Proceedings of the

9th European Lisp Symposium

AGH University of Science and Technology, Kraków, Poland
May 9 – 10, 2016
Irène Durand (ed.)

ISBN-13: 978-2-9557474-0-7
ISSN: 2677-3465

Contents

Demonstrations 91

Preface

Message from the Programme Chair

Welcome to the 9thth edition of the European Lisp Symposium!

From language implementation to applications through systems tools, language extensions and Domain Specific Languages, we have a great variety of contributions. Many dialects of the Lisp family are represented as well – Common Lisp, Racket, Clojure and Julia – showing the diversity of the community.
I thank the program committee members for their relevant reviews, their responsiveness and timeliness. I thank the steering commitee of ELS for entrusting me this task of which I had no previous experience. I thank Michał Psota for giving life to these contributions in organizing the conference in Kraków.
Many thanks to Didier Verna who assisted me all along the process, taking care of distributing calls and helping me whether for practical matters or for scientific decisions.
Finally, thanks to the authors, to all those that submitted contributions, and to the participants for keeping the community alive.

I hope you have a great time.

Irène Durand, Bordeaux, May 2016

Message from the Organizing Chair

Welcome to Kraków!

Kraków is one of the oldest cities in Poland, the place where many of the Polish kings lived and ruled, this was the capital city until the 18th century. According to an old legend the city was built by 'Krakus' above a cave occupied by a Dragon. The city is one of the leading centers of Polish academic, cultural, and artistic life. Today one can discover many attractions here – walking around the main market square, shopping at the old Cloth Hall, admiring the Wawel Castle – the residence of Polish Kings, listening to the famous trumpet signal from the top of St. Mary's Basilica that's being transmitted throughout Poland every day at noon through the national radio.

The AGH University of Science and Technology which is hosting us now, was established in 1919 as the University of Mining and Metallurgy. Today it is a respectable University with well-qualified staff, doing innovative research.

I am really glad so many people have registered and come to Poland to listen about the Lisp language family. I hope you will have good time not only at the lecture room, but also outside wandering around the city.

It would be really hard to organize a symposium without help. I would like to thank Didier Verna, his experience and patience to guide me through the process was invaluable (This is the first symposium I'm helping organizing). I would like to also thank "BIT" – a students' scientific association which is hosting us together with the Department of Computer Science. Last but not least I would like to thank all other sponsors for their support: EPITA, Université de Bordeaux, LispWorks, Franz Inc.

I hope you enjoy your time at the 9th European Lisp Symposium and those few days you are spending in Poland.

Michał Psota, Kraków, May 2016

Organization

Programme Chair

- Irène Anne Durand, University of Bordeaux, France

Local Chair

- Michał Psota — Emergent Network Defense, Kraków, Poland

Programme Committee

- Antonio Leitao — INESC-ID/IST, Universidade de Lisboa, Portugal
- Charlotte Heerzel — IMEC, Leuven, Belgium
- Christian Queinnec — University Pierre et Marie Curie, Paris 6, France
- Christophe Rhodes — Goldsmiths, University of London, United Kingdom
- Didier Verna — EPITA Research and Development Laboratory, France
- Erick Gallesio — University of Nice-Sophia Antipolis, France
- François-René Rideau, Google, USA
- Giuseppe Attardi — University of Pisa, Italy
- Kent Pitman, HyperMeta Inc., USA
- Leonie Dreschler-Fischer — University of Hamburg, Germany
- Pascal Costanza — Intel Corporation, Belgium
- Robert Strandh — University of Bordeaux, France

Organizing Committee

- Michał Psota — Emergent Network Defense, Kraków, Poland
- Maciej Ciołek — Scientific Society BIT, AGH University of Science and Technology, Kraków, Poland

Sponsors

We gratefully acknowledge the support given to the 9thth European Lisp Symposium by the following sponsors:

AGH
University of Science and Technology
Kraków, Poland
`www.agh.edu.pl/en`

Kolo Naukowe
Kraków, Poland `http://knbit.edu.pl/en/`

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

Université de Bordeaux
Excellence Initiative
Collège Science et Technologie
351 cours de la libération
33405 Talence Cedex
France
`www.u-bordeaux.fr`

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

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

Invited Contributions

Program Proving with Coq

Pierre Castéran, LaBRI, University of Bordeaux, France

This talk aims to present some aspects of the *Coq* proof assistant, mainly the possibility of verifying and/or synthesizing correct programs.

The talk will start with an introduction to the main features of *Coq*, both as a statically typed functional programming language and as a proof assistant, *i.e.* a tool for helping its user to prove theorems, including — but not limited to — correctness statements about his/her programs.

In the second part of the talk, we propose to use the efficient computation of powers of the form x^n as a pretext for presenting various techniques for obtaining *certified* programs within the *Coq* system, with various levels of automaticity. Besides "classical" proof methods, we will present briefly some approaches and compare them with respect to their ability to efficiently return correctness proofs.

- Proof by reflection,
- The "refinement for free" approach,
- Composition of certified components (correctness by construction)

Pierre Castéran is an associate professor at the University of Bordeaux. His research interests are type theory, interactive theorem proving and programming language semantics. He co-authored the book on Coq: "Interactive Theorem Proving and Program Development" with Yves Bertot, and is the scientific co-organizer of 7 international summer schools on Coq, 2 in Paris, 4 in China, and 1 in Japan. He also co-authored two tutorials on advanced features of Coq: inductive and co-inductive types, type classes and generalized rewriting. Pierre Castéran is now working on a sequel of the aforementionned book, dedicated to the presentation of recent trends in using the Coq proof assistant. Pierre Castéran is one of the nine co-laureates of the ACM Software Systems Award given to the Coq Proof Assistant (2013).

* * * * *

Julia: to Lisp or Not to Lisp?

Stefan Karpinski, Julia Computing and New York University, New York, USA

Julia is a general purpose dynamic language, designed to make numerical computing fast and convenient. Many aspects of Julia should be quite familiar since they are "stolen" straight from Lisp: it's expression-oriented, lexically scoped, has closures, coroutines, and macros that operate on code as data. But Julia departs from the Lisp tradition in other ways. Julia has syntax – lots of it. Macro invocations look different than function calls. Some dynamic behaviors are sacrificed to make programs easier to analyze (for both humans and compilers), especially where it allows simpler, more reliable program optimization.

Julia's most distinctive feature is its emphasis on creating lightweight types and defining their behavior in terms of generic functions. While many Lisps support multiple dispatch as an opt-in feature, in Julia all function are generic by default. Even basic operators like '+' are generic, and primitive types like 'Int' and 'Float64' are defined in the standard library, and their behavior is specified via multiple dispatch. A combination of aggressive method specialization, inlining and data-flow-based type inference, allow these layers of abstraction and dispatch to be eliminated when it counts – Julia generally has performance comparable to static languages. In the tradition of the great Lisp hackers, this talk will include lots of live coding in the REPL, with all the excitement, and possibility of failure entailed.

Stefan Karpinski is one of the co-creators of the Julia programming language and a co-founder of Julia Computing, Inc., which provides support, consulting and training for commercial usage of Julia. He has previously worked as a software engineer and data scientist at Akamai, Citrix, andEtsy. He is currently a Research Engineer at NYU as part of the Moore-Sloan Data Science Initiative.

* * * * *

Lexical Closures and Complexity

Francis Sergeraert, Institut Fourier, Grenoble, France

The power of Common Lisp for functional programming is well known, the key tool being the notion of "lexical closure", allowing the programmer to write programs which, during execution, *dynamically* generate functional objects of arbitrary complexity. Using this technology, new algorithms in Algebraic Topology have been discovered, implemented in Common Lisp, used, producing homology and homotopy groups so far unreachable. An algorithm is viewed as "tractable" if its theoretical complexity is not worse than polynomial. The study of this complexity for the aforementioned algorithms of Algebraic Topology requires a lucid knowledge of the concrete implementation of these lexical closures. The talk is devoted to a report about a result of polynomial complexity so obtained. The scope of the method is general and in particular no knowledge in Algebraic Topology is expected in the audience.

Francis Sergeraert got a PhD in Differential Analysis. Since 1984, his research concern the Algorithmics of Algebraic Topology. The Kenzo Common Lisp program (public domain) has been written down with several co-workers ; it is entirely devoted to high level Algebraic Topology (homology, homotopy, spectral sequences, ...). Kenzo is currently the only object having been able to compute some homology and homotopy groups.

* * * * *

Session I: Language design

Refactoring Dynamic Languages

Rafael Reia
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
rafael.reia@tecnico.ulisboa.pt

António Menezes Leitão
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

ABSTRACT

Typically, beginner programmers do not master the style rules of the programming language they are using and, frequently, do not have yet the logical agility to avoid writing redundant code. As a result, although their programs might be correct, they can also be improved and it is important for the programmer to learn about the improvements that, without changing the meaning of the program, simplify it or transform it to follow the style rules of the language. These kinds of transformations are the realm of refactoring tools. However, these tools are typically associated with sophisticated integrated development environments (IDEs) that are excessively complex for beginners.

In this paper, we present a refactoring tool designed for beginner programmers, which we made available in DrRacket, a simple and pedagogical IDE. Our tool provides several refactoring operations for the typical mistakes made by beginners and is intended to be used as part of their learning process.

Keywords

Refactoring Tool, Pedagogy, Racket

1. INTRODUCTION

In order to become a proficient programmer, one needs not only to master the syntax and semantics of a programming language, but also the style rules adopted in that language and, more important, the logical rules that allow him to write simple and understandable programs. Given that beginner programmers have insufficient knowledge about all these rules, it should not be surprising to verify that their code reveals what more knowledgeable programmers call "poor style," "bad smells," etc. As time goes by, it is usually the case that the beginner programmer learns those rules and starts producing correct code written in an adequate style. However, the learning process might take a considerable amount of time and, as a result, large amounts of poorly-written code might be produced before the end of the process. It is then important to speed up this learning process by showing, from the early learning phases, how a poorly-written fragment of code can be improved.

After learning how to write code in a good style, programmers become critics of their own former code and, whenever they have to work with it again, they are tempted to take advantage of the opportunity to restructure it so that it conforms to the style rules and becomes easier to understand. However, in most cases, these modifications are done without complete knowledge of the requirements and constraints that were considered when the code was originally written and, as result, there is a serious risk that the modifications might introduce bugs. It is thus important to help the programmer in this task so that he can be confident that the code improvements he anticipate are effectively applicable and will not change the meaning of the program. This has been the main goal of *code refactoring*.

Code refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [1]. Nowadays, any sophisticated IDE includes an assortment of refactoring tools, e.g., for moving methods along a class hierarchy, to extract interfaces from classes, and to transform anonymous classes into nested classes. It is important to note, however, that these IDEs were designed for advanced programmers, and that the provided refactorings require a level of code sophistication that is not present in the programs written by beginners. This makes the refactoring tools inaccessible to beginners.

In this paper, we present a tool that was designed to address the previous problems. In particular, our tool (1) is usable from a pedagogical IDE designed for beginners [2, 3], (2) is capable of analyzing the programmer's code and inform him of the presence of the typical mistakes made by beginners, and finally (3) can apply refactoring rules that restructure the program without changing its semantics.

To evaluate our proposal, we implemented a refactoring tool in DrRacket, a pedagogical IDE [4, 5] used in schools around the world to teach basic programming concepts and techniques. Currently, DrRacket has only one simple refactoring operation which allows renaming a variable. Our work significantly extends the set of refactoring operations available in DrRacket and promotes their use as part of the learning process.

2. RELATED WORK

There are many refactoring tools available. The large majority of these tools were designed to deal with large

9th ELS May 9–10, 2016, Kraków, Poland

statically-typed programming languages such a Java or C++ and are integrated in the complex IDEs typically used for the development of complex software projects, such as Eclipse or Visual Studio.

On the other hand, it is a common practice to start teaching beginner programmers using dynamically-typed programming languages, such as Scheme, Python, or Ruby, using simple IDEs. As a result, our focus was on the dynamic programming languages which are used in introductory courses and, particularly, those that promote a functional programming paradigm.

In the next sections, we present an overview of the refactoring tools that were developed for the languages used in introductory programming courses.

2.1 Scheme

In its now classical work [6], Griswold presented a refactoring tool for Scheme that uses two different kinds of information, namely, an Abstract Syntax Tree (AST) and a Program Dependence Graph (PDG).

The AST represents the abstract syntactic structure of the program, while the PDG explicitly represents the key relationship of dependence between operations in the program. The graph vertices's represent program operations and the edges represent the flow of data and control between operations. However, the PDG only has dependency information of the program and relying only in this information to represent the program could create problems. For example, two semantically unrelated statements can be placed arbitrarily with respect to each other. Using the AST as the main representation of the program ensures that statements are not arbitrarily reordered, allowing the PDG to be used to prove that transformations preserve the meaning and as a quick way to retrieve needed dependence information. Additionally, contours are used with the PDG to provide scope information, which is non existent in the PDG, and to help reason about transformations in the PDG. With these structures it is possible to have a single formalism to reason effectively about flow dependencies and scope structure.

2.2 Python

Rope[7, p. 109] is a Python refactoring tool written in Python, which works like a Python library. In order to make it easier to create refactoring operations, Rope assumes that a Python program only has assignments and function calls. Thus, by limiting the complexity of the language it reduces the complexity of the refactoring tool.

Rope uses a Static Object Analysis, which analyses the modules or scopes to get information about functions. Because its approach is time consuming, Rope only analyses the scopes when they change and it only analyses the modules when asked by the user.

Rope also uses a Dynamic Object Analysis that requires running the program in order to work. The Dynamic Object Analysis gathers type information and parameters passed to and returned from functions. It stores the information collected by the analysis in a database. If Rope needs the information and there is nothing on the database the Static object inference starts trying to infer it. This approach makes the program run much slower, thus it is only active when the user allows it. Rope uses an AST in order to store the syntax information about the programs.

Bicycle Repair Man[1] is another refactoring tool for Python and is written in Python itself. This refactoring tool can be added to IDEs and editors, such as Emacs, Vi, Eclipse, and Sublime Text. It attempts to create the refactoring browser functionality for Python and has the following refactoring operations: extract method, extract variable, inline variable, move to module, and rename.

The tool uses an AST to represent the program and a database to store information about several program entities and their dependencies.

Pycharm Educational Edition,[2] or Pycharm Edu, is an IDE for Python created by JetBrains, the creator of IntelliJ. The IDE was specially designed for educational purposes, for programmers with little or no previous coding experience. Pycharm Edu is a simpler version of Pycharm community which is the free python IDE created by JetBrains. It is very similar to their complete IDEs and it has interesting features such as code completion and integration with version control tools. However, it has a simpler interface than Pycharm Community and other IDEs such as Eclipse or Visual Studio.

Pycharm Edu integrates a python tutorial and supports teachers that want to create tasks/tutorials for the students. However, the refactoring tool did not received the same care as the IDE itself. The refactoring operations are exactly the same as the Pycharm Community IDE which were made for more advanced users. Therefore, it does not provide specific refactoring operations to beginners. The embedded refactoring tool uses the AST and the dependencies between the definition and the use of variables, known as def-use relations.

2.3 Javascript

There are few refactoring tools for JavaScript but there is a framework [8] for refactoring JavaScript programs. In order to guarantee the correctness of the refactoring operation, the framework uses preconditions, expressed as query analyses provided by pointer analysis. Queries to the pointer analysis produces over-approximations of sets in a safe way to have correct refactoring operations. For example, while doing a rename operation, it over-approximates the set of expressions that must be modified when a property is renamed in a safe manner.

To prove the concept, three refactoring operations were implemented, namely rename, encapsulate property, and extract module. By using over-approximations it is possible to be sure when a refactoring operation is valid. However, this approach has the disadvantage of not applying every possible refactoring operation, because the refactoring operations for which the framework cannot guarantee behavior preservation are prevented. The wrongly prevented operations accounts for 6.2% of all rejections.

2.4 Analysis

The Table 1 summarizes the data structures of the analyzed refactoring tools. It is clear that the AST of a program is an essential part of the refactoring tool information with every refactoring tool having an AST to represent the program. Regarding the PDG and Database it contains mainly information about the def-use-relation of the program. The PDG has also control flow information of the program. Our

[1] https://pypi.python.org/pypi/bicyclerepair/0.7.1

[2] https://www.jetbrains.com/pycharm-edu/

Table 1: Data Structures

Name	AST	PDG	Database	Others
Griswold	X	X		
Rope	X		X	
Bicycle	X		X	
Pycharm Edu	X		X	
Javascript				X

implementation uses the same data structures, the AST and the def-use-relations.

Some tools like the one build by Griswold focus on the correctness of the refactoring operations and therefore need more information about the program, such as the information provided by the PDG. Others, focus in offering refactoring operations for professional or advanced users. However, the goal of our refactoring tool is to provide refactoring operations designed for beginners. Therefore we are not interested in refactoring operations formerly proven correct or provide refactoring operations only used in advanced and complex use cases.

