point, and is pushed onto the stack whenever there are unexplored alternatives. The other is one of a set of objects that are pushed to record the effects of a decision. These objects hold data: e.g., when a new choice point is reached, ESS must push the unexplored alternatives from the last choice point. For each class

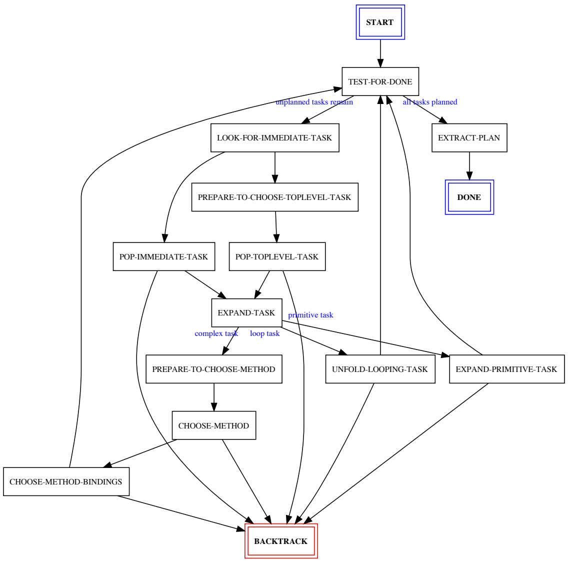


Figure 5: State machine for ESS.

of these objects, there must be an undo method provided, that can restore the state of the search machine, and of the search itself.

6 SHOP3 IMPLEMENTATION

In this section we discuss a number of miscellaneous topics concerning the implementation of Shop3, and features provided by and to the community. First, we have worked to clean up the syntax of the languages for the planner. We have also migrated away from using lists as the uniform data structure. Finally, we developed an extensive set of tests and to support it, created a library that integrates the FiveAM testing library with the ASDF build system.

The original logical language of Shop2 was idiosyncratic in syntax and in the set of capabilities provided. In Shop3 we provide standard features of Prolog, including all-solutions meta-predicates, etc. We also add more debugging features, including the ability for the programmer to explicitly raise runtime exceptions, singleton variable checks (for misspellings), and anonymous variables (to support singleton variable checks). We are gradually adding more load-time checks to domain definitions, although this is complicated by Shop3’s support for meta-programming.

Debugging and efficiency have both been improved by extensive introduction of special-purpose data structures in place of Shop2’s pervasive use of lists for all purposes. The pervasive use of lists made it quite difficult, for example, to debug the theorem-prover, which did not make a clear distinction between binding lists (lists of variable bindings, representing a single solution) and lists of binding lists (representing multiple solutions, in each element of which variables would be bound to different values). The pervasive use of lists also made Shop2 less efficient. Use of structures for inner-loop operations (e.g. variable bindings in the theorem-prover) has provided substantial speed-ups, as well as code that is easier to understand and maintain, because of fewer quiet failures.

As part of the Shop3 development effort, we developed the FiveAM-ASDF library. As its name suggests, this provides integration between the FiveAM Lisp testing library [2], and the ASDF build system [23]. It enables the programmer to designate a set of tests to be run as the `TEST-OP` of an ASDF system, and also provides conditions for test failures, etc. These conditions are necessary because the execution of ASDF operations does not provide a useful return value. This way the system can be tested either interactively, or one can encapsulate the test operation in a trivial bash script that will provide a non-zero error code in case of failures, thus making it suitable for use in a continuous integration framework. Because we found that errors in test definition could lead to silent failures, the system also provides the programmer the ability to specify the number of checks that are expected and FiveAM-ASDF will raise a (different) condition if an unexpected number of checks are run. As an aside, we are starting to decompose the regression tests into multiple sub-systems, as the library of tests has become so extensive, and testing is trivially parallelizable. The test suite takes approximately 45 minutes to run for each CL implementation.

7 CONCLUSIONS AND LESSONS LEARNED

We have described our use of CL and its language features to support extensive symbolic computations in the HTN planning systems Shop3, and Shop2. CL provided a good foundation for extensibility and refactoring of these systems, and more usability as practical tools.

One of the advantages of using CL for an AI planning system was the use of s-expressions as the universal data structure in the software. Our experience has been that this can be very convenient for initial prototyping, particularly because they provide more convenient inspection in the debugger, but they should be phased out rapidly for improving the scalability and modularity in the code. The foresight of the CL standardizers in providing `list`-type structures has been an immeasurable help in this process. Another substantial advantage was s-expressions as a convenient data structure for symbolic computing. Unless one has built symbolic computing systems in both CL and a more conventional language like Java or Python, it may be difficult to appreciate how great a help this is.

Going forward, the addition of rigorous gradual typing to CL would be very helpful. SBCL provides excellent type inference, but while it provides very valuable information about correctness, this is a byproduct of a concern for optimization, and the CL type system was not designed as a tool for program correctness.

Many of the things we have seen in our code and our predecessors’ in Shop2 may be common informal knowledge, but it is worth enumerating them so that they can enter the CL community’s *explicit* knowledge. Some examples include:

(1) *Ad hoc* development of domain-specific languages
(2) `&allow-other-keys` (and often the `&key` of CLOS) is an anti-pattern: the maintainer or library user trying to determine what arguments are supported will have to traverse class inheritance hierarchies (especially for `initialize-instance`), or method dispatch hierarchies, wasting time, creating confusion, and often leading to errors only caught at run-time. More specifically, if we have base class C, and we need to extend it to another class C', we realized that one should never do just that. Instead, factor out commonalities into a $C0$, and make C and C' both be children of $C0$. If we do not, sooner or later we found that

we would need to add a behavior to C that should not go on C', and then it would be a painful refactoring.

(3) Cascading `initialize-instance` methods also create confusion and cause issues with maintenance.

A foremost best practice that underlies many of the others is to *assume* that the code will break and ask yourself how it can be debugged, and especially how it can be debugged by others. A common mistake by the over-confident programmer is to provide a complex, but elegant structure that resists debugging because of not asking this question.

ACKNOWLEDGMENTS

We would like to thank Prof. Dana Nau and his students and post-docs at the University of Maryland for the original Shop2. John Maraist for many extensions during his time at SIFT. Rick Freedman, Dan Bryce, and other SIFT employees for the logo. The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory under Contract Number FA8750-17-C-0184. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, the Department of Defense, or the United States Government.

REFERENCES

[1] Christer Bäckström, Yue Chen, Peter Jonsson, Sebastian Ordyniak, and Stefan Szeider. 2012. The Complexity of Planning Revisited-A Parameterized Analysis.. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence.* AAAI Press, Toronto, Ontario.

[2] Edward Marco Baringer and Stelian Ionescu. [n. d.]. FiveAM. https://common-lisp.net/project/fiveam/

[3] Mark Burstein, Robert Goldman, Paul Robertson, Robert Laddaga, Robert Balzer, Neil Goldman, Christopher Geib, Ugur Kuter, David McDonald, John Maraist, Peter Keller, and David Wile. 2012. STRATUS: Strategic and Tactical Resiliency Against Threats to Ubiquitous Systems. In *Proceedings of SASO-12.*

[4] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. 2003. Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools. In *GRAPH DRAWING SOFTWARE.* Springer-Verlag, 127–148.

[5] Maria Fox and Derek Long. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20, 1 (Dec. 2003), 61–124. http://dl.acm.org/citation.cfm?id=1622452.1622454

[6] Emden R. Gansner and Stephen C. North. 2000. An Open Graph Visualization System and Its Applications to Software Engineering. *SOFTWARE - PRACTICE AND EXPERIENCE* 30, 11 (2000), 1203–1233.

[7] John Gaschnig. 1979. *Performance Measurement and Analysis of Certain Search Algorithms.* Technical Report CMU-CS-79-124. Carnegie-Mellon University.

[8] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Dan Weld, and David Wilkins. 1998. *PDDL – The Planning Domain Definition Language.* Technical Report CVC TR-98-003. Yale Center for Computational Vision and Control, New Haven, CT.

[9] Malik Ghallab, Dana Nau, and Paolo Traverso. 2004. *Automated Planning: Theory and Practice.* Morgan Kaufmann, San Francisco, CA.