We intend to have simple, useful, and correct refactoring operations to correct the typical mistakes made by beginners. With this we exclude from the refactoring tool scope macros usage, classes, and other complex structures not often used by beginners.

3. ARCHITECTURE

In this section we present the architecture of our refactoring tool. Although the tool can work with different programming languages, its main focus is, at this moment, the Racket programming language and, more specifically, the DrRacket IDE.

Racket is a language designed to support meta-programming and, in fact, most of the syntax forms of the language are macro-expanded into combinations of simpler forms. This has the important consequence that programs can be analyzed either in their original form or in their expanded form.

In order to create correct refactoring operations, the refactoring tool uses two sources of information, the def-use relations and the AST of the program. The def-use relations represent the links between the definition of an identifier and its usage. In the DrRacket IDE these relations are visually represented as arrows that point from a definition to its use. The opposite relation, the use-def relation, is also visually represented as an arrow from the use of an identifier to its definition. The AST is the abstract syntax tree of the program which, in the case of the Racket language, is represented by a list of syntax-objecs.

Figure 1 summarizes the workflow of the refactoring tool, where the Reader produces the non expanded AST of the program while the Expander expands the AST produced by the Reader. In order to produce the def-use relations it is necessary to use the expanded AST produced by the Expander because it has the correct dependency information. The Transformer uses the Code Walker to parse the ASTs and the information of the def-use relations to correctly perform the refactoring operations. Then it goes to the Writing module to produce the output in DrRacket's definitions pane.

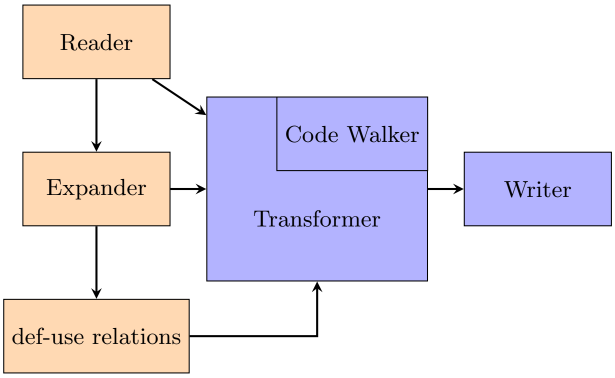


Figure 1: Main modules and information flow between modules. Unlabeled arrows represent information flow between modules.

3.1 Syntax Expressions

The syntax-object list represents the AST, which provides information about the structure of the program. The syntax-object list is already being produced and used by the Racket language and, in DrRacket, in order to provide error information to the user. DrRacket already provides functions which computes the program's syntax-object list and uses some of those functions in the Background Check Syntax and in the Check Syntax button callback.

3.1.1 Syntax Expression tree forms

DrRacket provides functions to compute the syntax-object list in two different formats. One format is the expanded program, which computes the program with all the macros expanded. The other format is the non-expanded program and computes the program with the macros unexpanded.

The expanded program has the macros expanded and the identifier information correctly computed. However, it is harder to extract the relevant information when compared with the non expanded program.

For example, the following program is represented in the expanded form, and in the non expanded form.

Listing 1: Original Code

```
(and alpha beta)
```

Listing 2: Expanded program

```
#<syntax:2:0
 (#%app call-with-values
 (lambda ()
    (if alpha beta (quote #f)))
  print-values)>
```

Listing 3: Non-expanded program

```
#<syntax:2:0 (and alpha beta)>
```

Note that the expanded program transforms the `and`, `or`, `when`, and `unless` forms into `if`s which makes refactoring operations harder to implement.

Racket adds internal representation information to the expanded program which for most refactoring operations is not necessary. In addition, the expanded program has a format that is likely to change in the future. Racket is an evolving language and the expanded form is a low level and internal form of representation of the program. However, the expanded program has important information regarding the binding information that is not available in the non-expanded form, and this information might be useful, e.g., to detect if two identifiers refer to the same binding. Additionally, we do not consider macro definitions as part of the code that needs to be refactored, since the refactoring tool is targeted at unexperienced programmers and these programmers typically do not define macros.

Taking the previous discussion into consideration, it becomes clear that it is desirable to use the non expanded form for the refactoring operations whenever possible and use the expanded form only when needed.

3.2 Def-use relations

Def-use relations hold important information needed in order to produce correct refactoring operations. They can be used to check whether there will be a duplicated name or to compute the arguments of a function that is going to be extracted.

The def-use-relations is computed by the compiler that runs in the background. However, it is only computed when a program is syntactically correct.

3.3 Code-walker

The code-walker is used to parse the syntax tree represented by a syntax element that is a list of syntax-object in Racket. A syntax-object can contain either a symbol, a syntax-pair, a datum (number, boolean or string), or an empty list. While a syntax-pair is a pair containing a syntax-object as it first element and either a syntax pair, a syntax element or an empty list as the second argument. Each syntax-object has information about the line where they are defined and this information is used to search for the correct elements.

Most of the time, the code-walker is used to search for a specific syntax element and the location information contained in the syntax-object is used to skip the syntax blocks that are before the syntax element wanted in the first place.

The Code-walker is a core part of the refactoring tool, ensuring that the selected syntax is correctly fed to the refactoring operations.

3.4 Pretty-printer

Producing correct output is an important part of the refactoring tool. It is necessary to be careful to produce indented code and we decided to use a pretty-printer that is already available in the Racket language. However, it should be noted that this pretty-printer does not follow some of the Racket style convention, such as `cond` clauses surrounded by square brackets. This is not considered a problem because Racket supports both representations. One possible solution is to use a different pretty-printer in order to keep the language conventions.

3.5 Comments preservation

Preserving the comment information after a refactoring transformation is an important task of the refactoring tool. If the comment in determined place of the program changes its location, affecting a different part of the program, it could confuse the programmer. However, comment preservation is not implemented yet, making it a limitation of this prototype.

One possible solution is to modify the syntax reader and add a comment node to the AST. While the new node will not be used during refactoring transformations it is used during the output part of the refactoring operation, preserving the comment with the correct syntax expression.

3.6 Syntax-Parse

The Syntax-Parse[9] function provided by Racket is very useful for the refactoring operations. It provides a wide range of options to help matching the correct syntax, using backtracking to allow several rules to be matched in the same syntax parser, which helps to create more sophisticated rules.

4. REFACTORING OPERATIONS

In this section we explain some of the more relevant refactoring operations and some limitations of the refactoring tool.

4.1 Semantic problems

There are some well-known semantic problems that might occur after doing a refactoring operation. One of them occurs in the refactoring operation that removes redundant `and`s in numeric comparisons. Although rarely known by beginner programmers, in Racket, numeric comparisons support more than two arguments, as in `(< 0 ?x 9)`, meaning the same as `(and (< 0 ?x) (< ?x 9)`, where, we use the notation `?x` to represent an expression. Thus, it is natural to think about a refactoring operation that eliminates the `and`. However, when the `?x` expression somehow produces side-effects, the refactoring operation will change the meaning of the program.

Despite this problem, we support this refactoring operation because, in the vast majority of the cases, there are no side-effects being done in the middle of numerical comparisons. This is explained by the fact that it is rare to have the same argument repeated in a comparison and, moreover, the short-circuit evaluation rule does not guarantee that the side effects will be done. Therefore, it is usually safe to apply this refactoring operation.

Another example of a semantic problem occurs when refactoring the following `if` expression.

Listing 4: Code sample

```
(if ?x
    (begin ?y ...)
    #f)
```

There are two different refactoring transformations possible:

Listing 5: Refactoring option 1

```
(when ?x
  ?y ...)
```

Listing 6: Refactoring option 2

```
(and ?x (begin ?y ...))
```

Note that the first refactoring option changes the meaning of the program, because if the test expression, in this case `?x`, is false, the result of the `when` expression is `#<void>`.

However, the programmer may still want to choose the first refactoring option if the return value when the `?x` is false is not important.

4.2 Extract Function

Extract function is an important refactoring operation that every refactoring tool should have. In order to extract a function it is necessary to compute the arguments needed to the correct use of the function. While giving the name to a function seems quite straightforward, it is necessary to check for name duplication in order to produce a correct refactoring as having two identifiers with the same name in the same scope produces an incorrect program. After the previous checks, it is straightforward to compute the function body and replacing the original expression by the function call.

However, the refactoring raises the problem where should the function be extracted to. A function can not be defined inside an expression, but it can be defined at the top-level or at any other level that is accessible from the current level.

As an example, consider the following program:

Listing 7: Extract function levels

```
;;top-level
(define (level-0)
  (define (level-1)
    (define (level-2)
      (+ 1 2))
    (level-2))
  (level-1))
```

When extracting the `(+ 1 2)` to a function where should it be defined? Top-level, Level-0, level-1, or in the current level, the level-2? The fact is that is extremely difficult to know the answer to this question because it depends on what the user is doing and the user intent. Accordingly, we decided that the best solution is to let the user decide where the user wants the function to be defined.

4.2.1 Computing the arguments

In order to compute the function call arguments we have to know in which scope the variables are being defined, in other words, if the variables are defined inside or outside the extracted function. The variables defined outside the function to be extracted are candidates to be the arguments of that function. However, imported variables, whether from the language base or from other libraries do not have to be passed as arguments. To solve this problem, we considered two possible solutions:

- Def-use relations + Text information
- Def-use relations + AST

The first approach is simpler to implement and more direct than the second one. However, it is less tolerant to future changes and to errors. The second one combines the def-use relations information with the syntax information to check whether it is imported from the language or from other library.

We choose the second approach in order to provide a more stable solution to correctly compute the arguments of the new function.

4.3 Let to Define Function

A `let` expression is very similar to a function, which may led the user to mistakenly use one instead of the other. Therefore we decided to provide a refactoring operation that would make such transition simpler.

There are several let forms, but since we want to explore the similarity between the `let` and the function we are going to focus in the ones that are more similar to a function, namely the `let` and the `named let`.

There are some differences between them, the `named let` can be directly mapped to a named function, using the `define` keyword, whereas the `let` can only be directly mapped to an anonymous function, `lambda`. We decided to focus first in the transformation of a `named let` to a function.

However, this refactoring operation which transforms a `named let` into a `define` function, could have syntax problems since a `let` form can be used in expressions, but the `define` can not. In the vast majority of cases this refactoring is correct, however, when a `named let` is used in an expression it transforms the program in an incorrect one. e.g.

Listing 8: Let in an expression

```
(and (let xpto ((a 1)) (< a 2)) (< b c))
```

Modifying this `named let` into a `define` would raise a syntax error since a `define` could not be used in an expression context. Encapsulate the `define` with the `local` keyword, which is an expression like the `named let`, can solve the problem. However, the `local` keyword is not used very often and might confuse the users. Therefore we decided keep the refactoring operation without the local keyword that works for most of the cases.

Listing 9: Let example

```
(let loop ((x 1))
  (when (< x 10)
    (loop (+ x 1))))
```

Listing 10: Let to Define Function example

```
(define (loop x)
  (when (< x 10)
    (loop (+ x 1))))
(loop 1)
```

4.4 Wide-Scope Replacement

The Wide-Scope replacement extends the extract function functionality by replacing the duplicated of the extracted code with the function call of the extracted function.

The Wide-Scope replacement refactoring operation searches for the code that is duplicated of the extracted function and then replaces it for the call of the extracted function and it is divided in two steps:

- Detect duplicated code
- Replace the duplicated code

Replacing the duplicated code is the easy part, however the tool might has to compute the arguments for the duplicated code itself.

Correctly detecting duplicated code is a key part for the correctness of this refactoring. Even the simplest form of duplicated code detection, where it only detects duplicated code when the code is exactly equal, may have some problems regarding the binding information. For example, if the duplicated code is inside a `let` that changes some bindings that must be taken into consideration. In order to solve the binding problem we can use functions already provided in Racket. However, that does not work if we use the program in the not expanded form to do the binding comparisons because there is not enough information for those bindings to work. Therefore, in order to compute the correct bindings, it is necessary to use the expanded form of the program.

The naive solution is to use the expanded program to detect the duplicated code and then use this information to do the replacing of the duplicated code. However, when expanding the program Racket adds necessary internal information to run the program itself that are not visible for the user. While this does not change the detecting of the duplicated code, it adds unnecessary information that would have to be removed. In order to solve this in a simple way we can use the expanded code to detect the correctly duplicated code and use the non expand program to compute which code will be replaced.

However, this duplicated code detection is a quadratic algorithm which might have some performance problems for bigger programs.

5. FEATURES

This section describes some of the features created to improve the usability by providing sufficient feedback to the user, and way to inform the user of the presence of the typical mistakes made by beginners.

5.1 User FeedBack

It is important to give proper feedback to the user while the user is attempting or preforming a refactoring operation. Previewing the outcome of a refactoring operation is an efficient form to help the users understand the result of a refactoring before even applying the refactoring. It works by applying the refactoring operation in a copy version of the AST and displaying those changes to the user.

5.2 Automatic Suggestions

Beginner programmers usually do not know which refactoring operations exist or which can be applied. By having a automatic suggestion of the possible refactoring operations available the beginner programmer can have an idea what refactoring operations can be applied or not.

In order to detect possible refactoring operations it parses the code from the beginning to the end and tries to check if a refactoring is applicable. To do that it tries to match every syntax expression using syntax parse. In other words it uses brute force to check whether a expression can be applied a refactoring operation or not.

To properly display this information it highlights the source-code indicating that there is a possible refactoring. This feature could be improved by having a set of colors for the different types of refactoring operations. Moreover, the color intensity could be proportional to the level of suggestion. e.g the recommended level to use extract function refactoring increases with the number of duplicated code found.

6. EVALUATION

In this section we present some code examples written by beginner programmers in their final project of an introductory course and their possible improvements using the refactoring operations available in our refactoring tool. The examples show the usage of some of the refactoring operations previously presented and here is explained the motivation for their existence.

The first example is a very typical error made by beginner programmers.

```
(if (>= n_plays 35)
  #t
  #f)
```

It is rather a simple refactoring operation, but nevertheless it improves the code.

```
(>= n_plays 35)
```

The next example is related with the conditional expressions, namely the `and` or `or` expressions. We decided to choose the `and` expression to exemplify a rather typical usage of this expression.

```
(and
 (and
  (eq? #t (correct-movement? player play))
  (eq? #t (player-piece? player play)))
 (and
  (eq? #t (empty-destination? play))
  (eq? #t (empty-start? play))))
```

Transforming the code by removing the redundant `and` expression makes the code cleaner and simpler to understand.

```
(and (eq? #t (correct-movement? player play))
     (eq? #t (player-piece? player play))
     (eq? #t (empty-destination? play))
     (eq? #t (empty-start? play)))
```

However, this code can still be improved, the `(eq? #t ?x)` is a redundant way of simple writing `?x`.

```
(and (correct-moviment? player play)
     (player-piece? player play)
     (empty-destination? play)
     (empty-start? play))
```

While a student is still learning, it is common to forget whether or not a sequence of expressions need to be wrapped in a `begin` form. The `when`, `cond` and `let` expressions have a implicit `begin` and as a result it is not necessary to add the `begin` expression. Moreover, sometimes they still keep the `begin` keyword because they often use a trial and error approach in writing code. Our refactoring tool checks for those mistakes and corrects them.

```
(if (odd? line-value)
    (let ((internal-column (sub1 (/ column 2))))
      (begin
        (if (integer? internal-column)
            internal-column
            #f)))
    (let ((internal-column (/ (sub1 column) 2)))
      (begin
        (if (integer? internal-column)
            internal-column
            #f))))
```

This is a simple refactoring operation and it does not have a big impact, however it makes the code clear and helps the beginner programmer to learn that a `let` does not need a `begin`.

```
(if (odd? line-value)
    (let ((internal-column (sub1 (/ column 2))))
      (and (integer? internal-column)
          internal-column))
    (let ((internal-column (/ (sub1 column) 2)))
      (and (integer? internal-column)
          internal-column)))
```

The next example shows a nested `if`. Nested `if`s are difficult to understand, error prone, and a debugging nightmare.

```
(define (search-aux? board line column piece)
  (if (> column 8)
    #f
    (if (= line 1)
      (if (eq? (house-board board 1 column) piece)
        #t
        (search-aux? board line (+ 2 column) piece))
      (if (= line 2)
        (if (eq? (house-board board 2 column) piece)
          #t
          (search-aux? board line (+ 2 column) piece))
        (if (= line 3)
          (if (eq? (house-board board 3 column) piece)
            #t
            (search-aux? board line (+ 2 column) piece))
          (if (= line 4)
            (if (eq? (house-board board 4 column) piece)
              #t
              (search-aux? board line (+ 2 column) piece))
            (if (= line 5)
              (if (eq? (house-board board 5 column) piece)
                #t
                (search-aux? board line (+ 2 column) piece))
              (if (= line 6)
                (if (eq? (house-board board 6 column) piece)
                  #t
                  (search-aux? board line (+ 2 column) piece))
                (if (= line 7)
                  (if (eq? (house-board board 7 column) piece)
                    #t
                    (search-aux? board line (+ 2 column) piece))
                  (if (= line 8)
                    (if (eq? (house-board board 8 column) piece)
                      #t
                      (search-aux? board line (+ 2 column) piece))
                    null))))))))))
```

It is much simpler to have a `cond` expression instead of the nested `if`. In addition, every true branch of this nested if contains `if` expressions that are or expressions and by refactoring those `if` expressions to `or`s it makes the code simpler to understand.

```
(define (search-aux? board line column piece)
  (cond [(> column 8) #f]
        [(= line 1)
         (or (eq? (house-board board 1 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 2)
         (or (eq? (house-board board 2 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 3)
         (or (eq? (house-board board 3 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 4)
         (or (eq? (house-board board 4 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 5)
         (or (eq? (house-board board 5 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 6)
         (or (eq? (house-board board 6 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 7)
         (or (eq? (house-board board 7 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 8)
         (or (eq? (house-board board 8 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [else null]))
```

Table 2: Refactoring Operations

Code #	1	2	3	4	5	6	7	8	Total
Initial LOC	582	424	332	1328	810	569	798	614	4045
Final LOC	545	373	300	1259	701	527	733	457	3705
Difference	37	51	32	69	109	42	65	140	340
Percentage	-6.36	-12.03	-9.64	-5.19	-13.46	-7.38	-8.14	-22.80	-10.63
Remove Begin	11	4	6	9	5	2	0	7	44
If to When	4	0	0	0	0	7	0	0	11
If to And	3	1	0	0	0	2	0	0	6
If to Or	6	6	1	13	20	3	2	0	51
Remove If	0	3	7	6	3	5	0	2	26
Remove And	0	2	0	4	0	0	0	0	6
Remove Eq	0	4	0	0	0	0	0	0	4
Extract function	0	3	0	0	4	0	1	5	13
If to Cond	0	0	0	2	3	0	1	0	6

However, this code could still be further improved by refactoring it into a `case`.