[10] Robert P. Goldman, Daniel Bryce, Michael J. S. Pelican, David J. Musliner, and Kyungmin Bae. 2016. A Hybrid Architecture for Correct-by-Construction Hybrid Planning and Control. In *NASA Formal Methods*, Sanjai Rayadurgam and Oksana Tkachuk (Eds.). Vol. 9690. Springer International Publishing, Cham, 388–394. https://doi.org/10.1007/978-3-319-40648-0_29

[11] Ulrich Harm. 2018. Bayeux Tapestry TItuli. http://www.hs-augsburg.de/~harsch/Chronologia/Lspost11/Bayeux/bay_tama.html

[12] Jörg Hoffmann, Stefan Edelkamp, Sylvie Thiébaux, Roman Englert, Frederico dos S. Liporace, and Sebastian Trüg. 2006. Engineering Benchmarks for Planning: The Domains Used in the Deterministic Part of IPC-4. *Journal of Artificial Intelligence Research* 26 (2006), 453–541. http://dx.doi.org/10.1613/jair.1982

[13] U. Kuter, M. Burstein, J. Benton, D. Bryce, J. Thayer, and S. McCoy. 2015. HACKAR: Helpful Advice for Code Knowledge and Attack Resilience. In *AAAI/IAAI Proceedings.*

[14] U. Kuter, R. P. Goldman, and J. Hamell. 2018. Assumption-based Decentralized HTN Planning. In *Proceedings of the ICAPS-18 Workshop on Hierarchical Planning.*

[15] U. Kuter, B. Kettler, J. Guo, M. Hofmann, V. Champagne, K. Lachevet, J. Lautenschlager, L. Asencios, J. Hamell, and R. P. Goldman. 2019. Profiles, Proxies, and Assumptions: Decentralized, Communications-Resilient Planning, Allocation, and Scheduling. In *Proceedings of the AAAI/IAAI-19.*

[16] Derek Long and Maria Fox. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20 (2003), 1–59.

[17] Drew V. McDermott. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 21, 2 (2000), 35–55.

[18] Joseph B Mueller, Christopher A Miller, Ugur Kuter, Jeff Rye, and Josh Hamell. 2017. A Human-System Interface with Contingency Planning for Collaborative Operations of Unmanned Aerial Vehicles. In *AIAA Information Systems-AIAA Infotech@ Aerospace (2017-1296).* AIAA Press. https://doi.org/10.2514/6.2017-1296

[19] David Musliner, Robert P. Goldman, Josh Hamell, and Chris Miller. 2011. Priority-Based Playbook Tasking for Unmanned System Teams. In *Proceedings AIAA.* American Institute of Aeronautics and Astronautics.

[20] D. J. Musliner, E. H. Durfee, and K. G. Shin. 1993. CIRCA: A Cooperative Intelligent Real-Time Control Architecture. *IEEE Transactions on Systems, Man, and Cybernetics* 23, 6 (Nov. 1993), 1561–1574. https://doi.org/10.1109/21.257754

[21] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, H. Muñoz-Avila, J. W. Murdock, D. Wu, and F. Yaman. 2005. Applications of SHOP and SHOP2. *IEEE Intelligent Systems* 20, 2 (March–April 2005), 34—41.

[22] E.P.D. Pednault. 1989. ADL: Exploring the Middle Ground between Strips and the Situation Calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Ron Brachman, Hector J. Levesque, and Raymond Reiter (Eds.). Morgan Kaufmann Publishers.

[23] Francois-René Rideau and Robert P. Goldman. 2010. Evolving ASDF: More Cooperation, Less Coordination. In *International Lisp Conference.* ACM Press. https://doi.org/10.1145/1869643.1869648

[24] J. A. Robinson. 1971. Computational Logic: The Unification Computation. *Machine Intelligence* 6 (1971).

[25] Juho Snellman. [n. d.]. CL-DOT – Generate Dot Output from Arbitrary Lisp Data. http://www.foldr.org/~michaelw/projects/cl-dot/

[26] U.Kuter, R. P. Goldman, D. Bryce, J. Beal, M. DeHaven, C. Geib, A. F. Plotnick, N. Roehner, and T. Nguyen. 2018. XPLAN: Experiment Planning for Synthetic Biology. In *Proceedings of the ICAPS-18 Workshop on Hierarchical Planning.*

Session VII: Racket

Tuesday, 2.4.2019

14:30–15:30	Matthew Flatt: Rebooting Racket (Guest talk)
15:30–16:00	**Coffee Break**

Session VIII: Ecosystem

Tuesday, 2.4.2019

16:00–16:30	Alessio Stalla: Symbols as Namespaces in Common Lisp
16:30–17:00	Didier Verna: Parallelizing Quickref
17:00–17:30	Lightning talks
17:30	**Conference end**

Symbols as Namespaces in Common Lisp

Alessio Stalla
alessiostalla@gmail.com

ABSTRACT

In this paper, we propose to extend Common Lisp's fundamental symbol data type to act as a namespace for other symbols, thus forming a hierarchy of symbols containing symbols and so on recursively. We show how it is possible to retrofit the existing Common Lisp package system on top of this new capability of symbols in order to maintain compatibility to the Common Lisp standard. We present the rationale and the major design choices behind this effort along with some possible use cases and we describe an implementation of the feature on a modified version of Armed Bear Common Lisp (ABCL).

CCS CONCEPTS

• **Software and its engineering → Modules / packages**; *Data types and structures*; Semantics.

KEYWORDS

Symbols, Namespaces, Packages, Hierarchy, Common Lisp, ABCL

ACM Reference Format:
Alessio Stalla. 2019. Symbols as Namespaces in Common Lisp. In *Proceedings of the 12th European Lisp Symposium (ELS'19).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.2648195

1 INTRODUCTION

1.1 Symbols

One of the distinguishing features of Lisp, that has set the language apart since its inception in the late 1950's [5], is the symbol data type. In Lisp, symbols are names for things, including parts of a program such as variables, functions, types, operators, macros, classes, etc. and including user-defined concepts and data. Each symbol has a name, which is a string, and a *property list*, an associative data structure where the language implementation, as well as libraries and programs, can store data and meta-data related to the symbol or associated concepts. The Lisp reader ensures that, when it encounters the same symbol name twice, it will return the same, identical symbol; so, when reading the textual representation of some source code, the same name appearing as a different string in several positions in the source text will refer to the same concept or program element.

Symbols are a data type usually found in compilers and interpreters. Lisp exposes the symbol concept to the user of the system and is itself built upon it – it is, in its core, a system for the manipulation of lists of symbols. This makes programming in Lisp qualitatively different from other programming languages [3], be they object-oriented, functional, imperative, and so on. Usually we tend to concentrate on lists (or, better, on conses) and to forget about the importance of symbols.

1.2 Packages

Early Lisps had a single namespace for symbols [6]. That is, a single global hash table that the Lisp reader consults when it encounters a symbol name: if it's already present in the table, the associated symbol is used, otherwise a fresh symbol is created and stored in the table. This operation is called `INTERN`. The table itself is called an `obarray` in Maclisp [8], Emacs Lisp [2], and perhaps other Lisps.

A single namespace suffers from the possibility of symbol clashes – that is, two programs assigning incompatible meanings to the same symbol. Eventually, this problem was addressed in Common Lisp with its *package system* [9], derived from an earlier system introduced in Lisp Machine Lisp [12].

In Common Lisp, packages are objects that map symbol names (strings) to symbols. More than one package can be defined; indeed, the Common Lisp standard defines three built-in packages, `COMMON-LISP` for the language itself, `KEYWORD` for keyword symbols that are constants that evaluate to themselves, and `COMMON-LISP-USER` for user symbols. However, exactly one package is current at any one time (per thread): it is, by definition, the value of the special variable `*PACKAGE*`.

The reader always interns unqualified symbol names in the current package. To refer to a symbol in another package, the symbol name is prefixed by the package name followed by a colon – or two, in certain cases which we'll explain shortly. For example, `ALEXANDRIA:WHEN-LET`.

Packages, like symbols, have a name which is a string. Additionally, packages can have multiple secondary names called *nicknames*. For example, the `COMMON-LISP` package is nicknamed `CL`. And, just like symbols in earlier Lisps, packages are registered in a single global map keyed by their names and nicknames. It is possible to remove a package from the map, using the `DELETE-PACKAGE` operator (which removes at once all the entries referring to that package, including nicknames). However, the Common Lisp standard does not specify any meaningful way in which a package object can be used once it's been deleted, and it does not define any operator to put a package in the global map back again. In practice, in a portable Common Lisp program, a deleted package is no longer usable in any way.

The package system also acts as a simple but effective read-time module system. A package can export some of its symbols; other packages can import them individually or they can *use* another package, thus automatically importing all its exported symbols. The names of non-exported symbols need to be prefixed with two colons when referring to them from other packages.

ELS'19, April 01–02 2019, Genova, Italy

ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2648195

2 PACKAGE PROBLEMS

Packages are arguably one of the most criticized (or poorly understood) features of the Common Lisp language:

- One issue is that the package system is not a very advanced module system, or not much of a module system at all. By design, it is just a system for organizing names and avoiding or handling clashes.
- Another issue is that the package system works at read-time, thus it relies on, and suffers from, read-time side effects [4].
- Then, as we've previously said, package names and nicknames live in a global shared namespace. With the ever-increasing amount of libraries (the Quicklisp distribution contains more that 1500 libraries [1]), each defining at least one package but quite often more than one, the possibility for package name clashes is real, especially considering the existence of nicknames which are often short mnemonic names or acronyms.
- Finally, since packages cannot portably exist as usable objects outside their global namespace, solutions using temporary or "unnamed" packages are awkward and feel hacky.

In this article, we'll focus on the last two issues: a single, global namespace for package names and nicknames, and the fact that packages are tied to this namespace.

3 EXISTING EXTENSIONS OF THE PACKAGE SYSTEM

Naming issues such as potential clashes need not necessarily be solved with technique alone. Solutions based on social conventions are often employed successfully. For example, the C language does not have any provision for namespacing. Developers simply prefix the names of functions and variables to avoid clashes.