The examples presented above appear repeatedly in almost every code submission of this final project supporting the need to proper support to beginner programmers.

As seen in Table 2 the average reduction in LOC is 10.63% which shows how useful are these refactoring operations. It also shows how many refactoring operations were applied. This tool is not only for beginners, during the development of the tool we already used some of the refactoring operations, namely the extract function, in order to improve the structure of the code.

6.1 Refactoring Python

Python is being promoted as a good replacement for Scheme and Racket in science introductory courses. It is an high-level, dynamically typed programming language and it supports the functional, imperative and object oriented paradigms. Using the architecture of this refactoring tool and the capabilities offered by Racket combined with an implementation of Python for Racket[10] [11] it is also possible to provide refactoring operations in Python.

Using Racket's syntax-objects to represent Python as a meta-language [12] it is possible use the same structure used for the refactoring operations in Racket to parse and analyze the code in Python.

However, there are some limitations regarding the refactoring operations in Python. Since Python is a statement base language instead of expression base, it raises some problems regarding the possibility of some refactoring operations.

Example of removing an If expression:

```
True if (alpha < beta) else False
```

```
(alpha < beta)
```

The next one shows an extract function:

```python
def mandelbrot(iterations, c):
    z = 0+0j
    for i in range(iterations+1):
        if abs(z) > 2:
            return i
        z = z*z + c
    return i+1
```

```python
def computeZ(z, c):
    return z*z + c

def mandelbrot(iterations, c):
    z = 0+0j
    for i in range(iterations+1):
        if abs(z) > 2:
            return i
        z = computeZ( z,c)
    return i+1
```

It is important to note that this refactoring operations for Python are just only a prototype and the work is still in progress.

7. CONCLUSION

A refactoring tool designed for beginner programmers would benefit them by providing a tool to restructure the programs and improve what more knowledgeable programmers call “poor style,” “bad smells,” etc. In order to help those users it is necessary to be usable from a pedagogical IDE, to inform the programmer of the presence of the typical mistakes made by beginners, and to correctly apply refactoring operations preserving semantics.

Our solution tries to help those users to improve their programs by using the AST of the program and the def-use-relations to create refactoring operations that do not change the program’s semantics. This structure is then used to analyze the code to detect typical mistakes using automatic suggestions and correct them using the refactoring operations provided.

There are still some improvements that we consider important for the user. Firstly, the detection of duplicated code is still very naive and improving the detection in order to understand if two variables represent the same even if the names are different or even if the order of some commutative expressions is not the same would make a huge improvement on the automatic suggestion. Then it is possible to improve the automatic suggestion of refactoring operations by having different colors for different types of refactoring operations. With a lower intensity for low ”priority” refactoring operations and a high intensity for higher ”priority”. Thus giving the user a better knowledge of what is a better way to solve a problem or what is a strongly recommendation to change the code. Lastly it would be interesting to detect when a developer is refactoring in order to help the developer finish the refactoring by doing it automatically [13].

8. ACKNOWLEDGMENTS

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

9. REFERENCES

[1] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[2] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, 39(4):204–223, 2007.

[3] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[4] Clements J. Flanagan C. Flatt M. Krishnamurthi S. Steckler P. & Felleisen M. Findler, R. B. DrScheme: A programming environment for Scheme. *Journal of functional programming*, 12(2):159–182, 2002.

[5] Flanagan C. Flatt M. Krishnamurthi S. & Felleisen M. Findler, R. B. DrScheme: A pedagogic programming environment for Scheme. *Programming Languages: Implementations, Logics, and Programs*, 12(2):369–388, 1997.

[6] William G Griswold. Program restructuring as an aid to software maintenance. 1991.

[7] Siddharta Govindaraj. *Test-Driven Python Development*. Packt Publishing Ltd, 2015.

[8] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. *ACM SIGPLAN Notices*, 46(10):119–138, 2011.

[9] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

[10] Pedro Palma Ramos and António Menezes Leitão. An implementation of python for racket. In *7 th European Lisp Symposium*, 2014.

[11] Pedro Palma Ramos and António Menezes Leitão. Reaching python from racket. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, page 32. ACM, 2014.

[12] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 154–164. IEEE, 2000.

[13] Xi Ge, Quinton L DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 211–221. IEEE, 2012.

Type-Checking of Heterogeneous Sequences in Common Lisp

Jim E. Newton
jnewton@lrde.epita.fr

Akim Demaille
akim@lrde.epita.fr

Didier Verna
didier@lrde.epita.fr

EPITA/LRDE
14-16 rue Voltaire
F-94270 Le Kremlin-Bicêtre
France

ABSTRACT

We introduce the abstract concept of *rational type expression* and show its relationship to rational language theory. We further present a concrete syntax, *regular type expression*, and a Common Lisp implementation thereof which allows the programmer to declaratively express the types of heterogeneous sequences in a way which is natural in the Common Lisp language. The implementation uses techniques well known and well founded in rational language theory, in particular the use of the Brzozowski derivative and deterministic automata to reach a solution which can match a sequence in linear time. We illustrate the concept with several motivating examples, and finally explain many details of its implementation.

CCS Concepts

•**Theory of computation** → **Regular languages;** *Automata extensions;* •**Software and its engineering** → **Data types and structures;** *Source code generation;*

Keywords

Rational languages, Type checking, Finite automata

1. INTRODUCTION

In Common Lisp [15] a *type* is identically a set of (potential) values at a particular point in time during the execution of a program [15, Section 4.1]. Information about types provides clues for the compiler to make optimizations such as for performance, space (image size), safety or debuggability [10, Section 4.3] [15, Section 3.3]. Application programmers may as well make explicit use of types within their programs, such as with `typecase`, `typep`, `the` etc.

While the programmer can specify a homogeneous type for all the elements of a vector [15, Section 15.1.2.2], or the type for a particular element of a list, [15, System Class CONS], two notable limitations, which we address in this article, are 1) that there is no standard way to specify heterogeneous types for different elements of a vector, 2) neither is there a standard way to declare types (whether heterogeneous or homogeneous) for all the elements of a list.

We introduce the concept of *rational type expression* for abstractly describing patterns of types within sequences. The concept is envisioned to be intuitive to the programmer in that it is analogous to patterns described by regular expressions, which we assume the reader is already familiar with.

Just as the characters of a `string` may be described by a rational expression such as $(a \cdot b^* \cdot c)$, which intends to match strings such as `"ac"`, `"abc"`, and `"abbbbc"`, the rational type expression $(\mathit{string} \cdot \mathit{number}^* \cdot \mathit{symbol})$ is intended to match vectors like `#("hello" 1 2 3 world)` and lists like `("hello" world)`. Rational expressions match character constituents of strings according to character equality. Rational type expressions match elements of sequences by element type.

We further introduce an s-expression based syntax, called *regular type expression* to encode a *rational type expression*. This syntax replaces the infix and post-fix operators in the rational type expression with prefix notation based s-expressions. The regular type expression `(:1 string (:* number) symbol)` corresponds to the rational type expression $(\mathit{string} \cdot \mathit{number}^* \cdot \mathit{symbol})$. In addition, we provide a parameterized type named **rte**, whose argument is a regular type expression. The members of such a type are all sequences matching the given regular type expression. Section 2 describes the syntax.

While we avoid making claims about the potential utility of declarations of such a type from the compiler's perspective [1], we do suggest that a declarative system to describe patterns of types within sequences has great utility for program logic, code readability, and type safety.

A discussion of the theory of rational languages in which our research is grounded, may be found in [9, Chapters 3,4]. This article avoids many details of the theory, and instead emphasizes examples of problems this approach helps to solve and explains a high level view of its implementation. A more in-depth report of the research including the source code is provided in [11].

2. THE REGULAR TYPE EXPRESSION

We have implemented a parameterized type named **rte** (regular type expression), via `deftype`. The argument of

9th ELS May 9–10, 2016, Kraków, Poland

`:*`	match zero or more times.
`:+`	match one or more times.
`:?`	match zero or one time.
`:1`	match exactly once.
`:or`	match any of the regular type expressions.
`:and`	match all of the regular type expressions.

Table 1: Regular type expression keywords

rte is a *regular type expression.*

A *regular type expression* is defined as either a Common Lisp type specifier, such as `number`, `(cons number)`, `(eql 12)`, or `(and integer (satisfies oddp))`, or a list whose first element is one of a limited set of keywords shown in Table 1, and whose trailing elements are other regular type expressions.

As a counter example, `(rte (:1 (number number)))` is invalid because `(number number)` is neither a valid Common Lisp type specifier, nor does it begin with a keyword from Table 1. Here are some valid examples.

`(rte (:1 number number number))`
corresponds to the rational type expression $(number \cdot number \cdot number)$ and matches a sequence of exactly three numbers.

`(rte (:or number (:1 number number number)))`
corresponds to $(number + (number \cdot number \cdot number))$ and matches a sequence of either one or three numbers.

`(rte (:1 number (:?  number number)))`
corresponds to $(number \cdot (number \cdot number)^{?})$ and matches a sequence of one mandatory number followed by exactly zero or two numbers. This happens to be equivalent to the previous example.

`(rte (:* cons number))`
corresponds to $(cons \cdot number)^{*}$ and matches a sequence of a `cons` followed by a `number` repeated zero or more times, *i.e.*, a sequence of even length.

The **rte** type can be used anywhere Common Lisp expects a type specifier as the following examples illustrate. The `point` slot of the class `C` expects a sequence of two numbers, *e.g.*, `(1 2.0)` or `#(1 2.0)`.

```
(defclass C ()
  ((point :type (and list (rte (:1 number number))))
   ...))
```

The Common Lisp type specified by `(cons number)` is the set of non-empty lists headed by a number, as specified in [15, System Class CONS]. The `Y` argument of the function `F` must be a possibly empty sequence of such objects, because it is declared as type `(rte (:* (cons number)))`. *E.g.*, `#((1.0) (2 :x) (0 :y "zero"))`.

```
(defun F (X Y)
  (declare
    (type (rte (:* (cons number)))
          Y))
  ...)
```

3. APPLICATION USE CASES

The following subsections illustrate some motivating examples of regular type expressions.

```
lambda-list :=
  (var*
   [&optional
     {var | (var [init-form [supplied-p-parameter]])}*]
   [&rest var]
   [&key {var | ({var | (keyword-name var)}
                 [init-form [supplied-p-parameter]]) }*
         [&allow-other-keys]]
   [&aux {var | (var [init-form])}*]
   )
```

Figure 1: CL specification syntax for the ordinary lambda list

3.1 RTE based string regular expressions

Since a string in Common Lisp is a sequence, the **rte** type may be used to perform simple string regular expression matching. Our tests have shown that the **rte** based string regular expression matching is about 35% faster than CL-PPCRE [16] when restricted to features strictly supported by the theory of rational languages, thus ignoring CL-PPCRE features such as character encoding, capture buffers, recursive patterns, etc.

The call to the function `remove-if-not`, below, filters a given list of strings, retaining only the ones that match an implicit regular expression `"a*Z*b*"`. The function, `regexp-to-rte` converts a string regular expression to a regular type expression.

```
(regexp-to-rte "(ab)*Z*(ab)*")
==>
  (:1 (:* (member #\a #\b))
      (:* (eql #\Z))
      (:* (member #\a #\b)))

(remove-if-not
  (lambda (str)
    (typep str
      `(rte ,(regexp-to-rte "(ab)*Z*(ab)*"))))
  '("baZab"
    "ZaZabZbb"
    "aaZbbbb"
    "aaZZZZbbbb"))
==>
("baZab"
 "aaZbbbb"
 "aaZZZZbbbb")
```

The `regexp-to-rte` function does not attempt the daunting task of fully implementing Perl compatible regular expressions as provided in CL-PPCRE. Instead `regexp-to-rte` implements a small but powerful subset of CL-PPCRE whose grammar is provided by [4]. Starting with this context free grammar, we use `CL-Yacc` [5] to parse a string regular expression and convert it to a regular type expression.

3.2 DWIM lambda lists

As a complex yet realistic example we use a regular type expression to test the validity of a Common Lisp lambda list, which are sequences which indeed are described by a pattern.

Common Lisp specifies several different kinds of lambda lists, used for different purposes in the language. *E.g..*, the *ordinary lambda list* is used to define lambda functions, the *macro lambda list* is for defining macros, and the *destructuring lambda list* is for use with `destructuring-bind`. Each of these lambda lists has its own syntax, the simplest of which is the *ordinary lambda list* (Figure 1). The following code shows examples of ordinary lambda lists which obey the specification but may not mean what you think.

```
(defun F1 (a b &key x &rest other-args)
  ...)

(defun F2 (a b &key ((Z U) nil u-used-p))
  ...)
```

The function `F1`, according to careful reading of the Common Lisp specification, is a function with three keyword arguments, `x`, `&rest`, and `other-args`, which can be referenced at the call site with a bizarre function calling syntax such as `(F1 1 2 :x 3 :&rest 4 :other-args 5)`. What the programmer probably meant was one keyword argument named `x` and an `&rest` argument named `other-args`. According to the Common Lisp specification [15, Section 3.4.1], in order for `&rest` to have its normal *rest-args* semantics in conjunction with `&key`, it must appear before not after the `&key` lambda list keyword. The specification makes no prevision for `&rest` following `&key` other than that one name a function parameter and the other have special semantics. This issue is subtle. In fact, SBCL considers this such a bizarre situation that it diverges from the specification and flags a `SB-INT:SIMPLE-PROGRAM-ERROR` during compilation: `misplaced &REST in lambda list: (A B &KEY X &REST OTHER-ARGS)`

The function `F2` is defined with an *unconventional* `&key` parameter which is not a symbol in the `keyword` package but rather in the current package. Thus the parameter `U` must be referenced at the call-site as `(F2 1 2 'Z 3)` rather than `(F2 1 2 :Z 3)`.

These situations are potentially confusing, so we define what we call the *dwim ordinary lambda list*. Figure 2 shows an implementation of the type `dwim-ordinary-lambda-list`. A Common Lisp programmer might want to use this type as part of a code-walker based checker. Elements of this type are lists which are indeed valid lambda lists for `defun`, even though the Common Lisp specification allows a more relaxed syntax.

The *dwim ordinary lambda list* differs from the *ordinary lambda list*, in the aspects described above and also it ignores semantics the particular lisp implement may have given to additional lambda list keywords. It only supports semantics for: `&optional`, `&rest`, `&key`, `&allow-other-keys`, and `&aux`.