Also, the package system itself natively provides tools for dealing with naming conflicts, such as the aforementioned nicknames and the `RENAME-PACKAGE` function.

Still, a number of extensions to the package system have been proposed and implemented over the years. We'll now review a couple of those.

3.1 Hierarchical Packages in Allegro Common Lisp

Allegro Common Lisp augments packages with some hierarchical structure [10]. Taking inspiration from languages such as Java and C#, it proposes a naming convention according to which package names are a sequence of names separated by dots.

The hierarchical structure in Allegro is just a naming convention; it does not mandate correspondence to a directory and file structure, as in Java. However, the convention is understood and enforced by a few functions that also allow users to shorten package names when they have some substructure in common, in a way that resembles file system paths. For example, `(find-package :.test)` would return the package `it.alessiostalla.app.test` if the current package were `it.alessiostalla.app`.

An informal survey by the author revealed that hierarchical packages appear to be seldom used, if at all, outside the internals of Allegro Common Lisp itself. Interestingly, the same facility exists as an open-source library by Pascal Bourguignon (https://gitlab.com/com-informatimago/com-informatimago/blob/master/common-lisp/lisp/relative-package.lisp).

3.2 Package-local Nicknames in SBCL and Other Lisps

Other Common Lisp implementations, in particular at least SBCL [11] and ABCL, allow to define package-local nicknames. That is, a package, say P, can specify local nicknames for other packages. When P is the current package, and only then, those nicknames can be used to refer to the nicknamed packages. Thus, the local nicknames do not pollute the global package namespace. Therefore, users can shorten frequently-used package names without fearing collisions with other unrelated packages that happen to have the same nickname.

This apparently simple feature is nevertheless quite powerful, and indeed, from an informal survey by the author of this paper, it is actively used or at least well regarded by a number of experienced developers on the Common Lisp Professionals mailing list.

4 SYMBOLS AS NAMESPACES

Let's now analyze the more radical option of using symbols as namespaces for other symbols. How would it look like? Is it worthwhile? Is it possible to extend Common Lisp with such a feature and to deprecate packages while maintaining backwards compatibility with the Common Lisp standard and with existing Common Lisp code?

4.1 What We Want to Achieve

We want to use symbols as namespaces for other symbols, and so on recursively. In other words, we want every symbol to potentially act as a package. We want these extended symbols to be recognized by the Lisp reader and Lisp printer. We'll use the syntax `foo:bar:baz` to represent a symbol named `baz`, which is an external symbol in the symbol named `bar`, which is itself an external symbol in the symbol named `foo`. So the idea is to extend the Common Lisp syntax to allow multiple, non-consecutive package markers. According to the Common Lisp standard, the treatment of tokens with multiple package markers is undefined and implementation-dependent [7], so this extended syntax is allowed by the standard. Similarly, we'll use two consecutive colon characters to refer to internal symbols.

Symbols that are not interned in other symbols, or in other words have no parent symbol, are called *root symbols*. We want to have exactly one canonical root symbol, that we'll denote here as `#<ROOT>`, such that the syntax `:foo` represents a symbol named `foo` whose parent is the canonical root symbol. Other root symbols can be created (for example, with the standard function `MAKE-SYMBOL`), but only the canonical one gets the special syntax described here. The canonical root symbol cannot be replaced. It is printed as a an empty string, thus it cannot be read by itself.

In general, we'll need to add new operators to our Common Lisp implementation to support these new features. Rather than adding them in a new package, we have chosen to intern them in the `SYMBOL` symbol, as it will become apparent in the next section.

4.2 Backwards Compatibility

The idea is to retrofit packages as a facade over namespacing symbols, in order to maintain Common Lisp compatibility for existing programs that do not make use of, or even know about, our new symbol type. So, our extended symbols will need to retain all the package features such as exporting, USEing other namespaces, shadowing, etc. We could leverage the symbols' property lists for those features, or we could add implementation-dependent fields to the symbol type itself. We provide an operator, `(symbol:as-package symbol)`, that given a symbol will return a package object reflecting that symbol's name and contents.

Furthermore, since packages can be given nicknames, our new symbols must support the same feature if we want them to replace packages. So, it must be possible to intern the same symbol with different names (aliases) in the same or in different namespaces. This is an important difference from Common Lisp, where symbols either have exactly one home package, or they are *uninterned*. Our symbols can have multiple aliases in multiple namespaces, but they always have a name and a parent symbol (or NIL). We provide two operators:

```
(symbol:alias symbol alias &optional export)
```

to create an alias of a symbol in its parent namespace, and

```
(symbol:remove-alias symbol alias)
```

to remove an alias. To create an alias of a symbol in a different namespace, we use the standard `IMPORT` function, which of course now works on symbols as well as on packages. Note that the hierarchical nature of symbols implies that aliases are local, just like the package-local nicknames extension in SBCL and ABCL.

5 THE ROOT SYMBOL, KEYWORDS AND TOP-LEVEL PACKAGES: A PROBLEM

It is apparent that keywords and symbols in the root namespace have something in common. In part this comes from the choice of syntax: since `foo:bar` is symbol `bar` with parent `foo`, `:foo` ought to designate the symbol `foo` with parent `#<ROOT>`, just like paths in a file system. But `:foo` in Common Lisp is already the syntax for the keyword named `foo`. It's not only a matter of the choice of syntax, though. Keywords are meant to be read uniformly and unambiguously no matter what the current package happens to be. Analogously, one is supposed to be able to reach to the root namespace with the same syntax no matter what the current namespace is. Even not considering syntax, keywords and symbols in the root namespace appear to be similar beasts.

Another seemingly obvious choice is that legacy Common Lisp packages – which are inherently top-level, global names – ought to live in the root symbol, so that package `COMMON-LISP` is actually the symbol `#<ROOT>:COMMON-LISP`, which is then printed as `:COMMON-LISP`. Thus the root symbol is the global map that contains all packages, which in standard Common Lisp is an object that users of the language cannot access.

These two apparently natural choices, however, don't play well together. In fact, to preserve backwards compatibility, the Lisp reader, when searching for a package (say, `COMMON-LISP`), must either search it locally to the current namespace (we'll call this option L for Local first), or it must search it in the root first, then in the current namespace (option R for Root first).

Choice (L) implies that, for, say, `CL:LIST` to be read consistently everywhere, every package must import the `:CL` symbol. More generally, every symbol which denotes a package must be accessible (imported) in every namespace. But if the symbol, such as `:CL`, is also a keyword by design, then it is a constant and it cannot be rebound, not even locally. This is a strong limitation and a problem for backwards compatibility, especially for packages with common names like `SEQUENCE`, `SYSTEM`, `EXTENSIONS` etc. which collide with symbols in the `COMMON-LISP` package or with symbols in user code. Clearly, having packages named by keywords and requiring all namespaces to import those keywords has heavy usability implications.

Choice (R) instead implies an inconsistency. In the expressions `CL:X` and `CL`, the two character strings "CL" might be read as different symbols: the first as `#<ROOT>:CL`, the second as, for example, `CL-USER:CL`. Also, with that scheme, `:KEYWORD` would be a symbol whose namespace is itself, which is confusing, but this is probably just a minor annoyance.

6 AN IMPERFECT SOLUTION

We can then decide that the root symbol is not the keyword package after all. This, however, has other problems. In fact, there is a read inconsistency for keywords. `:foo` must be read as a keyword at least for backwards compatibility, but in the expression `:foo:bar`, `foo` is not a keyword, it is symbol `FOO` in symbol `#<ROOT>`.

Also, there is still the inconsistency of, e.g., `SEQUENCE` in the expressions `'SEQUENCE:COUNT` and `'SEQUENCE` being two different symbols if `SEQUENCE` is a top-level package, unless the symbol `SEQUENCE` is imported in the current package. This is for backwards compatibility, because if a user evaluates `(defpackage foo)`, Common Lisp mandates that `foo::x` refers to the top-level package `FOO` no matter what the current package is.

However, there isn't the additional limitation of keywords being constants, so `SEQUENCE`-the-package and `SEQUENCE`-the-CL-symbol can be arranged to be the same symbol without drawbacks, by importing `:sequence` in CL (or by importing `cl:sequence` into `#<ROOT>`). For system packages, the implementation can probably arrange things like that automatically, so for users it's transparent. For user packages, this cannot be done by the implementation, users have to write the boilerplate manually if they want to avoid the inconsistency.

Ideally, if no backwards compatibility were required, we could mandate that the names of top-level packages be always prefixed by a colon – as in `:cl:count` and `:sequence:count` – unless locally imported. However, in an existing Common Lisp system, this is not possible. Legacy compatibility could be turned on and off with flags, but this still seems a bit of a mess. Things don't click, they're too complex. The beauty of the original idea seems lost.

7 THE REAL SOLUTION

Things flow much better if we take a different route. Namely, that top-level packages (actually, their names) are not top-level symbols. Instead, let's make them live in another, non-root symbol, say `:TOP-LEVEL-PACKAGES`. The root symbol, then, continues to be the home namespace of keyword symbols, that is, the Common Lisp keyword package, and the only "special" symbol and package in the system.