3.3 destructuring-case

```
(defun F3 (obj)
  (typecase obj
    ((rte (:1 symbol (:+ (eql :count) integer)))
     (destructuring-bind (name &key (count 0)) obj
        ...))
    ((rte (:1 symbol list (:* string)))
     (destructuring-bind (name data
                          &rest strings) obj
        ...))))
```

Notice in the code above that each `rte` clause of the `typecase` includes a call to `destructuring-bind` which is related and hand coded for consistency. The function `F3` is implemented such that the object being destructured is certain to be of the format expected by the corresponding destructuring lambda list.

We provide a macro `destructuring-case` which combines the capability of `destructuring-bind` and `typecase`. Moreover, `destructuring-case` constructs the `rte` type specifiers in an intelligent way, taking into account not only the structure of the destructuring lambda list but also any given type declarations.

```
(deftype var ()
  `(and symbol
        (not (or keyword
                 (member t nil)
                 (member ,@lambda-list-keywords)))))

(deftype dwim-ordinary-lambda-list ()
  (let* ((optional-var
          '(:or var
                (:and list
                  (rte
                    (:1 var
                        (:? t
                            (:? var)))))))
         (optional
          `(:1 (eql &optional)
                (:* ,optional-var)))
         (rest '(:1 (eql &rest) var))
         (key-var
          '(:or var
                (:and list
                   (rte (:1
                          (:or var
                             (cons keyword
                                (cons var
                                      null)))
                          (:? t
                              (:? var)))))))
         (key
          `(:1 (eql &key)
               (:* ,key-var)
               (:?
                 (eql &allow-other-keys))))
         (aux-var
           '(:or var
                 (:and list
                    (rte (:1 var (:? t))))))
         (aux `(:1 (eql &aux)
                   (:* ,aux-var))))
    `(rte
      (:1 (:* var)
          (:? ,optional)
          (:? ,rest)
          (:? ,key)
          (:? ,aux)))))
```

Figure 2: The `dwim-ordinary-lambda-list` type

```
(defun F4 (obj)
  (destructuring-case obj
    ((name &key count)
     (declare (type symbol name)
              (type integer count))
     ...)
    ((name data &rest strings)
     (declare (type name symbol)
              (type data list)
              (type strings
                    (rte (:* string))))
     ...)))
```

This macro is able to parse any valid destructuring lambda list and convert it to a regular type expression. Supported syntax includes `&whole`, `&optional`, `&key`, `&allow-other-keys`, `&aux`, and recursive lambda lists such as:

```
(&whole llist a (b c)
 &key x ((:y (c d)) '(1 2))
 &allow-other-keys)
```

A feature of `destructuring-case` is that it can construct regular type expressions much more complicated than would be practical by hand. Consider the following example which includes two destructuring lambda lists, whose computed regular type expressions pretty-print to about 20 lines each.

```
(:1 (:1 fixnum (:and list (rte (:1 fixnum fixnum))))
    (:and
      (:* keyword t)
      (:or
        (:1 (:? (eql :x) symbol (:* (not (member :y :z)) t))
            (:? (eql :y) string (:* (not (eql    :z))    t))
            (:? (eql :z) list   (:* t t)))
        (:1 (:? (eql :y) string (:* (not (member :x :z)) t))
            (:? (eql :x) symbol (:* (not (eql    :z))    t))
            (:? (eql :z) list   (:* t t)))
        (:1 (:? (eql :x) symbol (:* (not (member :y :z)) t))
            (:? (eql :z) list   (:* (not (eql    :y))    t))
            (:? (eql :y) string (:* t t)))
        (:1 (:? (eql :z) list   (:* (not (member :x :y)) t))
            (:? (eql :x) symbol (:* (not (eql    :y))    t))
            (:? (eql :y) string (:* t t)))
        (:1 (:? (eql :y) string (:* (not (member :x :z)) t))
            (:? (eql :z) list   (:* (not (eql    :x))    t))
            (:? (eql :x) symbol (:* t t)))
        (:1 (:? (eql :z) list   (:* (not (member :x :y)) t))
            (:? (eql :y) string (:* (not (eql    :x))    t))
            (:? (eql :x) symbol (:* t t))))))
```

Figure 3: Regular type expression matching destructuring lambda list Case-1

An example of the regular type expression matching `Case-1` is shown in Figure 3.

```
(destructuring-case data

  ;; Case-1
  ((&whole llist
     a (b c)
     &rest keys
     &key x y z
     &allow-other-keys)
   (declare (type fixnum a b c)
            (type symbol x)
            (type string y)
            (type list z))
   ...)

  ;; Case-2
  ((a (b c)
   &rest keys
   &key x y z)
   (declare (type fixnum a b c)
            (type symbol x)
            (type string y)
            (type list z))
   ...))
```

4. IMPLEMENTATION OVERVIEW

Using an **rte** involves several steps. The following subsections describe these steps.

1. Convert a parameterized **rte** type into code that will perform run-time type checking.

2. Convert the regular type expression to DFA (deterministic finite automaton, sometimes called a *finite state machine*).

3. Decompose a list of type specifiers into disjoint types.

4. Convert the DFA into code which will perform run-time execution of the DFA.

4.1 Type definition

The **rte** type is defined by `deftype`.

```
(deftype rte (pattern)
  `(and sequence
        (satisfies ,(compute-match-function
                       pattern))))
```

As in this definition, when the `satisfies` type specifier is used, it must be given a `symbol` naming a globally callable unary function. In our case `compute-match-function` accepts a regular type expression, such as `(:1 number (:* string))`, and computes a named unary predicate. The predicate can thereafter be called with a sequence and will return `true` or `false` indicating whether the sequence matches the pattern. Notice that the pattern is usually provided at *compile*-time, while the sequence is provided at *run*-time. Furthermore, `compute-match-function` ensures that given two patterns which are `EQUAL`, the same function name will be returned, but will only be created and compiled once. An example will make it clearer.

```
(deftype 3-d-point ()
  `(rte (:1 number number number)))
```

The type `3-d-point` invokes the **rte** parameterized type definition with argument `(:1 number number number)`. The `deftype` of **rte** assures that a function is defined as follows. The function name, `|(:1 number number number)|` even if somewhat unusual, is so chosen to improve the error message and back-trace that occurs in some situations.

```
(defun rte::|(:1 number number number)|
    (input-sequence)
  (match-sequence input-sequence
             '(:1 number number number)))
```

The following back-trace occurs when attempting to evaluate a failing assertion.

```
CL-USER> (the 3-d-point (list 1 2))

The value (1 2)
is not of type
  (OR (AND #1=(SATISFIES |(:1 NUMBER NUMBER NUMBER)|)
           CONS)
      (AND #1# NULL) (AND #1# VECTOR)
      (AND #1# SB-KERNEL:EXTENDED-SEQUENCE)).
   [Condition of type TYPE-ERROR]
```

It is also assured that the DFA corresponding to `(:1 number number number)` is built and cached, to avoid unnecessary re-creation at run-time. Finally, the type specifier `(rte (:1 number number number))` expands to the following.

```
(and sequence
     (satisfies |(:1 number number number)|))
```

A caveat of using **rte** is that the usage must obey a restriction posed by the Common Lisp specification [15, Section DEFTYPE]. A self-referential type definition is not valid. Common Lisp specification states: *Recursive expansion of the type specifier returned as the expansion must terminate, including the expansion of type specifiers which are nested within the expansion.*

As an example of this limitation, here is a failed attempt to implement a type which matches a unary tree, *i.e.* a type whose elements are `1`, `(1)`, `((1))`, `(((1)))`, etc.

```
CL-USER> (deftype unary-tree ()
           `(or (eql 1)
                (rte unary-tree)))
UNARY-TREE
RTE> (typep '(1) 'unary-tree)
Control stack exhausted (no more space for function call
frames).  This is probably due to heavily nested or
```

$$
\begin{aligned}
\partial_a \emptyset &= \emptyset \\
\partial_a \varepsilon &= \emptyset \\
\partial_a a &= \varepsilon \\
\partial_a b &= \emptyset \text{ for } b \neq a \\
\partial_a (r \cup s) &= \partial_a r \ \cup \ \partial_a s \\
\partial_a (r \cdot s) &= \begin{cases} (\partial_a r) \cdot s, & \text{if } r \text{ is not nullable} \\ (\partial_a r) \cdot s \ \cup \ \partial_a s, & \text{if } r \text{ is nullable} \end{cases} \\
\partial_a (r \cap s) &= \partial_a r \ \cap \ \partial_a s \\
\partial_a (r^*) &= (\partial_a r) \cdot r^* \\
\partial_a (r^+) &= (\partial_a r) \cdot r^*
\end{aligned}
$$

Figure 4: Rules for the Brzozowski derivative

```
infinitely recursive function calls, or a tail call that
SBCL cannot or has not optimized away.

PROCEED WITH CAUTION.
 [Condition of type SB-KERNEL::CONTROL-STACK-EXHAUSTED]
```

4.2 Constructing a DFA

In order to determine whether a given sequence matches a particular regular type expression, we conceptually execute a DFA with the sequence as input. Thus we must convert the regular type expression to a DFA. This need only be done once and can often be done at compile time.

4.2.1 Rational derivative

In 1964, Janusz Brzozowski [3] introduced the concept of the *Rational Language Derivative*, and provided a theory for converting a regular expression to a DFA. Additional work was done by Scott Owens *et al.* [12] which presented the algorithm in easy to follow steps.

To understand what the rational expression derivative is and how to calculate it, first think of a rational expression in terms of its language, *i.e.* the set of sequences the expression *generates*. For example, the language of $((a|b) \cdot c^* \cdot d)$ is the set of words (finite sequences of letters) which begin with exactly one letter a or exactly one letter b, end with exactly one letter d and between contain zero or more occurrences of the letter c.

The *derivative of the language* with respect to a given letter is the set of suffixes of words which have the given letter as prefix. Analogously the *derivative of the rational expression* is the rational expression which generates that language. *E.g.*, $\partial_a((a|b) \cdot c^* \cdot d) = (c^* \cdot d)$.

The Owens [12] paper explains a systematic algorithm for symbolically calculating such derivatives. The formulas listed in Figure 4 detail the calculations which must be recursively applied to calculate the derivative.

4.2.2 DFA for regular expressions

Another commonly used algorithm for constructing a DFA was inspired by Ken Thompson [18, 17] and involves decomposing a rational expression into a small number of cases such as base variable, concatenation, disjunction, and Kleene star, then following a graphical template substitution for each case. While this algorithm is easy to implement, it has a serious limitation. It is not able to easily express automata resulting from the intersection or complemention of rational expressions. We rejected this approach as we would like to support regular type expressions containing the keywords `:and` and `:not`, such as in `(:and (:* t integer) (:not (:* float t)))`.

We chose the algorithm based on Brzozowski derivatives.

Initial state Create the single initial state, and label it with the original rational expression. Seed a *to-do* list with this initial state. Seed a *visited* list to $\emptyset$.

States While the *to-do* list is non empty, operate on the first element as follows:

1. Move the state from the *to-do* list to the *visited* list.
2. Get the expression associated with the state.
3. Calculate the derivative of this expression with respect to each letter of the necessarily finite alphabet.
4. Reduce each derivative to a canonical form.
5. For each canonical form that does not correspond to a state in the *to-do* nor *visited* list, create a new state corresponding to this expression, and add it to the *to-do* list.

Transitions Construct transitions between states as follows: If S_1 is the expression associated with state P_1 and S_2 is the expression associated with state P_2 and $\partial_a S_1 = S_2$, then construct a transition from state P_1 to state P_2 labeled a.

Final states If the rational expression labeling a state is *nullable*, *i.e.* if it matches the empty word, label the state a final state.

Brzozowski argued that this procedure terminates because there is only a finite number of derivatives possible, modulo multiple equivalent algebraic forms. Eventually all the expressions encountered will be algebraically equivalent to the derivative of some other expression in the set.

4.2.3 DFA for regular type expressions

The set of sequences of Common Lisp objects is not a rational language, because for one reason, the perspective alphabet (the set of all possible Common Lisp objects) is not a finite set. Even though the set of sequences of objects is infinite, the set of sequences of type specifiers is a rational language, if we only consider as the alphabet, the set of type specifiers explicitly referenced in a regular type expression. With this choice of alphabet, sequences of Common Lisp type specifiers conform to the definition of *words* in a *rational language*.

There is a delicate matter when the mapping of sequence of objects to sequence of type specifiers: the mapping is not unique. This issue is ignored for the moment, but is discussed in Section 4.4.

Consider the extended rational type expression $P_0 = (\mathit{symbol} \cdot (\mathit{number}^+ \cup \mathit{string}^+))^+$. We wish to construct a DFA which recognizes sequences matching this pattern. Such a DFA is shown in Figure 5.

First, we create a state P_0 corresponding to the given rational type expression.

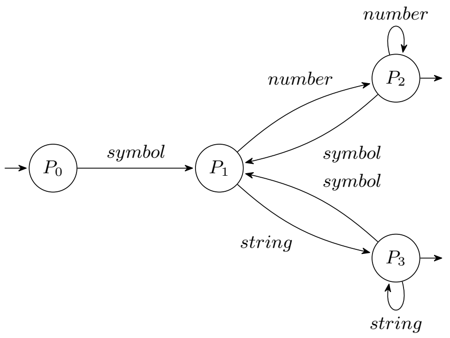


Figure 5: Example DFA

Next we proceed, to calculate the derivative with respect to each type specifier mentioned in P_0 and construct states P_1, P_2, and P_3 as those are the unique derivative forms which are obtained by the calculation. We discard the $\emptyset$ value.

$$\begin{aligned}
\partial_{number} P_0 &= \emptyset \\
\partial_{string} P_0 &= \emptyset \\
\partial_{symbol} P_0 &= (number^+ \cup string^+) \\
&\quad \cdot (symbol \cdot (number^+ \cup string^+))^* \\
&= P_1 \\
\partial_{number} P_1 &= number^* \\
&\quad \cdot (symbol \cdot (number^+ \cup string^+))^* \\
&= P_2 \\
\partial_{string} P_1 &= string^* \cdot (symbol \cdot (number^+ \cup string^+))^* \\
&= P_3 \\
\partial_{symbol} P_1 &= \emptyset \\
\partial_{number} P_2 &= P_2 \\
\partial_{string} P_2 &= \emptyset \\
\partial_{symbol} P_2 &= P_1 \\
\partial_{number} P_3 &= \emptyset \\
\partial_{string} P_3 &= P_3 \\
\partial_{symbol} P_3 &= P_1
\end{aligned}$$

Next, we label the transitions between states with the type specifier which was used in the derivative calculation between those states. We ignore transitions from any state to the $\emptyset$ state.

Finally, we label the *final* states. They are P_2 and P_3 because only those two states are nullable. *I.e.* $(number^* \cdot (symbol \cdot (number^+ \cup string^+))^*)$ can match the empty sequence, and so can $(string^* \cdot (symbol \cdot (number^+ \cup string^+))^*)$

4.3 Optimized code generation

The mechanism we chose for implementing the execution (as opposed to the generation) of the DFA was to generate specialized code based on `typecase`, `block`, and `go`. As an example, consider the DFA shown in Figure 5. The code in Figure 6 was generated given this DFA as input.

The code is organized according to a regular pattern. The `typecase`, commented as `OUTER-TYPECASE` switches on the type of the sequence itself. Whether the sequence, `seq`, matches one of the carefully ordered types `list`, `simple-vector`, `vector`, or `sequence`, determines which functions are used to access the successive elements of the sequence: `svref`, `incf`, `pop`, etc.

```
(lambda (seq)
  (declare
    (optimize (speed 3) (debug 0) (safety 0)))
  (block check
    (typecase seq  ; OUTER-TYPECASE
      (list
       (tagbody
        0
          (when (null seq)
            (return-from check nil)) ; rejecting
          (typecase (pop seq) ; INNER-TYPECASE
            (symbol (go 1))
            (t (return-from check nil)))
        1
          (when (null seq)
            (return-from check nil)) ; rejecting
          (typecase (pop seq) ; INNER-TYPECASE
            (number (go 2))
            (string (go 3))
            (t (return-from check nil)))
        2
          (when (null seq)
            (return-from check t)) ; accepting
          (typecase (pop seq) ; INNER-TYPECASE
            (number (go 2))
            (symbol (go 1))
            (t (return-from check nil)))
        3
          (when (null seq)
            (return-from check t)) ; accepting
          (typecase (pop seq) ; INNER-TYPECASE
            (string (go 3))
            (symbol (go 1))
            (t (return-from check nil)))))
      (simple-vector
       ...)
      (vector
       ...)
      (sequence
       ...)
      (t nil))))
```

Figure 6: Generated code recognizing an RTE

The final case, `sequence`, is especially useful for applications which wish to exploit the SBCL feature of *Extensible sequences* [10, Section 7.6] [13]. One of our **rte** based applications uses extensible sequences to view vertical and horizontal *slices* of 2D arrays as sequences in order to match certain patterns within row vectors and column vectors.