So, when it encounters the expression `foo:bar`, the reader looks for `foo` in the current namespace first; if it is not present or it is not a namespace, i.e. it doesn't contain other symbols (a distinction that is necessary to avoid excessively shadowing top-level packages), then it continues its search in the symbol `:TOP-LEVEL-PACKAGES`. An inconsistency can still happen – the expressions `foo:bar` and `foo` referring to different `foo` symbols – but only if a local symbol named `foo` exists and it is not used as a namespace. In that case, it is reasonable that the same sequence of characters "foo" refers to different things according to whether it's denoting a namespace or a symbol name – after all, that's how today's Common Lisp works.

As a minor annoyance, to spell, say, the `CL:LOOP` symbol in its absolute form, one must write `:top-level-packages:cl:loop`; that is, the abstraction leaks a bit.

8 IMPACT ON COMMON LISP ASSUMPTIONS

The change we are proposing cuts deep in the fundamentals of Common Lisp and arguably of Lisp as it was first conceived. What are the consequences?

One area that should definitely be explored as further work is read-print inconsistencies, as it is apparent from the previous sections. We haven't studied the issue enough to report something meaningful here.

Another problematic impact is the interplay with the function `DELETE-PACKAGE`.

8.1 DELETE-PACKAGE

In Common Lisp, `DELETE-PACKAGE` removes a package with all its nicknames from the global namespace and renders the package object unusable in portable code. In our extended Common Lisp, `DELETE-PACKAGE` should do the same thing to any symbol.

However, with symbols being potentially imported, exported, aliased, used as namespaces in several places, to delete a symbol atomically from the system requires a lock on the whole symbol system. And, even ignoring concurrency issues, there are new possibilities for failure that are currently absent in Common Lisp. Removing an alias can uncover a conflict between two used packages, for example. `DELETE-PACKAGE` is necessary because packages do not exist outside their global map; symbols do live just as well without a parent, and they can be uninterned. Once a symbol is no longer referenced by any live object it can be garbage collected. So, if our proposal is to be adopted, `DELETE-PACKAGE` should be restricted to work only on symbols and aliases in `:TOP-LEVEL-PACKAGES`, or it should be deprecated altogether in favor of `REMOVE-ALIAS` and `UNINTERN`, or both.

9 APPLICATIONS

So far hierarchical symbols might seem just a cool feature, a bizarre experiment or a hack for the sake of hacking. In our opinion, they make packages "better citizen" in a world where everything is a first-class object and can be created, manipulated and discarded at will. They also arguably (if we don't consider the backwards-compatibility complexities) provide a better, more consistent design of symbols as "things that give names to things", all the way down. However, they have practical applications, too. Here we propose one and hint at a few others.

9.1 A File System Facility

Common Lisp's pathnames are another frequently debated feature that is known to have made most users scratch their heads in confusion. Here we propose a simple library that provides an easier API for basic pathname usage, leveraging hierarchical symbols. This example will show how hierarchical symbols are a versatile feature that allows to represent all sort of hierarchical names in the language and will showcase some of the functions supporting our new symbols.

The key idea of this small library is to represent pathnames as symbols. One can mount a physical pathname, with its implementation-dependent syntax, to a given symbol. Then one can construct related pathnames by interning symbols in it and so on recursively, without setting up complex translations, manipulating strings or using the awkward Lisp pathnames API. When done with it, one can also unmount a symbol, i.e. remove all filesystem-related information from it.

So by evaluating `(mount 'foo (user-homedir-pathname))` one can represent paths such as `'foo::Downloads::virus.exe` (having readtable-case set to `:PRESERVE` helps as file systems can be case sensitive). The pathname of a given symbol can be obtained with the `pathname` operation:

```
(pathname 'foo::Downloads::virus.exe)
=> #P"/Users/alessio/Downloads/virus.exe"
```

Symbols-as-pathnames can be tested for existence, opened for reading or writing, and operated upon in all the ways supported by the native Common Lisp pathname facility.

9.2 FFI and Interoperation

There are other areas where having composite, hierarchical names can be beneficial. One is, of course, interoperation with languages that themselves have, or simulate, such names. For example, a Java FFI could allow the following:

```
(jffi:import 'java:lang:String) ;optionally :as 'java-string
(jffi:new 'String "a string") ;New object creation
(String::valueOf 42) ;Static method call
(String::toCharArray **) ;Instance method call
```

9.3 Addressing Other Kinds of Paths

Generally, every time we might want to map paths or hierarchies to symbolic data, hierarchical symbols can offer an advantage. For example:

- JSON or XML paths (XPath)
- mapping objects to database systems (e.g., schema:table:column)
- representing URL's and network path
- invoking remote functions, services, procedures

10 IMPLEMENTATION

An implementation of the above concepts and a few support functions has been realized on Armed Bear Common Lisp (ABCL). ABCL is a Common Lisp running on the JVM, written in a combination of Java and Lisp and with a significant amount of code inherited from CMUCL/SBCL.

The result is a working ABCL that has the symbol data type described earlier and fails the same ANSI tests it failed before the changes. Most modifications involved only 4 files: Symbol.java (the symbol type), Package.java (the package type), Primitives.java (primitive functions) and Stream.java (where most of the reader is defined).

As an additional consequence, ABCL's serialization of symbols in FASL files, which was brittle, broke irreparably and was rewritten to be more solid (basically printing symbols with *print-readably* bound to T, which causes them to be printed in their absolute form starting from the root, e.g. :TOP-LEVEL-PACKAGES::COMMON-LISP::T). However, this causes a certain increase in FASL size and a deterioration of load times, particularly at startup or when loading big systems, which are already a pain point in ABCL.

The implementation can be found at https://github.com/alessiostalla/abcl on the branch hierarchical-symbols.

11 FURTHER WORK

The work here is just a foundation. The implications of hierarchical symbols in Common Lisp should be analyzed further. In particular, that there are no read-print inconsistencies in corner cases.

A low-hanging fruit is to enhance our work by providing a few missing usability features. For example, since symbols can be aliased, have an :import-from :as option in defpackage/define-namespace.

A particular area of interest is the porting of the feature to other Lisp implementations. Implementing it on ABCL has been relatively easy, but it might be just a fortunate case. Also, investigating whether it is possible to implement this proposal in pure Common Lisp, with no modifications to the implementation, is a worthwhile goal.

ACKNOWLEDGMENTS

Thanks to the ABCL developers and the users of the Common Lisp Professionals mailing list for their insight and experience.

REFERENCES

[1] Zach Beane. Quicklisp beta. URL https://www.quicklisp.org/beta/.
[2] Inc. Free Software Foundation. Gnu emacs lisp reference manual. URL https://www.gnu.org/software/emacs/manual/html_node/elisp/Creating-Symbols.html#Creating-Symbols.
[3] Richard P. Gabriel. The structure of a programming language revolution. URL https://www.dreamsongs.com/Files/Incommensurability.pdf.
[4] Ron Garrett. Lexicons: First-class global lexical environments for common lisp. URL http://www.flownet.com/ron/lisp/lexicons.pdf.
[5] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL http://doi.acm.org/10.1145/367177.367199.
[6] John McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962. ISBN 0262130114.
[7] Kathy; et al. Pitman, Kent; Chapman. The common lisp hyperspec - section 2.3.5 valid patterns for tokens, . URL http://www.lispworks.com/documentation/HyperSpec/Body/02_ce.htm.
[8] Kent Pitman. The revised maclisp manual (the pitmanual), . URL http://www.maclisp.info/pitmanual/symbol.html#10.9.1.
[9] Guy L. Steele, Jr. *Common LISP: The Language (2Nd Ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.
[10] Unknown. The allegro common lisp documentation - packages, . URL http://franz.com/support/documentation/current/doc/packages.htm.
[11] Unknown. Sbcl 1.4 user manual, . URL http://www.sbcl.org/manual/#Package_002dLocal-Nicknames.
[12] Daniel Weinreb. *Lisp Machine Manual*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1981. ISBN B0006Y4UVA.

Parallelizing Quickref

Didier Verna
EPITA
Research and Development Laboratory
Le Kremlin-Bicêtre, France
didier@lrde.epita.fr

ABSTRACT

Quickref is a global documentation project for Common Lisp software. It builds a website containing reference manuals for Quicklisp libraries. Each library is first compiled, loaded, and introspected. From the collected information, a Texinfo file is generated, which is then processed into HTML. Because of the large number of libraries in Quicklisp, doing this sequentially may require several hours of processing. We report on our experiments parallelizing Quickref. Experimental data on the morphology of Quicklisp libraries has been collected. Based on this data, we are able to propose a number of parallelization schemes that reduce the total processing time by a factor of 3.8 to 4.5, depending on the exact situation.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**; • **Software and its engineering → Software performance**; Software libraries and repositories; • **Applied computing** → Hypertext languages;

KEYWORDS

Parallelization, Multi-Threading, Software Performance, Software Documentation, Typesetting

ACM Reference Format:
Didier Verna. 2019. Parallelizing Quickref. In *Proceedings of the 12th European Lisp Symposium (ELS'19)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.2632534

1 INTRODUCTION

Quickref is a global documentation project for Common Lisp [9] software. It builds a website containing reference manuals for libraries available in Quicklisp[1]. Quickref is freely available[2], so anyone can create a local version of the documentation website for personal use, but we also maintain a public website documenting the whole Quicklisp world[3].

Quickref itself is actually not much more than a layer of integration "glue" cadencing the inter-operation of four external software components (see Section 2). Until recently, it essentially consisted in a big loop, iterating over every Quicklisp library, and sequentially executing all the steps required in producing the corresponding reference manual, from downloading the library to actually producing an HTML file. Because Quicklisp is quite large (it currently provides more than 1700 libraries), this process could take between 1h30 and 7 hours on our test machine, depending on the exact conditions. Even if 7 hours, the worst case scenario, still fits nicely into one night of batch processing, it *is* worth trying to improve the performance of the system, notably by means of parallelism. The purpose of this article is to report on this work.

Section 2 describes the tool-chain involved in the generation of the reference manuals and some important characteristics of the involved software components. Section 3 presents the experimental conditions and the various configurations used to perform timing measurements. Section 4 provides and analyzes some preliminary global measurements, giving us a general idea of what to expect. Section 5 proposes different parallel solutions, each one coming with its *pros* and *cons*. Finally, after the conclusion, an extensive discussion is proposed in Section 7.

2 QUICKREF TOOL-CHAIN

Figure 1 depicts the typical reference manual production pipeline used by Quickref, for a library named `foo`.

(1) Quicklisp is first used to make sure the library is installed upfront. This is done by calling `ql-dist:ensure-installed`, and results in the presence of a directory for that library (a *release* in Quicklisp terms) in the Quicklisp directory tree. Currently, Quickref only considers one system per library, called the *primary system*. How exactly this system is computed is unimportant for this paper.
(2) Declt[4] is then run on the primary system to generate the documentation. Declt is another library of ours, written 5 years before Quickref, but with that kind of application in mind right from the start. In particular, it is for that reason that the documentation generated by Declt is in an intermediate format called Texinfo.
(3) The Texinfo file is finally processed into HTML. Texinfo[5] is the GNU official documentation format. There are two main reasons why this format was chosen when Declt was originally written. First, it is particularly well suited to technical documentation. More importantly, it is designed as an abstract, intermediate format from which human-readable documentation can in turn be generated in many different forms (PDF and HTML notably).

[1] https://www.quicklisp.org
[2] https://gitlab.common-lisp.net/quickref
[3] https://quickref.common-lisp.net
[4] https://www.lrde.epita.fr/~didier/software/lisp/misc.php#declt
[5] https://www.gnu.org/software/texinfo/

ELS'19, April 01–02 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2632534

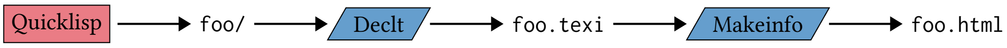


Figure 1: Reference Manual Generation
🟥 Main thread, ▱ External Process

Quickref essentially runs this pipeline on every available library (it currently has the ability to limit itself to what is already installed, or process the whole Quicklisp world). Some important remarks need to be made about this process.

First of all, Declt works by introspection: it uses ASDF[6]'s `load-system` function to load the system being processed, which may involve compiling and loading some dependencies as well as the system itself. It then introspects the system to collect documentation items, notably by way of the `sb-introspect` facility from SBCL[7]. Given the size of Quicklisp, it would be unreasonable to load almost two thousand libraries in a single Lisp image. For that reason, Quickref doesn't actually run Declt directly, but instead uses `uiop:run-program` to fork an SBCL script to do it.

Similarly, `makeinfo` (`texi2any` in fact), the program used to convert the Texinfo files to HTML, is an external program written in Perl (with some parts in C), not a Lisp library. Thus, here again, `uiop:run-program` is used to fork a `makeinfo` process out of the Quickref Lisp instance.

In light of these remarks, the reader must keep in mind the following points.

- Declt and Texinfo are treated as monolithic black boxes (in fact, ASDF as well), that is, we don't attempt to alter their operation. Any attempt at parallelization will hence boil down to scheduling the `declt` and `makeinfo` processes in a specific way. Thus, when we speak of "threads" in the remainder of this paper, we actually mean small Lisp functions that essentially fork external processes and wait for them, in a loop.
- Because running Declt may require the compilation of Lisp components, possibly including dependencies, and because different libraries may share the same dependencies, there is an inherent concurrency problem in writing the compilation files. Care must hence be taken to protect against those potentially concurrent accesses when needed.

3 EXPERIMENTAL CONDITIONS

3.1 Environment

All benchmarks reported in this paper were collected on a Debian Gnu/Linux[8] system, version 9.6 "Stretch". Quicklisp currently requires Debian 9 for testing, and advertises the list of required foreign dependencies. This environment hence guarantees that as many libraries as possible could be handled. All foreign dependencies were pre-installed, most of them already packaged by Debian. We used SBCL 1.4.0, cloned from its Git repository and manually compiled with `--fancy` (which, among other things, activates multi-threading).

The computer used was a Dell Precision T1600, equipped with 16 GB of RAM, a 120 GB mechanical hard drive and an Intel Xeon E3-1245 processor. This processor has 4 hyper-threaded cores[7], so that 8 threads are actually available. Note that as of version 2.4, the Linux kernel is aware of hyper-threading. Debian 9 includes version 4.9. Although the various timings reported in this paper were collected from single runs (as opposed to averaging several ones), the machine was freshly rebooted in single-user mode, in order to avoid non-deterministic operating system or hardware side-effects as much as possible.

For the Quickref tool-chain, the following software components were used: Declt 2.4.1, Makeinfo (`texi2any`) 6.5, and an up-to-date Quicklisp version 2019-01-07. This version of Quicklisp contains 1719 libraries. It is worth noting that Quickref currently fails on less than 2% of those libraries, for various reasons: dependency problems (foreign or not), compilation problems, Declt problems *etc.* These issues are out of the scope of this paper, so we simply filtered out the problematic libraries in our reports.

3.2 Configuration

While Quickref is primarily meant to build a complete documentation website for Quicklisp, a number of options are available, which need to be taken into account in our experiments.

3.2.1 Libraries and updating policy. By default, Quickref attempts to globally update Quicklisp before processing, which is the right thing to do for the public website. Individual users, however, also have the possibility to create a local website for their personal working environment only. To this end, Quickref makes it possible to only consider the libraries already installed on the local machine (instead of the whole Quicklisp world), and also to avoid updating those if that is unwanted. As a consequence, and depending on the exact situation, documenting a library with Quickref may or may not lead to downloading some code, and may or may not trigger some Lisp compilation (dependencies included) before actually loading and introspecting it.

3.2.2 Cache policy. On several occasions, we observed problems related to the compilation of common dependencies. One typical problem is when two libraries depend on a third one, and that dependency needs to be compiled in two different ways. A *global* compilation cache, as provided by ASDF by default is bound to fail. Another problem (which, at least in our opinion, should be regarded as a bug) is when the compilation or loading of a library leads to global side-effects on the top-level environment. The latest example we saw is that of `common-lisp-stat`, globally changing the reader's default float format from `single-float` to `double-float`[10]. This kind of behavior is bound to cause problems, especially when almost two thousand libraries are involved. Because of that, Quickref now has an option for making ASDF use a different, *local*, compilation cache for every documented library.

[6] https://common-lisp.net/project/asdf/
[7] http://www.sbcl.org/
[8] http://www.debian.org

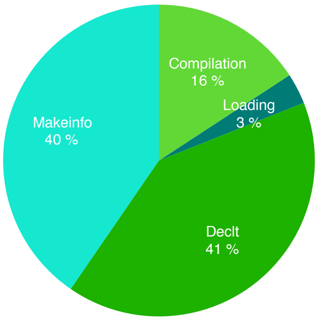


Figure 2: Time Distribution w/ compilation

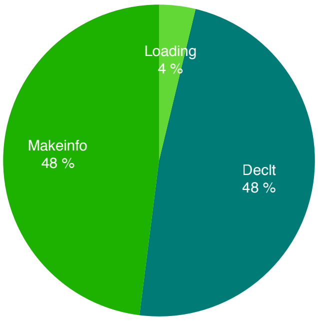


Figure 3: Time Distribution w/o compilation

3.2.3 Scenarios. In order to take into account all those different options, we have experimented on 3 different situations.

(1) All the libraries are already compiled, so Declt just needs to load them. This is the best-case scenario. Also, note that whether the compilation cache is global or local doesn't matter here.
(2) The libraries are not compiled, but a global compilation cache is used, so that no redundant processing occurs. This should be regarded as the intermediate, most frequent scenario.
(3) The libraries are not compiled, and local compilation caches are used. This is the absolute worst-case scenario.