While the code is iterating through the sequence, if it encounters an unexpected end of sequence, or an unexpected type, the function returns `nil`. These cases are commented as `rejecting`. Otherwise, the function will eventually encounter the end of the sequence and return `t`. These cases are commented `accepting` in the figure.

Within the inner section of the code, there is one label per state in the state machine. In the example, the labels P_0, P_1, P_2, and P_3 are used, corresponding to the states in the DFA in Figure 5. At each step of the iteration, a check is made for end-of-sequence. Depending on the state either `t` or `nil` is returned depending on whether that state is a final

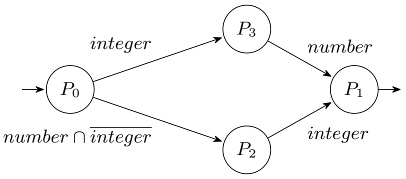


Figure 7: Example DFA with disjoint types

state of the DFA or not.

The next element of the sequence is examined by the INNER-TYPECASE, and depending of the type of the object encountered, control is transferred (via go) to a label corresponding to the next state.

One thing to note about the complexity of this function is that the number of states encountered when the function is applied to a sequence is equal or less than the number of elements in the sequence. Thus the time complexity is linear in the number of elements of the sequence and is independent of the number of states in the DFA.

In some cases the same type may be specified with either the **rte** syntax or with the Common Lisp native cons type specifier. For example, a list of three numbers can be expressed either as (cons number (cons number (cons number null))) or as (rte (:1 number number number)).

Should the **rte** system internally exploit the cons specifier when possible, thus avoiding the creation of finite state machines? We began investigating this possibility, but abandoned the investigation on discovering that it lead to significant performance degradation for long lists. We measured roughly a 5% penalty for lists of length 5. The penalty grew for longer lists: 25% with a list length of 10, 40% with a list length of 20.

4.4 The overlapping types problem

In the example in Section 4.2.3, all the types considered (symbol, string, and number) were disjoint. If the same method is naively used with types which are intersecting, the resulting DFA will not be a valid representation of the rational expression. Consider the rational expression involving the intersecting types *integer* and *number*: $P_0 = ((number \cdot integer) \cup (integer \cdot number))$. The sequences which match this expression are sequences of two numbers, at least one of which is an integer. Unfortunately, when we calculate $\partial_{number} P_0$ and $\partial_{integer} P_0$ we arrive at a different result.

$$\begin{aligned}
\partial_{number} P_0 &= \partial_{number}(\ (number \cdot integer) \\
&\qquad\qquad \cup\ (integer \cdot number)) \\
&= \partial_{number}(number \cdot integer) \\
&\quad\ \cup\ \partial_{number}(integer \cdot number) \\
&= (\partial_{number} number) \cdot integer \\
&\quad\ \cup\ (\partial_{number} integer) \cdot number \\
&= \varepsilon \cdot integer\ \cup\ \emptyset \cdot number \\
&= integer \cup \emptyset \\
&= integer
\end{aligned}$$

Without continuing to calculate all the derivatives, it is already clear that this result is wrong. If you start with the set of sequences of two numbers one of which is an integer, and out of that find the subset of sequences starting with a number, we get back the entire set. The set of suffixes of this set is **not** the set of singleton sequences of integer.

To address this problem, we augment the algorithm of Brzozowski with an additional step. Rather than calculating the derivative at each state with respect to each type mentioned in the regular type expression, some of which might be overlapping, instead we calculate a disjoint set of types. More specifically, given a set $\mathcal{A}$ of overlapping types, we calculate a set $\mathcal{B}$ which has the properties: Each element of $\mathcal{B}$ is a subtype of some element of $\mathcal{A}$, any two elements $\mathcal{B}$ are disjoint from each other, and $\cup\mathcal{A} = \cup\mathcal{B}$.

Figure 7 illustrates such a disjoint union. The set of overlapping types $\mathcal{A} = \{number, integer\}$ has been replaced with the set of disjoint types $\mathcal{B} = \{number \cap \overline{integer}, integer\}$.

This extra step has two positive effects on the algorithm. 1) it assures that the constructed automaton is deterministic, *i.e.*, we assure that all the transitions *leaving* a state specify disjoint types, and 2) it forces our treatment of the problem to comply with the assumptions required by the the Brzozowski/Owens algorithm.

The algorithm for decomposing a set of types into a set of disjoint types is an interesting research topic in its own right. While this topic is still under investigation, we have several algorithms which work very well for a small number of types (*i.e.* lists of up to 15 types). At the inescapable core of each algorithm is Common Lisp function subtypep [2]. This function is crucial not only in type specifier simplification, needed to test equivalence of symbolically calculated Brzozowski derivatives, but also in deciding whether two given types are disjoint. For example, we know that string and number are disjoint because (and string number) is a subtype of nil.

We explicitly omit further discussion of that algorithm in this article. We will consider it for future publication. For a complete exposition of our ongoing research into this topic, see the project report on the LRDE website [11].

5. RELATED WORK

Attempts to implement destructuring-case are numerous. We mention three here. R7RS Scheme provides case-lambda [14, Section 4.2.9] which appears to be syntactically similar construct, allowing argument lists of various fixed lengths. However, according to the specification nothing similar to Common Lisp style destructuring is allowed.

The implementation of destructuring-case provided in [6] does not have the feature of selecting the clause to be executed according to the format of the list being destructured. Rather it uses the first element of the given list as a case-like key. This key determines which pattern to use to destructure the remainder of the list.

The implementation provided in [7], named destructure-case, provides similar behavior to that which we have developed. It destructures the given list according to which of the given patterns matches the list. However, it does not handle destructuring within the optional and keyword arguments.

```
(destructuring−case '(3 :x (4 5))
  ((a &key ((:x (b c))))
   (list 0 a b c)) ;; this clause should be taken
  ((a &key x)
   (list 2 a x)))  ;; not this clause
```

In none of the above cases does the clause selection consider the types of the objects within the list being destructured. Clause selection also based on type of object is a distinguishing feature of the **rte** based implementation of destructuring-case.

The **rte** type along with destructuring-bind and type-case as mentioned in Section 3.3 enables something similar to pattern matching in the XDuce language [8]. The XDuce language allows the programmer to define a set of functions with various lambda lists, each of which serves as a pattern available to match particular target structure within an XML document. Which function gets executed depends on which lambda list matches the data found in the XML data structure.

XDuce introduces a concept called *regular expression types* which indeed seems very similar to *regular type expressions*. In [8] Hosoya *et al.* introduce a *semantic type* approach to describe a system which enables their compiler to guarantee that an XML document conform to the intended type. The paper deals heavily with assuring that the regular expression types are well defined when defined recursively, and that decisions about subtype relationships can be calculated and exploited.

A notable distinction of the **rte** implementation as opposed to the XDuce language is that our proposal illustrates adding such type checking ability to an existing type system and suggests that such extensions might be feasible in other existing dynamic or reflective languages.

The concept of regular trees, is more general that what **rte** supports, posing interesting questions regarding apparent shortcomings of our approach. The semantic typing concept described in [8] indeed seems to have many parallels with the Common Lisp type system in that types are defined by a set of objects, and sub-types correspond to subsets thereof. These parallels would suggest further research opportunities related to **rte** and Common Lisp. However, the limitation that **rte** cannot be used to express trees of arbitrary depth as discussed in Section 4.1 seems to be a significant limitation of the Common Lisp type system. Furthermore, the use of `satisfies` in the **rte** type definition, seriously limits the `subtypep` function's ability to reason about the type. Consequently, programs cannot always use `subtypep` to decide whether two **rte** types are disjoint or equivalent, or even if a particular **rte** type is empty. Neither can the compiler dependably use `subtypep` to make similar decisions to avoid redundant assertions in function declarations.

It is not clear whether Common Lisp could provide a way for a type definition in an application program to extend the behavior of `subtypep`. Having such a capability would allow such an extension for **rte**. Rational language theory does provide a well defined algorithm for deciding such questions given the relevant rational expressions [9, Sections 4.1.1, 4.2.1]. It seems from the specification that a Common Lisp implementation is forbidden from allowing self-referential types, even in cases where it would be possible to do so.

6. CONCLUSIONS

In this paper we presented a Common Lisp type definition, **rte**, which implements a declarative pattern based approach for declaring types of heterogeneous sequences illustrating it with several motivating examples. We further discussed the implementation of this type definition and its inspiration based in rational language theory. While the total computation needed for such type checking may be large, our approach allows most of the computation to be done at compile time, leaving only an $\mathcal{O}(n)$ complexity calculation remaining for run-time computation.

Our contributions are

1. recognizing the possibility to use principles from rational theory to address the problem dynamic type checking of sequences in Common Lisp,

2. adapting the Brzozowski derivative algorithm to sequences of lisp types by providing an algorithm to symbolically decompose a set of lisp types into an equivalent set of disjoint types,

3. implementing an efficient $\mathcal{O}(n)$ algorithm to pattern match an arbitrary lisp sequence, and

4. implementing concrete **rte** based algorithms for recognizing certain commonly occurring sequence patterns.

For future extensions to this research we would like to experiment with extending the `subtypep` implementation to allow application level extensions, and therewith examine run-time performance when using **rte** based declarations within function definitions.

Another topic we would like to research is whether the core of this algorithm can be implemented in other dynamic languages, and to understand more precisely which features such a language would need to have to support such implementation.

7. REFERENCES

[1] Declaring the elements of a list, discussion on comp.lang.lisp, 2015.

[2] H. G. Baker. A decision procedure for Common Lisp's SUBTYPEP predicate. *Lisp and Symbolic Computation*, 5(3):157–190, 1992.

[3] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[4] R. D. Cameron. Perl style regular expressions in Prolog, CMPT 384 lecture notes, 1999.

[5] J. Chroboczek. CL-Yacc, a LALR(1) parser generator for Common Lisp, 2009.

[6] P. Domain. Alexandria implementation of destructuring-case.

[7] N. Funato. Public domain implementation of destructuring-bind, 2013.

[8] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, Jan. 2005.

[9] J. D. U. Johh E. Hopcroft, Rajeev Motwani. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 2001.

[10] W. H. Newman. Steel Bank Common Lisp user manual, 2015.

[11] J. Newton. Report: Efficient dynamic type checking of heterogeneous sequences. Technical report, EPITA/LRDE, 2016.

[12] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, Mar. 2009.

[13] C. 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.

[14] A. Shinn, J. Cowan, and A. A. Gleckler. Revised 7 report on the algorithmic language scheme. Technical report, 2013.

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

[16] E. Weitz. *Common Lisp Recipes: A Problem-solution Approach.* Apress, 2015.

[17] G. Xing. Minimized Thompson NFA. *Int. J. Comput. Math.*, 81(9):1097–1106, 2004.

[18] F. Yvon and A. Demaille. *Théorie des Langages Rationnels.* 2014.

A CLOS Protocol for Editor Buffers

Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
robert.strandh@u-bordeaux.fr

ABSTRACT

Many applications and libraries contain a data structure for storing and editing text. Frequently, this data structure is chosen in a somewhat arbitrary way, without taking into account typical use cases and their consequence to performance. In this paper, we present a data structure in the form of a CLOS protocol that addresses these issues. In particular, the protocol is divided into an *edit* protocol and an *update* protocol, designed to be executed at different frequencies. The update protocol is based on the concept of *time stamps* allowing multiple *views* without any need for *observers* or similar techniques for informing the views of changes to the model (i.e., the text buffer).

In addition to the protocol definition, we also present two different implementations of the definition. The main implementation uses a splay tree of lines, where each line is represented either as an ordinary vector or as a gap buffer, depending on whether the line is being edited or not. The other implementation is very simple and supplied only for the purpose of testing the main implementation.

CCS Concepts

•Applied computing → Text editing;

Keywords

CLOS, Common Lisp, Text editor

1. INTRODUCTION

Many applications and libraries contain a data structure for storing and editing text. In a simple input editor, the content can be a single, relatively short, line of text, whereas in a complete text editor, texts with thousands of lines must be supported.

In terms of abstract data types, one can think of an editor buffer as an *editable sequence*. The problem of finding a good data structure for such a data type is made more interesting because a data structure with optimal asymptotic worst-case complexity would be considered as having too much overhead, both in terms of execution time, and in terms of memory requirements.

For a text editor with advanced features such as keyboard macros, it is crucial to distinguish between two different control loops:

- The innermost loop consists of inserting and deleting individual items[1] in the buffer, and of moving one or more *cursors* from one position to an adjacent position.
- The outer loop consists of updating the *views* into the buffer. Each view is typically an interval of less than a hundred lines of the buffer.

When the user inserts or deletes individual items, the inner loop performs a single iteration for each iteration of the outer loop, i.e., the views are updated for each elementary operation issued by the user.

When operations on multiple items are issued, such as the insertion or deletion of *regions* of text, the inner loop can be executed a large number of iterations for a single iteration of the outermost loop. While such multiple iterations could be avoided in the case of regions by providing operations on intervals of items, doing so does not solve the problem of *keyboard macros* where a large number of small editing operations can be issued for a single execution of a macro. Furthermore, to avoid large amounts of special-case code, it is preferable that operations on regions be possible to implement as repeated application of elementary editing operations.

Roughly speaking, we can say that each iteration of the outer loop is performed for each character typed by the user. Given the relatively modest typing speed of even a very fast typist, as long as an iteration can be accomplished in a few tens of milliseconds, performance will be acceptable. This is sufficient time to perform a large number of fairly sophisticated operations.

An iteration of the inner loop, on the other hand, must be several orders of magnitude faster than an iteration of the outer loop.

9th ELS May 9–10, 2016, Kraków, Poland

[1] In a typical editor buffer, the items it contains are individual characters. Since our protocols and our implementations are not restricted to characters, we refer to the objects contained in it as "items" rather than characters. An item is simply an object that occupies a single place in the editable sequence that the buffer defines.

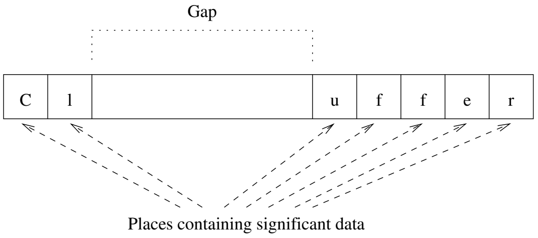


Figure 1: Gap buffer.

In this paper, we propose a data structure that has fairly low overhead, both in terms of execution time and in terms of storage requirements. More importantly, our data structure is defined as a collection of CLOS *protocols* each one aimed either at the inner or the outer control loop.

In Section 2, we provide an overview of existing representations of editor buffers, along with the characteristics of each representation. We give examples of existing editors with respect to which representation each one uses.

2. PREVIOUS WORK

2.1 Representing items in a buffer

There are two basic techniques for representing the items in an editor buffer, namely *gap buffer* and *line oriented.*

2.1.1 Gap buffer

A gap buffer can be thought of as a vector holding the items of the buffer but with some additional free space. In a typical gap-buffer implementation, a possibly empty *prefix* of the buffer content is stored at the beginning of the vector, and a possibly empty *suffix* of the content is stored at the end of the vector, leaving a possibly empty *gap* between the prefix and the suffix. This representation is illustrated in Figure 1.

Buffer items are moved from the end of the prefix to the beginning of the suffix, and vice-versa, in order to position the gap where an item is about to be inserted or deleted. The typical use case for text editing has a very high probability that two subsequent editing operations will be *close* to each other (in terms of the number of items between the two). Therefore, in most cases, few items will have to be moved, making this data structure very efficient for editing operations corresponding to this use case.

Clearly, in the worst case, all buffer items must be moved for every editing operation. This case happens when editing operations alternate between the beginning of the buffer and the end of the buffer. Even so, moving all the items even in a very large buffer does not represent a serious performance problem. For example, on a 64-bit 3MHz[2] Intel-based GNU/Linux system purchased in 2010 and runing SBCL, using the Common Lisp function `replace` to move 10^8 full words in a vector takes less than a second, and that number of words is several orders of magnitude larger than most files being edited. Furthermore, the pathological case can be largely avoided by considering the vector holding the items as being *circular* (as Flexichain [6] does).

The gap-buffer representation has one great advantage, namely that a large region of text can be treated as a single interval, even when the region covers several lines of text. Newlines are handled just like any other character, making operations on a region simple no matter how large that region is.

Perhaps the main disadvantage of representing the entire buffer as a single gap buffer is that it is difficult to associate additional information with specific points in the buffer. One might, for instance, want to associate some state of an *incremental parser* that keeps track of the buffer content in a more structured form. One possible solution to this problem is to introduce a *cursor*[3] at the points where it is desirable to attach information.

Another difficulty with the gap-buffer representation has to do with updating possibly multiple *views.* As we discussed in Section 1, views are updated at the frequency of the event loop, whereas the manipulation of *regions* of items and especially the use of *keyboard macros* may make the frequency of editing operations orders of magnitude higher.

Finally, using a gap buffer for the entire buffer makes it more difficult to handle multiple threads of control that simultaneously update different parts of the buffer, such as in a scenario with collaborative editing, or when the buffer is updated by a running program while a user is editing that same buffer.

2.1.2 Line oriented

Another common way of representing the editor buffer recognizes that text is usually divided into *lines*, where each line typically has a very moderate number of items in it.

In a line-oriented representation, we are dealing with a *two-level sequence.* At the outer level, we have a sequence of lines, and each element of that sequence is a sequence of items. Every possible combination of representations of these two sequences is possible. However, since the number of items in an individual line is usually small, most existing editors do not go to great lengths to optimize the representation of individual lines. Furthermore, while the number of lines in a buffer is typically significantly greater than the number of items in a line, a typical buffer may contain at most a few thousand lines, making the representation of the outer sequence fairly insignificant as well.

Perhaps the main disadvantage of a line-oriented representation compared to a gap-buffer representation is that transferring items to and from a file is slower. With a gap-buffer representation, the representation in memory and the representation in a file are very similar, making the transfer almost trivial. With a line-oriented representation, when a buffer is created from the content of a file, each line separator must be handled by the creation of a new representation of a line.

However, with modern processors, the time to load and store a buffer is likely to be dominated by the input/output operations. Furthermore, the number of lines in a typical buffer is usually very modest. For that reason, a line-oriented representation does not incur any serious performance penalty compared to a gap buffer.

[2] For this experiment, memory latency is of course much more important than CPU clock frequency. We intend this experiment to provide a rough idea of the expected performance, rather than exact number, which is why detailed information on the system is omitted.

[3] What we call a *cursor* in this paper is called a *point* in GNU Emacs terminology.

Another disadvantage with a line-oriented representation is that operations on regions that span over several lines become more difficult. For example, deleting such a region, may involve deleting one or more lines, and modifying the contents of the line where the region starts as well as the one where it ends.

2.2 Updating views

When interactive full-display text editors first started to appear, the main issue with updating a view was to minimize the number of bytes that had to be sent to a CRT terminal; this issue was due to the relative slowness of the communication line between the computer and the terminal. To accomplish this optimization, the *redisplay* function compared the previous view to the next one, and attempted to issue terminal-specific editing operations to turn the screen content into the updated version. Of course, most of the time, the task consisted of positioning the cursor and inserting a single character.

Today, there is no need to minimize the number of editing operations on a terminal; it is perfectly feasible to redraw the entire view for each iteration of the event loop. However, today we have many more requirements on a text editor. In the most advanced cases, we would like for an *incremental parser* in the view to keep a structured version of the buffer content, for various purposes, such as syntax highlighting, language-specific completion and parsing, etc. An incremental parser may require considerable computing power. It is therefore of utmost importance that as little work as possible is done each time around the event loop. Representing the entire editor buffer as a gap buffer does not lend itself to such advanced incremental processing.

In fact, most existing editors have very primitive parsers, mainly because the buffer representation does not necessarily lend itself to efficient incremental parsing.

2.3 Existing editors

2.3.1 GNU Emacs

GNU Emacs [1, 4] uses a *gap buffer* for the entire buffer of text, as described in Section 2.1.1.

Creating sophisticated parsers for the content of a buffer in GNU Emacs is not trivial. For that reason, existing parsers are typically fairly simple. For example, the parser for Common Lisp source code is unable to recognize the role of symbols in different contexts, such as the use of a Common Lisp symbol as a lexical variable. As a result, syntax highlighting can become confusing, and indentation is sometimes incorrect.

2.3.2 Multics Emacs

Multics Emacs[4] [3] was the first Emacs implementation written in Lisp, specifically, Multics MacLisp. It therefore pre-dates GNU Emacs.

Multics Emacs used a doubly linked list of lines, with the line content itself separate from the linked structure. All but a single line were said to be *closed*, and the content of a closed line was represented as a compact character string.

For the current line, a new MacLisp data type was added to the Multics MacLisp implementation, and it was called a *rplacable string*. Such a string could be seen as an ordinary MacLisp string, but could also have characters inserted or deleted through the use of primitives written in assembler and using special instructions on the GE 645 processor.

2.3.3 Climacs

Like GNU Emacs, Climacs[5] uses a gap buffer for the entire buffer. It avoids the bad case by using a circular buffer. In fact, it uses Flexichain [6].

Climacs is able to accommodate fairly sophisticated parsers for the buffer content. But in order to avoid a complete analysis of the entire buffer content for each view update, such parsers must be *incremental*.

Information about the state of such parsers at various positions in the buffer must be kept and compared between view updates. Unfortunately, the gap-buffer representation does not necessarily lend itself to storing such information. The workaround used in Climacs is to define a large number of *cursors* to hold parser state at various places in the buffer, but managing these cursors is a non-trivial task.

2.3.4 Others

Hemlock[6] uses a doubly linked list of lines. Each line is a `struct` containing a reference to the previous line and a reference to the next line. No more than one line is *open* at any point in time, and then the content is stored separately in a gap buffer. The gap-buffer data is contained in special variables and not encapsulated in a class or a `struct`.

Goatee[7] was written to be the input editor of McCLIM. Like Hemlock, it uses a doubly linked list of lines, with the difference that the line content itself is separate from the doubly linked structure. Lines are represented by a gap buffer. The gap buffer is encapsulated in a library called Flexivector, which was later extended to become the Flexichain library.

To take one example that is not a member of the Emacs family, VIM[8] represents the buffer contents as a tree of blocks. A block contains one or more lines of text. A line can not span block boundaries. Most blocks are exactly one page (as defined by the operating system) in size. Only a block containing a (single) line that is larger than one page can be larger than a page. Blocks are file backed, making it possible to represent text buffers that are larger than the swap space of the computer.

3. OUR TECHNIQUE

3.1 Protocols

Recall from Section 1 the existence of two nested control loops, the *inner* control loop in which each iteration is executing a single edit operation, and the *outer* control loop for the purpose of updating views.

The inner control loop is catered to by two different protocols; one containing operations on individual *lines* of items and one containing operations at the *buffer* level, concerning mainly the creation and deletion of lines. While we supply reasonable implementations of both these protocols, we also allow for sophisticated clients to substitute specific implementations of each one.

[4] The description in this section is a summary of the information found here: http://www.multicians.org/mepap.html

[5] https://common-lisp.net/project/climacs/

[6] https://www.cons.org/cmucl/hemlock/index.html

[7] Goatee is part McCLIM which can be found at this location: https://github.com/robert-strandh/McCLIM

[8] https://github.com/vim/

Although the protocol containing operations one individual lines and the protocol containing operations at the buffer level are independent, some high-level operations may involve both protocols simultaneously. For example, as mentioned in Section 2.1.2, deleting a region that spans several lines may require deleting several lines and editing the two lines at the beginning and the end of the region. Deleting a line is an operation that belongs to the buffer-level protocol, whereas editing a line is an operation that belongs to the line-level protocol.

The outer control loop is catered to by the *update protocol*. This protocol is based on the concept of *time stamps*. In order to request an update, client code supplies the time stamp of the previous similar request in addition to four different functions (`sync`, `skip`, `modify`, and `create`). These functions can be thought of as representing editing operations on the lines of the buffer. Our protocol implementation calls these functions in an order that will update the buffer content from its previous to its current state. The implementations of these functions are supplied by client code according to its own representation of the buffer content.

Presumably, client code maintains a sequence of lines and some kind of index into that sequence. The function `skip` is called with a positive integer argument, indicating that client code should increment the sequence index by that amount. The function `create` is called with a single line as an argument and it indicates that the line should be inserted at the place indicated by the sequence index. The function `modify` is also called with a single line as an argument. If the sequence index does not refer to the line of the argument, client code must firsts delete lines at the sequence index until the modified line becomes the one at the index. The `sync` function is called with a single unmodified line as an argument. It is called after a sequence of calls to the functions `modify` and `create`. Again, client code responds by deleting lines until the one passed as an argument become the one at the sequence index.

There are two main challenges that are addressed by the update protocol. The first one is that it has to be efficient when there is little or no modification to the buffer between two calls to the `update` function by client code. Our protocol handles this case by calling the `skip` function with a number of consecutive unmodified lines, as described above.

The second challenge has to do with deleted lines. The buffer does not hold on to deleted lines, but the protocol does not contain a function for deleting a line. Instead, it requires the contents of a line to be joined at the end of the preceding line. With this technique, a sequence of deleted lines leaves a trace in the form of an updated time stamp on the line immediately preceding the sequence of deleted ones. The `sync` operation allows for client code to be informed about the end of the sequence of deleted lines.

Figure 2 illustrates the relationship between these protocols.

The protocols illustrated in Figure 2 are related to one another by the *protocol classes* that they operate on. The buffer-edit protocol operates on instances of the protocol class named `buffer`. The line-edit protocol operates on instances of the two protocol classes `line` and `cursor`. These protocols are tied together by classes and other code, serving as *glue* to hold the two protocols together. Figure 3 illustrates the participation of these protocol classes in the different protocols, omitting the update protocol.

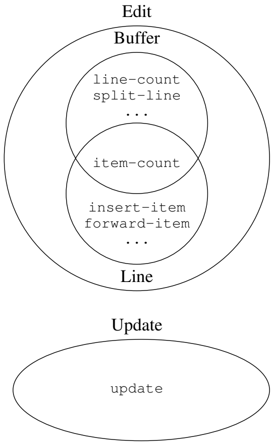


Figure 2: External protocols.

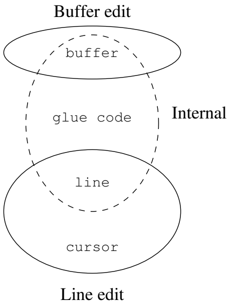


Figure 3: Participation of classes in protocols.

The internal protocol contains generic functions for which methods must be created that specialize to different *implementations* of the buffer-edit and the line-edit protocols. Client code using the library is not concerned with the existence of the internal protocol.

3.2 Supplied implementations

For the *line protocol*, we supply two different implementations, the *standard line* implementation and the *simple line* implementation. Similarly, for the *buffer protocol*, we supply two different implementations, the *standard buffer* implementation and the *simple buffer* implementation.

3.2.1 Simple line implementation

We supply an implementation, called the *simple line*, for the line editing protocol. The main purpose of this implementation is to serve as a reference for *random tests*. The idea here is that the implementation of the simple line is trivial, so that the correctness of the implementation is mostly obvious from inspecting the code, and in any case, it is unlikely that a defect in the simple line and another defect in the standard line will result in the same external behavior on a large body of randomly generated operations.

In addition to serving as a reference implementation for testing the standard line, this implementation can also serve as a reference for programmers who would like to create their own implementation of the line editing protocol.

The simple line implementation provides a single line abstraction, implemented as a Common Lisp simple vector. Each editing operation is implemented as reallocation of a new vector followed by calls to `replace` to copy items from the original line content to the one resulting from the editing operation. Clearly, this technique is very inefficient. For that reason, it is not recommended to use the simple implementation in client code.

3.2.2 Standard line implementation

The standard line implementation is the one that a typical application would always use, unless an application-specific line implementation is desired.

To appreciate the design of the standard line, we need to distinguish between two different *categories* of operations on a line. We call these categories *editing operations* and *content queries*, respectively. An editing operation is one in which the content of the line is modified in some way, and is the result of the interaction of a user typing text, inserting or removing a *region* of text, or executing a *keyboard macro* that results in one or more editing operations. A content query happens as a result of an *event loop* or a *command loop* updating one or more *views* of the content.

A crucial observation related to these categories is that content queries are the result of *events* (typically, the user typing text or executing commands). The frequency of such events is fairly low, giving us ample time to satisfy such a query. Editing operations, on the other hand, can be arbitrarily more frequent, simply because a single keystroke on the part of the user can trigger a very large number of editing operations.[9]

This implementation supplies two different representations of the line that we call *open* and *closed* respectively. A line is *open* if the last operation on it was an editing operation. It is *closed* if the last operation was a content query in the form of a call to the generic function `items`. Accordingly, a line is changed from being open to being closed whenever there is a content query, and from closed to open when there is a call to an editing operation.

A closed line is represented as a Common Lisp simple vector. An open line is represented as a gap buffer. (See Section 2.) The protocol specifically does not allow for the caller of a content query to modify the vector returned by the query. This restriction allows us to return the same vector each time there is a content query without any intervening editing operation, thus making it efficient for views to query closed lines repeatedly. Similarly, repeated editing operations maintain the line open, making such a sequence of operations efficient as well.

Clearly, the typical use case when a user issues keystrokes, each one resulting in a simple editing operation such as inserting or deleting an item, followed by an update of one or more views of the buffer content is not terribly efficient. The reason for this inefficiency is that this use case results in a line being alternately opened (as a result of the editing operation) and closed (as a result of the view update) for each keystroke. However, this use case does not have to be very efficient, again because the costly operations are invoked at the frequency of the event loop. The use case for which the standard line design was optimized is the one where a single keystroke results in several simple editing operations, i.e., the exact situation in which performance is crucial.

3.2.3 Simple buffer implementation

As with the implementations of the line-edit protocol, we supply an implementation, called the *simple buffer*, for the buffer editing protocol as well. Again, the main purpose of this implementation is to serve as a reference for *random tests*. As with the simple line implementation, the implementation of the simple buffer is trivial, so that the correctness of the implementation is mostly obvious from inspecting the code.

The simple buffer implementation represents the buffer as a Common Lisp vector of nodes, where each node contains a line and time stamps indicating when a line was created and last modified.

3.2.4 Standard buffer implementation

The main performance challenge for the buffer implementation is to obtain acceptable performance in the presence of multiple views (into a single buffer) that are far apart, and that both issue editing operations in each interaction. The typical scenario would be a user having two views, one close to the beginning of the buffer and one close to the end of the buffer, while executing a keyboard macro that deletes from one of the views and inserts into the other.

This time, the performance challenge has to do with the *update protocol* rather than with the edit protocols. A naive buffer implementation would have to iterate over all the lines each time the update protocol is invoked.

To obtain reasonable performance in the presence of mul-

[9] It is of course possible to supply *aggregate* operations that alleviate the problem of frequent editing operations. In particular, it is possible to supply operations that insert a *sequence* of items, and that delete a *region* of items. However, such operations complicate the implementations of the protocol. Worse, there are still cases where many simple editing operations need to be executed, in particular as a result of executing keyboard macros.

tiple views, the standard buffer implementation uses a *splay tree* [5] with a node for each line in the buffer. A splay tree is a *self-adjusting* binary tree, in that nodes that are frequently used migrate close to the root of the tree. Although the typical use of splay trees and other tree types is to serve as implementations of *dictionaries*, an often overlooked fact is that all trees can be used to implement *editable sequences*; that is how we use the splay tree here.

In addition to containing a reference to the associated line, each node in the splay tree contains time stamps corresponding to when the line was created and last modified. In addition, each node also contains summary information for the entire subtree rooted at this node. This summary information is what allows us to skip entire subtrees when a view requests update information and no node in the subtree has been modified since the last update request.

Finally, each node contains both a line count and an item count for the entire subtree, so that the offset of a particular line or a particular item can be computed efficiently, at least for nodes that are close to the root of the tree.

4. BENEFITS OF OUR TECHNIQUE

There are several advantages to our technique compared to other existing solutions.

First, most techniques expose a more concrete representation of the buffer to client code, such as a doubly linked list of lines. Our technique is defined in terms of an abstract CLOS protocol that can have several potential implementations.

Furthermore, our *update protocol* based on time stamps provides an elegant solution to the problem of updating multiple views at different times and with different frequencies. As opposed to the technique of using *observers* preferred in the object-oriented literature [2], time stamps require no communication from the model to the views as a result of modifications; indeed, such communication would be undesirable because of the high frequency of modifications to the model compared to the frequency of view updates. Instead, view updates are at the initiative of the views that need updating, and only when they need to be updated. The standard buffer implementation provided by our library provides an efficient implementation of the update protocol.

Because of the way the line-editing protocol is designed, a line can contain any Common Lisp object, and therefore any characters. The standard implementation of the line editing protocol uses simple vectors[10] to store the data. But the standard implementation in no way restricts the contents of a line to be characters. Client code can store any object in a line that it is prepared to receive when the contents of a line is asked for.

Our technique can be *customized* by the fact that the buffer editing protocol and the line-editing protocol are independent. Client code with specific needs can therefore replace the implementation of one or the other or both according to its requirements. Thanks to the existence of the CLOS protocol, such customization can be done gradually, starting with the supplied implementations and replacing them as requirements change. A typical customization might be to optimize the implementation of the line-editing protocol so that when every object in a particular line is an ASCII character, an efficient string representation is used instead of a simple vector.

The standard line implementation supplied makes it possible to obtain reasonable performance for aggregate editing operations even when these operations are implemented as iterative calls to elementary editing operations. This quality makes it possible for client code to be simpler, for obvious benefits.