Note that regardless of the scenario, we always process the entirety of Quicklisp (modulo the failures), and the 1719 libraries in question are already downloaded. Network connectivity is considered too fluctuating to be included in benchmarks, and besides, including it would hinder the idea of experimenting in single-user mode, in order to be as deterministic as possible. Under those conditions, we measured that in the original, sequential version of Quickref, scenario 1 takes 1h 27m, scenario 2 takes 1h 51m, and scenario 3 takes 7h 01m.

4 PRELIMINARY ANALYSIS

In order to get a general idea on the behavior of the different software components involved, we performed a set of preliminary measurements, which we partially report and analyze in this section. We separately collected ASDF load/compile times, and Declt and Makeinfo processing times for every Quicklisp library. As of this writing, the code used to collect that experimental data is available in the `benchmark` branch of the Quickref repository. The data itself is also publicly available[9].

4.1 Time Distribution

Figures 2 and 3 depict the time distribution for scenarios 2 and 1 respectively, that is, with a global compilation cache, with or without compilation. The actual values were obtained by summing the ones collected individually for each library, but they confirm the global timings reported at the end of section 3, which were obtained in another run of the scenarios. Namely, compilation takes 16m 49s, loading requires 03m 22s, Declt needs 43m 19s, and Makeinfo runs for 43m 06s. Note that the only measured redundancy here is in ASDF load times. Indeed, compilation, Declt, and Makeinfo processing occur only once per library. However, the load time measurements include not only the libraries themselves, but also their dependencies, such that the actual measure for one library corresponds to the number of times it appears as a dependency, plus one.

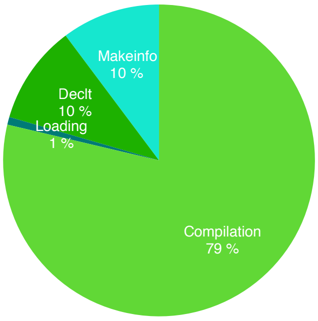


Figure 4: Time Distribution w/ separate compilation

The important information we get is that Declt and Makeinfo require practically the same amount of time to run in total. By summing ASDF time and Declt time, we see that in scenario 1 (no compilation required), Texinfo generation takes 52% of the time, versus 48% for HTML generation. In scenario 2, Texinfo generation takes 60% of the time, versus 40% for HTML generation. We did not collect numbers for scenario 3 (separate compilation directories for every library), but we can reconstruct them quite easily. Indeed, the Makeinfo, Declt, and ASDF load times are the same. The remainder of the 7h 01m, which amounts to 5h 32m, is thus devoted to (redundant) compilation. In this scenario, shown in Figure 4, Texinfo generation takes 90% of the time while HTML generation involves only the other 10%.

4.2 Time Shapes

Figures 5 and 6 provide two different views on the Declt processing real times. The first one displays the timings on a logarithmic scale, library per library (the libraries appear by lexicographic order on the X axis). The second one provides a histogram of the same data, with logarithmic scale on both axis. The actual numbers unimportant. What is important, on the other hand, is the general shape and characteristics of the data distribution.

[9] https://github.com/didierverna/quickref-benchmarks

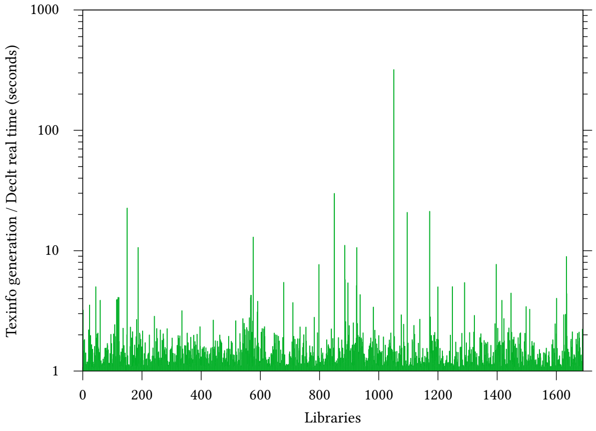


Figure 5: Declt real time per library

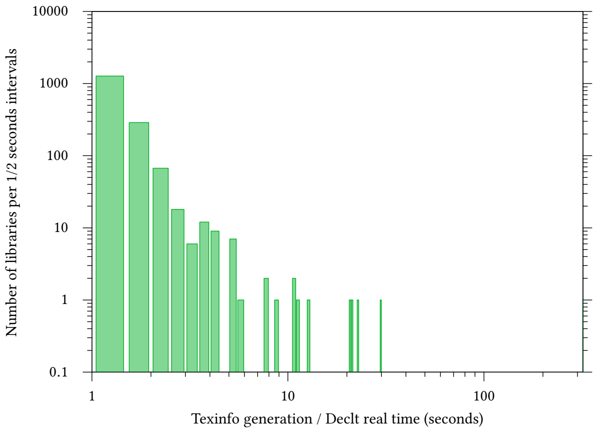


Figure 6: Declt real time histogram

The first remark is the very wide range of processing times. They spread from approximately 1s to 5m 19s. The second remark is that most of the processing times are short compared to the maximum value. 75% of the libraries are processed in less than 1.5s, 92% in less than 2s, and 96% under 2.5s. Only 20 libraries require more than 5s for processing. Unfortunately, and this is the third remark, we cannot really take advantage of this knowledge in our parallel design, because there isn't any actual probability law underlying this data distribution. The average time is 1.67s, the median is of 1.23s, but the standard deviation is 7.99, which is meaningless, given the fact that all our timings are positive. In fact, the reason for this is that we cannot discard the "aberrant" values as experimental accidents, because they aren't: they *are* reproducible. For example, the library taking more than 5 minutes of Declt processing is `lisp-interface-library`. Further investigation shows that no matter how many times we repeat the Declt run, the timing *will* remain within that order of magnitude. Thus, when a thread is busy running Declt on that library, it *will* stay busy for around 5 minutes, a time during which 180 average libraries could be processed. That is 10% of Quicklisp!

We have conducted the same analysis for ASDF compile & load times, ASDF load times only (pre-compiled), and Makeinfo processing times. We do not report the results here. Suffice to say that in every case, we note the same kind of morphology: very wide range of values, high concentration of smaller values with a small, yet undiscardable number of "aberrations".

5 PARALLEL SOLUTIONS

As of this writing, the parallel solutions presented below are all implemented in a Quickref subsystem, automatically loaded when SBCL has multi-threading support compiled in. They may be dynamically selected and parametrized (*e.g.* number of threads for each task) through a set of keywords to the main Quickref entry point (the `build` function).

5.1 Solution 1

The first proposed solution is presented in Figure 7. This solution uses only two threads, and takes advantage of the natural sequencing of operations to establish a shared buffer of Texinfo files between Declt and Makeinfo. The main thread builds the Texinfo files sequentially (in any order). The second thread waits for them, grabs them (possibly by batches, emptying the shared buffer in one shot), and converts them to HTML.

5.1.1 Advantages. This solution is very simple to implement. There is only one shared resource: the buffer of Texinfo files. Only two threads are required, so it can work on older CPU's (*e.g.* dual-core without hyper-threading), or be less demanding on an otherwise busy or shared computer. Because the libraries are processed sequentially by Declt, no concurrent compilation occurs, so this solution may be used in either of our three scenarios.

5.1.2 Drawbacks. This solution's strengths flow from the same well as its weaknesses. Because only two threads are used, it will not take full advantage of the available resources. Besides, we know from Section 4.1 that depending on the scenario, Texinfo processing takes 48%, 40%, or only 10% of the time. This means that the HTML thread will in general be waiting more than working.

5.1.3 Experimentation. Experimentation with this algorithm confirms our analysis. Scenario one (no compilation) now takes 48m 30s instead of 1h 27m (almost twice as fast). Scenario 2 (global cache policy) takes 1h 05m instead of 1h 51m (we save roughly 40% of the time). Scenario 3 (local cache) takes 6h 22m instead of 7h 01m (we save around 10% of the time).

Note that because the Texinfo files are completely independent of each other and have no dependencies, it is straightforward to add more threads for Texinfo processing (the exact same function may be spawned multiple times). This, however, would be useless, as Texinfo processing is *not* where most of the time is spent. Again, experimentation also confirms this. Only in the next solutions will it become profitable to parallelize HTML generation.

5.2 Solution 2

Solution 2, presented in Figure 8 is a logical extension to solution 1. This time, the main process spawns several threads building Texinfo

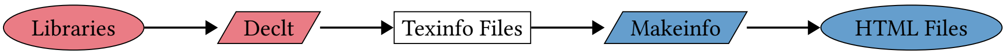


Figure 7: Solution 1
🟥 Main thread, 🟦 HTML thread

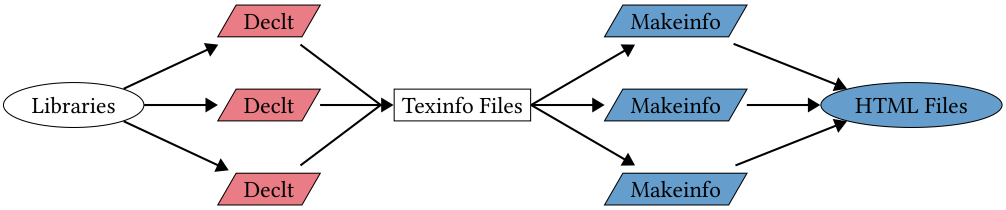


Figure 8: Solution 2
🟥 Declt threads, 🟦 HTML threads

files in parallel, and several others generating the HTML ones. As before, a shared buffer of Texinfo files is used, but compared to solution 1, there are some notable differences or complications.