A reasonable question that one might ask is whether the additional abstraction layer in the form of generic functions will have a negative impact on overall performance of an editor that uses our technique, especially since the protocols allow for the buffer to contain not only characters, but arbitrary objects. One important reason for designing these protocols was that we are convinced that the performance bottleneck is not in the communication between the buffer and the views, in particular given the design of the update protocol. Instead, the main performance challenge lies in how the views organize the data supplied by our protocols and how they interpret and display the data, and such design decisions are independent of the buffer protocols.

Finally, our technique is not specific to the abstractions of any particular existing editor, making our library useful in a variety of potential clients. In fact, we are already aware of one project for using our library in order to create a VIM-like editor.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have defined a CLOS protocol for manipulating text editor buffers. The protocol is divided into several sub-protocols, so that sophisticated clients can provide specific implementations according to the requirements of the application.

The *update* sub-protocol was designed to be used by an arbitrary number of *views*. The implementation we supply is fast so that this protocol can be invoked for events such as keystrokes, exposures, and changes of window geometry.

The features provided by CLOS are invaluable for the design of protocols like these. These features allow for partial or total customization of a large number of the details of the protocols while still providing reasonable default design choices that are workable for the majority of clients.

Our technique will have very little influence on the harder parts of editor design, such as managing variable-size fonts and the mixture of text and other objects to be displayed. The objective of our technique is more modest, namely to eliminate the necessity for creators of editors to make decisions about the organization of the buffer, and as a consequence also diminish the maintenance requirements on the code for buffer management.

In the remainder of this section, we outline future plans for the library.

5.1 Layer for Emacs compatibility

We plan to define a protocol layer on top of the edit protocols with operations that have the same semantics as the buffer protocol of GNU Emacs. Mainly, this work involves hiding the existence of individual lines, and treating the separation between lines as if it contained a *newline* character.

When a cursor is moved forward beyond the end of a line, or backward beyond the beginning of a line, this compatibility layer will have to detach the cursor from the line that it is currently attached to and re-attach it to the following

[10] A simple vector is a Common Lisp one-dimensional array, capable of storing any Common Lisp object.

or preceding line as appropriate.

Other minor operations need to be adapted, such as computing the item count of the entire buffer. This calculation will have to consider the separation of each pair of lines to contribute another item to the item count of the buffer.

5.2 Incremental Common Lisp parser

One of the essential reasons for the present work is to serve as an intermediate step towards the creation of a fully featured editor for Common Lisp code, entirely written in Common Lisp. Such an editor must be able to analyze the buffer content at least at the same frequency as that of the update of a view.

To that end, we plan to create a framework that allows the incremental parsing of the buffer content as Common Lisp code. Such a framework should allow for features such as syntax highlighting and automatic code indentation. Preferably, it should have a fairly accurate representation of the code such that the role of various code fragments can be determined. For example, it would be preferable to distinguish between a symbol in the `common-lisp` package when it is used to name a Common Lisp function and when (as the Common Lisp standard allows) it is used as a lexical variable with no relation to the standard function.

The first step of this incremental parser framework will be to adapt an implementation of the Common Lisp `read` function so that it can be used for incremental parsing, and so that the interpretation of *tokens* can take into account the specific situation of an editor buffer.

To take into account the different roles of symbols, the framework needs to include a *code walker* so that the occurrence of macro calls will not hamper the analysis.

5.3 Thread safety

The current implementation assumes that access is single-threaded. We plan to make multi-threaded access possible and safe. Implementing thread safety is not particularly difficult in itself. The interesting part would be to determine whether it is possible to achieve multi-threaded access without using a global lock for the entire buffer for every elementary operation. Clearly some high-level operations such as deleting a region that spans several lines may require a global lock, since such an operation involves several elementary operations involving both the buffer-level and the line-level editing protocols.

Since each line is a separate object, it would appear that locking a single line would be sufficient for most operations such as inserting or deleting a single item. However, the current implementation also keeps the *item count* of the entire buffer up to date for each such operation.

Fortunately, the item count for the entire buffer is typically asked for only at the frequency of the update protocol, for instance, in order to display this information to the end user. Other situations exist when this information is needed, for example when an operation to go to a particular item offset is issued. But such operations are relatively rare.

This analysis suggests that it may be possible to update the global item count lazily. Each line would be allowed to have a different item count from what is currently stored in the buffer data structure, and the buffer itself would maintain a set of lines with modified item counts. When the global item count of the buffer is needed, this set is first processed so that the global item count is up to date.

6. ACKNOWLEDGMENTS

We would like to thank Daniel Kochmański, Bart Botta, and Matthew Alan Martin for providing valuable feedback on early versions of this paper.

APPENDIX

A. PROTOCOL

In this section, we describe the protocols that are implemented by our library.

For each class, generic function, and condition type, we include only a brief description. In particular, we do not include a complete description of the exceptional situations possible. For a complete description, see the `Documentation` subdirectory in the repository at GitHub.[11]

A.1 Classes

`buffer` [*Protocol Class*]

This class is the base class of all buffers. Each different buffer implementation defines specific implementation classes to be instantiated by client code.

`line` [*Protocol Class*]

This class is the base class of all lines. Each different line implementation defines specific implementation classes to be instantiated by client code.

`cursor` [*Protocol Class*]

This class is the base class of all cursors. Each different line implementation defines specific implementation classes to be instantiated by client code.

A.2 Generic functions

`item-count` *entity* [*GF*]

If *entity* is a line, then return the number of items in that line. If *entity* is a cursor, return the number of items in the line in which *cursor* is located. If *entity* is a buffer, then return the number of items in the buffer.

`item-at-position` *line position* [*GF*]

Return the item located at *position* in *line.*

`insert-item-at-position` *line item position* [*GF*]

Insert *item* into *line* at *position.*

After this operation completes, what happens to cursors located at *position* before the operation depends on the class of the cursor and of *line.*

`delete-item-at-position` *line position* [*GF*]

Delete the item at *position* in *line.*

`cursor-position` *cursor* [*GF*]

Return the position of *cursor* in the line to which it is attached.

`(setf cursor-position)` *new-position cursor* [*GF*]

Set the position of *cursor* to *new-position* in the line to which *cursor* is attached.

`insert-item` *cursor item* [*GF*]

Calling this function is equivalent to calling `insert-item-at-position` with the line to which *cursor* is attached, *item,* and the position of *cursor.*

`delete-item` *cursor* [*GF*]

Delete the item immediately after *cursor.*

Calling this function is equivalent to calling `delete-item-at-position` with the line to which *cursor* is attached and the position of *cursor.*

[11] `https://github.com/robert-strandh/Cluffer`

`erase-item` *cursor* [*GF*]
Delete the item immediately before *cursor*.

Calling this function is equivalent to calling `delete-item-at-position` with the line to which *cursor* is attached and the position of *cursor* minus one.

`cursor-attached-p` *cursor* [*GF*]
Return *true* if and only if *cursor* is currently attached to some line.

`detach-cursor` *cursor* [*GF*]
Detach *cursor* from the line to which it is attached.

`attach-cursor` *cursor line* `&optional` *(position 0)* [*GF*]
Attach *cursor* to *line* at *position.*

`beginning-of-line-p` *cursor* [*GF*]
Return *true* if and only if *cursor* is located at the beginning of the line to which *cursor* is attached.

`end-of-line-p` *cursor* [*GF*]
Return *true* if and only if *cursor* is located at the end of the line to which *cursor* is attached.

`beginning-of-line` *cursor* [*GF*]
Position *cursor* at the very beginning of the line to which it is attached.

`end-of-line` *cursor* [*GF*]
Position *cursor* at the very end of the line to which it is attached.

`forward-item` *cursor* [*GF*]
Move *cursor* forward one position.

`backward-item` *cursor* [*GF*]
Move *cursor* backward one position.

`update` *buffer time sync skip modify create* [*GF*]
This generic function is the essence of the update protocol. The *time* argument is a time stamp that can be `nil` (meaning the creation time of the buffer) or a value returned by previous invocations of `update`. The arguments *sync*, *skip*, *modify*, and *create*, are functions. The *sync* function is called with the first unmodified line following a sequence of modified lines. The *skip* function is called with a number indicating the number of lines that have not been altered. The *modify* function is called with a line that has been modified. The *create* function is called with a line that has been created.

B. REFERENCES

[1] C. A. Finseth. *The Craft of Text Editing – Emacs for the Modern World.* Springer-Verlag, 1991.

[2] E. Gamma, J. Vlissides, R. Johnson, and R. Helm. *Design Patterns CD: Elements of Reusable Object-Oriented Software, (CD-ROM).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[3] B. S. Greenberg. Multics emacs (prose and cons): A commercial text-processing system in lisp. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 6–12, New York, NY, USA, 1980. ACM.

[4] B. Lewis, D. LaLiberte, and R. Stallman. *GNU Emacs Lisp Reference Manual.* Free Software Foundation, Boston, MA, USA, 2014.

[5] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[6] R. Strandh, M. Villeneuve, and T. Moore. Flexichain: An editable sequence and its gap-buffer implementation. In *Proceedings of the Lisp and Scheme Workshop*, 2004.

Session II: Domain Specific Languages

Using Lisp Macro-Facilities for Transferable Statistical Tests

Kay Hamacher
Dept. of Computer Science, Dept. of Physics, Dept. of Biology
Schnittspahnstr. 10
64287 Darmstadt, Germany
hamacher@bio.tu-darmstadt.de

ABSTRACT

Model-free statistical tests are purely data-driven approaches to assess correlations and other interdependencies between observable quantities. The few, distinct patterns how to perform these tests on the myriad of potentially different interdependence measures prompted us to use (Common) Lisp's macro capabilities for the development of a general, domain-specific language (DSL) of expectation values under so-called resampling techniques. Herein, we give an introduction into this statistical approach to big data, describe our solution, and report on application as well as on further research opportunities in statistical DSLs. We illustrate the results based on a toy example.

CCS Concepts

•**Computing methodologies → Symbolic and algebraic manipulation;** Shared memory algorithms; •**Software and its engineering → Domain specific languages;** •**Mathematics of computing →** *Nonparametric statistics; Statistical software;*

Keywords

Macros; Domain Specific Language; Statistical Modeling; Permutation Test

1. INTRODUCTION

Data analysis relies heavily on statistical tests. For example, in the traditional frequentist approach such procedures test, e.g., the significance under a null hypothesis (shortened to "the null"). In some cases, one cannot or does not want to formulate a (rather involved) null, but rather relies on a completely data-driven, model-free approach – so called parameter-free statistics[1]

Then the test is often carried out under so-called *resampling*. This avoids the need for knowledge of the underlying probability distribution of any test statistic. Resampling techniques can be distinguished based on the sample of the available data that they create and thus by the underlying question to be answered:

Jackknifing is capable of estimating variance and bias in a test statistic [15]. Here, we take the original data and delete d many entries from it, recompute the test statistic. Upon repetition we get a quantification of the sensitivity of the test statistic under finite-size effects. The parameter d characterizes the procedure.

Bootstraping creates an approximate distribution of the test statistic [6]. To this end, one draws random samples from the test data repeatedly with replacement.

Permutation Tests have a long tradition [5]. Here, one shuffles different data columns randomly in order to destroy potential correlations. This procedure is therefore able to assess correlation in the original data on the background of uncorrelated data via permutations/shuffles.

Typically, one is interested in the distribution of expectation values over a (resampled) data set (under any of the above procedures). The resampling procedure then computes a function – the test statistic – of the entries in the original data set first. Then, within a loop, we compute the same function over and over again over the resampled data set and obtain a collection of function values. This collection is what we are interested in: a computational approximation of the underlying, real distribution of function values. With respect to this distribution, we can then assess the "relevance"[2] of the function value for the original, un-sampled data set. Note, that the three methods above are the "basic" variants with a lot of variations in domain-specific applications; thus, a general framework to allow for easy implementation of variants is desirable – begging for a DSL.

1.1 Notation

In the subsequent part, we will deal with real-valued matrices $\mathcal{X}$ where the columns contain observations or measurements of different variables and each row $\vec{x_i}$ is one measurement of all the variables simultaneously. Note, that time

[1] In contrast to parameter statistics: here, an explicit model and its respective parameters are fitted to the data and the significance is judged on the fitting outcome.

[2] significance in statistical parlance

9th ELS May 9–10, 2016, Kraków, Poland

series analysis for, e.g., auto-correlation is possible by copying a time-lagged version from one column of the matrix into another.

A frequently employed notation for expectation values over a sample is

$$\left\langle f\left(\tilde{\mathcal{X}}\right)\right\rangle = \frac{1}{N_\sigma}\sum_{i=1}^{N_\sigma} f\left(\vec{\tilde{x}}_{\sigma(i)}\right) \tag{1}$$

The sampling function contained in the functions $\sigma(i) \to j$ and $\mathcal{X} \to \tilde{\mathcal{X}}$ implements the three procedures (permutation test, jackknife, bootstrap) from above. Some procedures – such as the jackknife – might change the number of data points from N to N_σ.

1.2 Related Work

First, there was a package for Lisp implementing general statistical methods called XLISP-STAT. This system was, however, abandoned by the proponents [4]. The main reason as discussed in the cited paper lies in the fact that a shift in the user community was recognized; eventually favoring R. Although we also employ R frequently, it has become apparent since the 90s that it is too much a compromise; on the surface you can be object-orientated, functional, even macros are (somewhat) possible. But in the end, neither of these traits is implemented thoroughly, not to speak about performance issues. Thus, the motivation [4] to abandon XLISP-STAT might have been in the light of the 90's a good one, our experience tells us otherwise in the meantime. Especially for prototyping computational procedures by (semi-)experts a new, more expressive and computational efficient technology needs to be employed.

Then, there exists some code [2] that implements statistical procedures and methods or interfaces to external systems like R [13]. Note, however, that this package implements concrete procedures, rather than a macro-based, general framework that automatically creates (Lisp-)code for *any conceivable* test statistic. Another attempt on parameter statistics [12] seems to have been abandoned.

Furthermore, several software packages outside the Lisp ecosystems are available, that follow the same, traditional, procedure-orientated lines: implementing resampling techniques but without any notion of domain-specific language that allows for the automatic generation of resampling for a novel or user-defined measure to work with. The most prominent ones are R [11], Julia [3] and (I)python [10]. While these languages are capable of self-introspection of code and Julia has a macro facility similar to Lisp-like languages they are all, however, *not homoiconic* to the extent of Common Lisp. Julia is the only language converging to the capabilities of Lisp. Thus, almost all these language lack the ability to mix code fragments of a domain specific language (DSL) for resampling as well as standard mathematical expressions of that language.

Work on DSLs in general has a long history [1] – eventually being strongly interwoven with the history of Lisp [7, 9] A full review is beyond the scope of the present work but it is fair to say, that quite a lot of work in the realm of software engineering has be done using DSLs. Most of this focus on one special question and thus is barely related to the concrete question on resampling procedures we address.

1.3 Our Contribution

We have implemented a general framework for resampling techniques. The data randomization and sampling procedure can be easily replaced by any function coherent to the interface of existing procedures. We will illustrate this in Sec. 2.1.

Our framework consists mainly of two macros that expand nested expectation values of expressions which implement any conceivable, real-valued function f in Eq. 1. By this, we can now easily write "statistical formulas" that contain results from resampling procedures.

We illustrate this by implementing and applying a traditional measure (covariance) to a multi-dimensional, dynamic system that produces synthetic data (cmp. Sec. 3.1) for test purposes, see Sec. 3.

2. LISP TO THE RESCUE: MACROS FOR PERMUTATION TESTS

Above, we have described how permutation tests show a repeated pattern: iterating several times over (partially) mixed choices from an underlying data set, each time recalculating a particular measure.

This pattern could potentially be applied to any code that implements such a measure f of Eq. 1 – as long as this pattern can access and introspect the code of the measure and "program the program code" to rewrite itself to implement this pattern again and again for each user-/programmer-supplied pattern – the realm in which Lisp excels due to this inherent macro capabilities. To this end, we define here a domain specific language that describes how a code fragment - namely a function uses data – and thus makes the measure accessible to a macro implementing any of the above discusses resampling approaches.

Formulas for a user-provided f in Eq. 1 need to refer to data elements contained in the (resampled) data set to perform their computations. We introduce a notation to this end that is motivated by the dataframe syntax of the statistical language R. In our DSL we make the data entry of any row i of the full data set $\tilde{\mathcal{X}}$ available as D$$i$ where i is a string or a number serving as a "name" for a particular column.

Thus, we are able to write a measure, e.g., $f = x_a \cdot x_b$ as a form (* D$a D$b). The macro needs then to expand this to a combination of columns a and b. Note, that in practice we are always interested in all pairings (a, b) of columns in the data set. Therefore, D$1 does *not* refer to the first column of the data, but rather represents the first column index currently under investigation.