- Because multiple Declt threads exist, the original pool of libraries now becomes a shared resource. The Declt threads must hence synchronize on it as well as on the shared buffer of Texinfo files.
- Contrary to solution 1, the Makeinfo threads must *not* empty the buffer at once, grabbing a whole batch of Texinfo files to process. Indeed, we have learnt from Section 4.2 that the time distribution is not homogeneous, and that some libraries take an extremely long time to process, compared to the average. Thus, if we were to grab batches of Texinfo files, we would risk an *accumulation effect*, whereby multiple "short" Texinfo files would be blocked behind a "long" one, essentially re-sequentializing HTML generation.
- For the exact same reason, it would seem ill-advised to simply split the initial pool of libraries into as many Declt threads as there are, and let them process their own batch sequentially. Instead, each thread will just process one library at the time.

5.2.1 Advantages. This solution will let us fine-tune the number of threads devoted to each task, depending on the machine at hand, or the scenario involved.

5.2.2 Drawbacks. Because the libraries are processed in parallel by Declt, in no particular order, concurrent compilation of common dependencies may occur. This solution can thus be safely used in scenarios 1 and 3 only.

5.2.3 Experimentation. Fortified by the time distribution reported in Section 4.1, we were able to fine-tune the number of threads in this solution to match our expectations. For scenario 1 (no compilation), the best results are achieved with the same number of threads for Declt and Makeinfo, specifically 4 and 4, corresponding to the hyper-threaded quad-core hardware configuration used in the experiments. It now takes 21m 47s to complete scenario one, which corresponds roughly to 25% of the original 1h 27m. For scenario 3 (local cache), the best results are achieved with 8 Declt threads and 2 Makeinfo threads. It now takes 1h 51m to complete scenario 3, which corresponds roughly to 26% of the original 7h 01m.

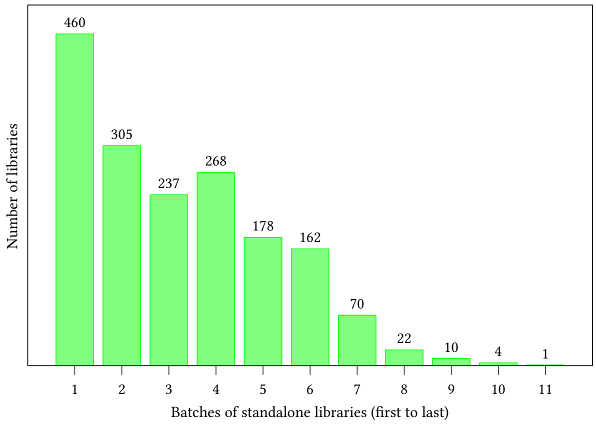


Figure 9: Library batches

5.3 Solution 3

Solution 3 is a refinement of solution 2 aiming at making it work with scenario 2 (global compilation cache). Remember that the complication comes from two libraries sharing common dependencies. Any attempt at loading them in parallel could result in the simultaneous compilation of the same dependency, followed by concurrent writing of the same *fasl* file. In order to prevent this, we must ensure that libraries processed in parallel by Declt do not have any dependencies in common, or only already compiled ones. We call those *standalone* libraries.

This problem is of course closely related to that of topological sorting[6], with the exception that we don't need full serialization. On the contrary, we want to retrieve batches of standalone libraries for parallel processing. The proposed solution is quite simple. First, we build a dependency graph of the libraries. The leaves in this graph do not have any dependencies, so they can be processed in parallel. We collect them; they constitute our first batch. We remove them from the graph, which leads to a new set of leaves, constituting the second batch. We repeat this process until the graph is exhausted. For the curious, Figure 9 shows those batches

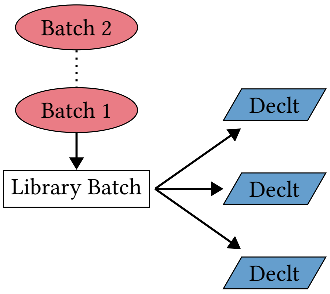


Figure 10: Solution 3, stage 1
🟥 Main thread, ▱ Declt threads

in the current Quicklisp distribution. We got 11 batches of 460 to only 1 libraries, from first to last.

In order to adapt solution 2 to this new scheme, a new shared buffer is created (see Figure 10). The Declt threads pick libraries from it instead of from the original libraries pool. The main thread sends successive batches of standalone libraries to this buffer, and waits for them to have been exhausted before sending the next batch in. The rest of solution 2 is unchanged (in particular, the Html generation code can be re-used without modification).

5.3.1 Advantages. At the expense of a slightly more complicated synchronization logic, this solution may be used in any of our 3 scenarios. In the current status of Quicklisp, the dependency graph is relatively small (less than two thousand nodes), which means that the additional computation time required to handle it is negligible compared to the 21m 47s of our current most optimistic situation.

5.3.2 Drawbacks. Before sending the next batch in, the main thread must wait for *all* libraries in the current batch to have been entirely processed by a Declt thread; not just have been picked up by one of them. At a first glance, this may not appear as a serious issue because we only have 11 batches and a few threads handling them. However, remember again from Section 4.2 that some libraries *will* take a very long time to process. If, for example, such a "long" library is part of a small batch, the batch will be quickly emptied, and all Declt threads will essentially become dormant until the "long" library is treated. This is yet another form of *accumulation effect* that can potentially hinder the parallelization.

5.3.3 Experimentation. Because the time required to maintain the dependency graph is negligible, this solution is not expected to make much difference in scenarios 1 (no compilation) and 3 (local cache), as it would boil down to handling the libraries in a different order. For scenario 2, the best result was obtained with an equal number of threads for Declt and Makeinfo, namely, 4 of each (again, corresponding to the hyper-threaded quad-core hardware configuration used in the experiments). There, the overall computation time fell down to 29m 21s, that is, 26% of the original sequential time. Given the time distribution in Figure 2, we also tried matching that proportion, for example with 5 Declt threads and 3 Makeinfo ones. We only got similar (inconclusive) result only differing by less than 5%.

6 CONCLUSION

As mentioned in the introduction, the absolute worst case scenario for Quickref, which is to build the complete Quicklisp documentation from scratch, takes around 7 hours on our test machine. Even if such a duration may appear reasonable for batch processing, we still believe that parallelization is not a vain endeavor. First of all, the ability to use Quickref interactively (creating for example one's own local documentation website) makes it worth improving its efficiency as much as possible. Secondly, Quicklisp itself is an ever-growing repository (monthly updates usually add at least a dozen new libraries to the pool), and so is the time to generate the documentation for it.

In this paper, we have devised a set of parallel algorithms, and experimented with them in different scenarios corresponding to the typical use-cases of Quickref. On our test machine, we were able to reduce the required processing time roughly by a factor of 4 compared to the naive sequential version, which is already quite satisfactory. The absolute worst-case scenario fell under 2 hours, and the most frequent one under half an hour. For all that, and in spite of the fact that gracefully handling concurrency is always a tricky business, our parallel solutions remain quite simple. The implementation of solution 3, for example, requires only 3 shared resources (2 buffers and a counter), 2 mutexes and 3 condition variables. It was implemented directly with Sbcl's multi-threading layer, without resorting to higher level libraries.

This work also lead us to perform various preliminary measurements and analysis on Common Lisp libraries (compilation and load time, Declt and Makeinfo run time, dependency graphs, *etc.*). As mentioned before, the collected experimental data and their interpretation is publicly available. We think this data could be useful for other projects, and we already know for a fact that the current Texinfo maintainers are interested. Only a small part of those results have been presented in this paper. We are confident the rest will be extremely useful for future refinements. Indeed, there are still many things that can be done to improve the situation even more.

7 DISCUSSION & PERSPECTIVES

7.1 Alternative Solution

Yet another, alternative, parallel solution exists, depicted in its entirety in Figure 11. This solution consists in processing the libraries in parallel, yet, without breaking the Declt / makeinfo chain. Multiple threads (8 would probably be an appropriate number on our test machine) pick libraries to process, and sequentially run Declt followed by Makeinfo on them. As solution 2 (Section 5.2), this algorithm can be made to work on scenarios 1 and 3 only, or, as solution 3 (Section 5.3) can be combined with library batches in order to also work on scenario 2. This is what Figure 11 depicts. In the future, and mostly out of curiosity, we may experiment with this solution.

Note however that we don't expect it to make much difference compared to solution 3. In solution 3, we have indeed fewer threads picking libraries up for Declt processing, but on the other hand, these threads also return more quickly to the library pool / batch, since they are not in charge of Makeinfo. In fact, our gut feeling is

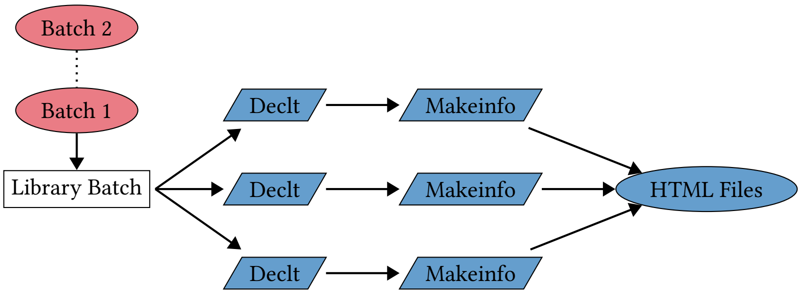


Figure 11: Alternative Solution
🟥 Main thread, ▱ Declt & Makeinfo threads

that solution 3 may remain slightly better, as it is probably more gentle on the overall waiting times.

7.2 Dependency Management Issues

Dependency management, required by solution 3, is a relatively fragile mechanism. Currently, we base our knowledge of dependencies on static information provided by Quicklisp directly, more specifically, the `required-systems` slot from the `ql-dist:system` class. This information is based on ASDF's `depends-on`, `defsystem-depends-on`, and also comes from observing the state of the environment before and after loading the library.

The reliability of that information is somehow relative, however. Any inaccuracy in that information can potentially lead to a corrupted dependency graph, in turn risking unprotected concurrent compilation. Here is, for example, one such scenario, reported by the author of Quicklisp. This problem is currently known to affect a couple of libraries.

> *Consider systems A and B, where A requires B to build. When Quicklisp test-builds A, the A prerequisites are built in such a way that B also successfully builds to satisfy A's requirements. But when Quicklisp test-builds B on its own, the environment is different in a way that precludes B from building. In that case, the metadata in Quicklisp specifies that A requires B, but B is not listed at all, because it does not build on its own.*

7.3 Library Ordering Refinements

In Section 5.3, we introduced the idea of *library batches*, that is, sets of libraries the loading of which wouldn't entail any compilation conflicts, and we mentioned the need to wait for batch exhaustion before sending in the next one. This requirement, which *is* a limitation, actually comes from the fact that only static information (namely, the dependency relations) is used to create the batches in question.

It is however possible to refine this idea. Indeed, the most pertinent information for us is *not* that library 1 depends on library 2, but that the compilation of library 2 is over. In other words: concurrent compilation is problematic; concurrent loading is not.

In order to improve on solution 3, we hence need one additional piece of (run-time) information: we need to be notified when a Declt thread has finished processing a library. The refinement can then go as follows. We create the same dependency graph as before, and also initialize the library queue with the first batch as before (the libraries with no dependencies), but this time, *without* removing them from the graph. From now on, as soon as a library is done processing by a Declt thread, we remove it from the graph. Any *new* leaf in the graph stemming from that removal can then safely be pushed immediately to the library queue.

An even better refinement would be to *not* wait for Declt to finish processing, but only for ASDF to finish compiling (this would require a communication channel between the main thread and the external Declt process though). This refinement has not been implemented yet, but it is a high priority, as we expect a somewhat substantial gain from it. Note also that it can be used in the alternative solution proposed in Section 7.1.

Currently, our dependency graph is implemented as a hash table of *adjacency lists*[3]: the hash keys are the library names, and the hash values are the lists of dependencies. Another possible (and classical) implementation consists in using an *adjacency matrix*[1], a potentially more compact representation. Whether one representation would be more beneficial than the other is currently unknown. In particular, more investigation on the dependencies morphology should be conducted, notably to discover whether the adjacency matrix would risk being sparse or not (very likely). In any case, the choice of representation is not expected to have much impact on the performance, again, because the dependency graph is relatively small (less than two thousand nodes), in front of at least 20 minutes of total processing.

7.4 CPU *vs.* I/O Consumption

While the performance improvements obtained from solution 1 are to be expected, getting *only* an improvement factor of 4 in solution 2 or 3 may appear somewhat surprising, even disappointing, especially since our test machine has 8 virtual cores (4 hyper-threaded actual cores). Of course, a factor of 8 would be unrealistic. Studies (from Intel or otherwise) have shown that an improvement of 30% is not unreasonable to expect from hyper-threading[2, 11]. The problem we have here is the fact that both Declt and Makeinfo are treated as monolithic black boxes, so we don't have any control on their CPU *vs.* I/O consumption, an otherwise important aspect of parallelization.

Declt works in two stages: first, an abstract in-memory representation of the documentation is constructed by introspecting the library. Next, the Texinfo file is generated from that abstract representation. The first stage is CPU-intensive, the second one is

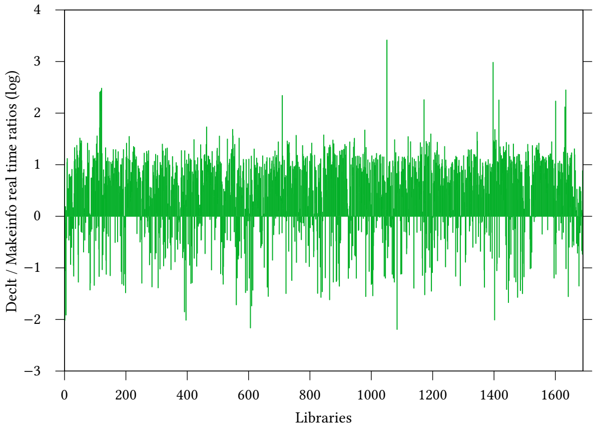


Figure 12: Declt / Makeinfo comparative timings

more pressing on I/O. On top of that, remember that the library needs to be loaded by Asdf first, possibly with some compilation. This will also entail several CPU or I/O intensive phases.

It turns out that Makeinfo works in a similar fashion, at least for Html production (for Pdf, TEX is used). The first stage reads the Texinfo file into an abstract in-memory representation. This stage is written in C, with a Perl interface. Then, the abstract representation is altered in various ways, and the Html file is finally created. This last stage is entirely done in Perl, and (according to the current Texinfo maintainers) is probably much slower than the previous ones.

Having no control over these different processing phases is unfortunate, and is likely to be the cause of the "25% threshold" that we seem to reach. It may very well happen, for instance, that regardless of the number of threads that we have, they all end up in an I/O phase at the same time, essentially waiting on the same disk, while subsequent CPU-intensive computation could have been started. Improving that situation would require cracking the Declt and Makeinfo "black boxes" open, possibly even the Asdf one, in order to introduce parallelism at a lower level. Although we already have some ideas about this, it would be a completely different project.

7.5 Scheduling

In addition to the points raised in the previous sections (the idea of library ordering in particular), the general question of scheduling could be raised. For example, we could get inspiration from the operating system theory, and think of improving things by minimizing the waiting time in the various queues, as the SJF (Shortest Job First) does in process scheduling[8]. The problem here is to get a notion of what makes for the complexity (hence the time) of the various tasks. Very preliminary investigation gives a somewhat pessimistic impression. For example, the collected experimental data shows that there is no correlation between Declt and Makeinfo processing time (see Figure 12). For some libraries, Declt takes more time than Makeinfo; for some others, it is the other way around, *etc.* In the worst case scenario, we would need to run Quickref and collect the data in question (which we did for this article), and use it on the next Quicklisp release, hoping that the situation didn't change too much.

7.6 SSD Technology

Finally, note that the validity of the present study is highly dependent on the fact that our test machine was equipped with a traditional, mechanical hard drive. We haven't had the opportunity to experiment with SSD (Solid State Drive[4]) technology yet, but their dramatically quicker access time and lower latency[5] is very likely to redefine the parameters of our study.

ACKNOWLEDGMENTS

The author would like to thank Zach Beane for his feedback on how Quicklisp handles dependencies across libraries, and Karl Berry, Gavin Smith, and Patrice Dumas for their insight into the inner workings of the Texinfo system.

REFERENCES

[1] Norman Biggs. *Algebraic Graph Theory.* Cambridge University Press, 1993. Definition 2.1, p. 7.

[2] Shawn D. Casey. How to determine the effectiveness of hyper-threading technology with an application. *Intel Technology Journal*, 6(1), 2011.

[3] Thomas Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2009.

[4] Neal Ekker, Tom Coughlin, and Jim Handy. An introduction to solid state storage. SNIA White Paper, January 2009.

[5] Vamsee Kasavajhala. Solid state drive vs. hard disk drive price and performance study. Dell Technical White Paper, May 2011.

[6] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968. Section 2.2.3.

[7] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), February 2002.

[8] Andrew S. Tannenbaum and Herbert Bos. *Modern Operating Systems.* Pearson, 4th edition, 2014. Section 2.4.

[9] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[10] Didier Verna. Standard i/o syntax, and the robustness principle. https://www.didierverna.net/blog/index.php?post/2017/10/27/Standard-IO-syntax-and-the-Robustness-Principle, November 2017. Blog Entry.

[11] Duc Vianney. Hyper-threading speeds linux. https://www.ibm.com/developerworks/library/l-htl/index.html, 2003.