Following Peter Seibel's suggestion [14] to first write down what a macro needs to achieve, we illustrate here how the resampling of a covariance $\langle (D\$a - \langle D\$a\rangle) \cdot (D\$b - \langle D\$b\rangle)\rangle$ between any pair of columns should be implemented:

```
(with-resampling test-dataframe 100
      #'permutation-test
      expectation-value
       (*
         (- D$a (expectation-value D$a))
         (- D$b (expectation-value D$b))))
```

Here test-dataframe is an array with test data for the model in Sec. 3.1. The function permutation-test is described in Sec. 2.1.

We achieve this by the macro with-resampling (shown in Algorithm 1) that first 1) extracts all D$x symbols, 2) sets up iterators over the combination of columns to be combined in the abstract syntax tree (AST) implementing the user's f, and 3) returns an array with the values of f for the original data and for the resampled ones.

While we could have implemented `with-resampling` as a function we took the deliberate design decision to implement it as a macro: we hope to accommodate future extension such as user-provided aggregation function (histogram building, for example) beyond the simple expectation value with this step and make the procedure more widely applicable.

Within the resampling procedure we need to parse the AST of f and insert appropriate code for expectation value computations.

To this end, we implement `parse-ev-calls` as a macro to make the current, looped-over `indices` of the sampled data set available to any formula internal to its respective macro invocation. We show in Algorithm 2 the concrete implementation for the expectation value. Thus a code fragment like above can be converted to an expression in which `D$`a and `D$`b are replaced by respective `aref`s to the resampled data.

Note, that we cannot naively use `subst`, `subst-if`, and similar facilities to walk the AST tree[3] : we must not substitute `D$`x symbols at different levels of nesting of `expectation-value` occurrences. One could achieve this by a rather involved predicate definition, but we decided to achieve the same result by an appropriate base case in the recursive definition of our `parse-ev-calls` macro.

2.1 Implementing Sampling Variants

We show in Algorithms 3, 4, 5, and 6 the implementations of the permutation test, the jackknife, and the bootstrap with a shared interface.

Furthermore, we have implemented the `identity-sample` function, which is a `permutation-test` without shuffling at all – thus we are able to compute the test statistic for the original data using `identity-sample`. Later on, we will illustrate the usefulness of this detail and the necessary redundancy in `identity-sample`.

Note, the subtle differences in the argument list between `identity-sample`, `bootstraping`, `identity-sample` on the one hand and `jackknifing` on the other: `jackknifing` needs an additional parameter d. We can, however, easily have a concrete jackknife for a fixed value of d conforming to the common parameter pattern by employing a lexical closure as in, e.g.,

```
(defun jack2 (idxs n)
  (let ((d 2))
    (jackknifing idxs n d)))
```

3. RESULTS

All tests were run on a machine under Linux Kernel version 4.3.3, SBCL 1.3.1.

3.1 Test Data

To apply our package we created synthetic data[4] for a system of three coupled dynamic variables with complex dynamics:

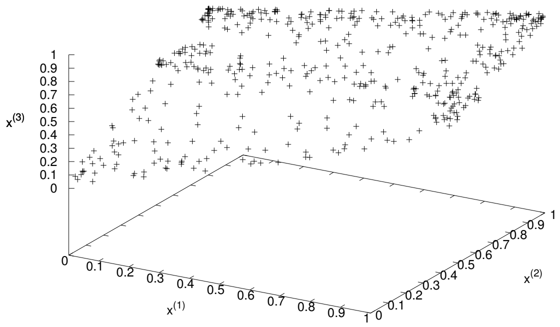


Figure 1: The 3D embedding of the time series $\left(x^{(1)}; x^{(2)}; x^{(3)}\right)$. Clearly each pair of variables covary, but also the 3-tupel of all three variables shows rich dynamics and interdependence.

$$f(x) = 4 \cdot x \cdot (1 - x) \tag{2}$$

$$\begin{aligned} a_{i+1} &= 0.5 \cdot f(a_i) + f(a_{i-3}) \\ x_{i+1}^{(1)} &= f\left(x_i^{(1)}\right) \\ x_{i+1}^{(2)} &= 0.8 \cdot f\left(x_i^{(2)}\right) + 0.2 \cdot f\left(x_i^{(1)}\right) \\ x_{i+1}^{(3)} &= 0.5 \cdot f\left(x_i^{(3)}\right) + 0.25 \cdot f\left(x_i^{(2)}\right) + 0.25 \cdot f\left(x_i^{(1)}\right) \end{aligned} \tag{3}$$

The function f is the logistic equation in the chaotic regime. Therefore, the series of $x^{(1)}$ and a values are independent, chaotic trajectories. At the same time, $x^{(2)}$ and $x^{(3)}$ are coupled instantaneously to the driving system $x^{(1)}$ and thus should show correlaction to $x^{(1)}$. We simulated the dynamics for $i \in [1 \ldots 500]$. A embedding plot of the trajectory for the three components $x^{(1)}$, $x^{(2)}$, $x^{(3)}$ is shown in Fig. 1. The trajectory of a serves as an example to which the other values $x^{(1)}$, $x^{(2)}$, $x^{(3)}$ cannot be correlated and thus any metric must vanish and/or be insignificant. Initial condition were chosen by a random generator.

3.2 An Example : Covariance

As an example we show the covariance between two data vectors $\vec{X} = (X_1, \ldots X_N)$ and $\vec{Y} = (Y_1, \ldots Y_N)$ defined as

$$\operatorname{covar}\left(\vec{X}, \vec{Y}\right) := \frac{1}{N} \sum_{i=1}^{N} \left(X_i - \bar{X}\right) \cdot \left(Y_i - \bar{Y}\right) \tag{4}$$

$$\text{with} \qquad \bar{X} = \sum_i X_i \quad \text{and} \quad \bar{Y} = \sum_i Y_i.$$

$\vec{X}$ and $\vec{Y}$ are any pairs of columns in the data set created in Sec. 3. We demanded above from our DSL this to be implementable with the code

```
(with-resampling test-dataframe 100
      #'permutation-test
```

[3] as, e.g., suggested at listips.com webpage

[4] Note, that for illustration purposes the used data set is irrelevant; we have, however, opted for a system under our complete control to be able to distinguish, e.g., spurious correlations from real ones ($a \leftrightarrow x^{(1)}$ and e.g. $x^{(2)} \leftrightarrow x^{(1)}$, respectively).

Algorithm 1 The macro implementing the “frame” for calling general AST representing a measure f. `multi-dim-homogenous-iter` is a CLOS-class that iterates over a regular, multi-dimensional grid of column indices (eventually the ones represented be the a, b, c, ... in the D$$x$ terminals present within the AST). This mapping of a, b, c, ... to real column numbers is done via a hash table that is modified by `modify-hash-table`. We omit several utility functions for brevity here, such as `repetition`, ...

```
(defmacro with-resampling(data N method &rest ast)
 `(let* ((arr-dims (array-dimensions ,data))
        (dimens (cadr arr-dims))
        (idxs (from-zero-to (car arr-dims)))
        (indices (identity-sample idxs dimens)))
   (multiple-value-bind (Ds data-hash-table) (extract-ds ',ast)
     (let* ((grid (repetition Ds dimens))
            (z (make-array (append grid (list (1+ ,N))) :initial-element 0.0)))
       (loop for nn from 0 to ,N do
            (let ((idx-iterator (make-instance 'multi-dim-homogenous-iter
                                               :dimensionality Ds :N dimens)))
              (loop while (not (donep idx-iterator)) do
                   (let ((idx (next idx-iterator)))
                     (setf data-hash-table
                           (modify-hash-table data-hash-table idx))
                     (let ((w (parse-ev-calls data-hash-table indices ,data  ,@ast)))
                       (setf (apply #'aref z
                                    (append (coerce idx 'list) (list nn)))
                             w)))))
            (setf indices (funcall ,method idxs dimens)))
       (values z)))))
```

Algorithm 2 A macro implementing the DSL-specific keyword `expectation-value`. Note, how we access the data set elements via an array of (sampled) indices and craft an S-expression in line numbers 7-9 to access those elements. We traverse the abstract syntax tree (AST) recursively.

```
1  (defmacro parse-ev-calls(Dsht indices data &body ast)
2    (labels ((walk (Dsht i d en runID trafo-ast)
3                (cond ((null en) trafo-ast)
4                      ((atom en) (if (and (symbolp en)
5                                          (start-with-D$p en))
6                                     (let ((IDX (gensym)))
7                                       (append `(let ((,IDX (gethash ',en ,Dsht)))
8                                                  (aref ,d (aref ,i ,runID ,IDX) ,IDX))
9                                               trafo-ast))
10                                    (if (eq en 'quote)
11                                        trafo-ast
12                                        (cons-non-nil en trafo-ast))))
13                     ((listp en)
14                      (if (and
15                           (symbolp (car en))
16                           (eq (car en) 'expectation-value) )
17                          (let* ((runID2 (gensym))
18                                 (cont (walk Dsht i d (cdr en) runID2 trafo-ast)))
19                            (concatenate 'list `(let((r 0.0)
20                                                     (II (car (array-dimensions ,indices))))
21                                                 (dotimes (,runID2 II) (incf r ,@cont))
22                                             (/ r II) ) trafo-ast))
23                          (mapcar #'(lambda(x)
24                                      (walk Dsht i d x runID nil)) en))))))
25     (let ((runID (gensym)))
26       (walk Dsht indices data ast runID nil))))
```

Algorithm 3 The identity mapping of indices – here, redundancy is necessary to easily implement the permutation test later on.

```
(defun identity-sample (idxs n)
  (let* ((rowMax (coerce (length idxs) 'number))
         (dims (list rowMax n))
         (current (coerce idxs 'vector))
         (z (make-array dims :initial-element 0)))
    (loop for j from 0 to (1- rowMax) do
         (loop for i from 0 to (1- n) do
              (setf (aref z j i) (aref current j))))
    (values z)))
```

Algorithm 4 Mapping of indices under the permutation test – each column is shuffled individually to destroy potential correlations. `nshuffle` implements Knuth's shuffling procedure.

```
(defun permutation-test (idxs n)
  (let* ((z (identity-sample idxs n)) ; initialize with identity
         (rowMax (coerce (length idxs) 'number))
         (current (coerce idxs 'vector)))
    (loop for i from 1 to (1- n) do ; shuffle all but the first column
         (setf current (nshuffle current))
         (loop for j from 0 to (1- rowMax) do
              (setf (aref z j i) (aref current j)))
         finally (return z))))
```

Algorithm 5 The bootstrap – indices are drawn randomly and might occur several times in the created index array.

```
(defun bootstraping (idxs n)
  (let* ((rowMax (coerce (length idxs) 'number))
         (dims (list rowMax n))
         (current  (coerce idxs 'vector))
         (z (make-array dims :initial-element 0))
         (w 0))
    (loop for j from 0 to (1- rowMax) do
         (setf w (random rowMax))
         (loop for i from 0 to (1- n) do
              (setf (aref z j i) (aref current w)))
       finally (return z))))
```

Algorithm 6 Jackknifing – d many, randomly chosen samples need to be omitted in this procedure.

```
(defun jackknifing (idxs n d)
  (let* ((rowMax (- (coerce (length idxs) 'number) d))
         (dims (list rowMax n))
         (current (nshuffle (coerce idxs 'vector)))
         (z (make-array dims :initial-element 0)))
    (loop for j from 0 to (1- rowMax) do
         (loop for i from 0 to (1- n) do
              (setf (aref z j i) (aref current j)))
       finally (return z))))
```

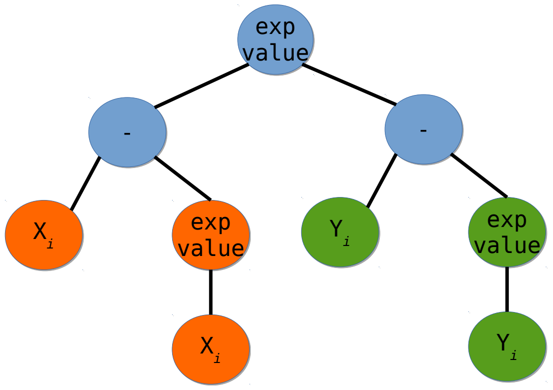


Figure 2: Tree-representation of the S-expression for the covariance in Eq. 4. The data vectors X and Y are used several times.

	a	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
a	0.0667	0.0660	0.0056	-0.0006
$x^{(1)}$	-0.0001	-0.0003	0.0009	0.0029
$x^{(2)}$	0.0056	-0.0005	0.1323	0.1323
$x^{(3)}$	0.0395	-0.0066	0.0462	0.0026

Table 1: Covariance values for the *original* data from Eq. 2. Note, that `parse-ev-calls` was expanded so that a and b in `D$`a and `D$`b took on all the combinations of columns, e.g., $\left(a = a, b = x^{(1)}\right)$; $\left(a = x^{(3)}, b = x^{(2)}\right)$; and so forth.

```
        expectation-value
          (*
            (- D$a (expectation-value D$a))
            (- D$b (expectation-value D$b))))
```

which is illustrated in Fig. 2. For brevity, we cannot show the full macro-expansion here, but publish it on the WWW[5]. The length and complexity clearly shows the benefits of a DSL. Furthermore, one can see in the macro-expanded code that no local variables other than the ones generated via `gensyms` exist, so no variable capture can occur.

When we apply our system to the four-dimensional test system from above we obtain the resulting covariance matrix between all variables in Tab. 1.

From the covariance alone we cannot judge on the influence of any variable onto the other. Eventually, we must find no connection between a and any of the xs as they are independent in Eq. 2.

Applying the permutation test with 100 repetitions we obtain the covariance values for 100 shuffled data sets. From this we can compute the (one-sided[6]) percentile as the fraction of covariance matrix entries that turned out to be smaller in the permutation test than for the original data. We obtain the output

	a	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
a	0.04	0.92	0.44	0.53
$x^{(1)}$	0.92	0.54	**1.00**	**1.00**
$x^{(2)}$	0.44	**1.00**	0.80	**1.00**
$x^{(3)}$	0.53	**1.00**	**1.00**	0.76

in which we marked the statistic insignificant results under a one-sided test in red. For significance we apply Fisher's well-known criterion of a so-called p-value smaller than 5% (or a percentile of larger than 0.95). Our permutation test based assessment shows, that we cannot find a statistically significant covariance between a and and any of the xs variables[7].

4. OUTLOOK

Above we have demonstrated the first development of a DSL for statistical resampling technqiues. We motivated our choice and described the used macros as well as their rationale. We applied it to a test problem and illustrated the necessity of such involved resampling techniques as otherwise one might attribute wrongly dependencies between variables obtained from stochastic processes.

Although our system is at present slower for simple measures like the covariance in comparison to manually tuned code like the ones we published for statistics using GPUs [16], we have with the present work laid ground for a *general* system for prototyping, interactive data science, and hypothesis generating. These steps are increasingly necessary in the realm of big data as their might be high-dimensional correlations present for which one cannot always hand-craft individual solutions. Rather, one can rely on Lisp's macro to do the job. At present, our contribution enables data scientists to implement general measures and their resampling tests easily and fast, with – at present – costs in performance.

As this approach is work in progress, several improvements will be implemented and researched in the future. In particular, we hope to encourage participation of the larger Lisp community on these issues:

Parallelization the resampling procedures laid out above are all data-parallel in the columns of a a dataframe. The iterator over those columns in Algo. 1 is thus "embarrassingly parallel" [8], begging for parallelization via, e.g., the `lparallel` library.

Sampling variants The implemented algorithms 4, 5, and 6 have a common pattern, but also some subtle differences. It seems to be promising find at an abstract level a macro to implement any conceivable resampling/reshuffling technique. Furthermore, in the literature all three resampling techniques can be found in variants; thus a DSL is no overkill, but rather a first step to offer a general framework.

User-provided aggregation mechanisms The expectation value as a sum over samples as in Eq. 1 is the

[5] http://www.kay-hamacher.de/macro-expanded.lisp

[6] a one-sided test tests for original data to be larger or smaller than the resampling ensemble; a two-sided test would test for the *absolute value* to be smaller.

[7] For the diagonal entries: as this a degenerated case, we compute the variance of a variable that does not change under the permutation test, thus we can – by construction of the test – not obtain any significant values.

predominant procedure in statistics. It aggregates the function values of a measure over a randomized sample into the arithmetic mean. This is, at present, hard-coded into the macro `parse-ev-calls`. Still, other aggregation procedures are also conceivable. We will extend the package to provide for any user-provided mechanism.

Auto-generation of measures On can use Lisp S-expressions and Lisp's ability to modify these to "evolve" ASTs implementing an appropriate f via, e.g., genetic programming.

Caching / Memoization The AST representing the measure f to be evaluated might be a rather complex, time-consuming function. Here, memoization techniques are one way to cope with this; this route will be taken in the future development of our system.

A first version of the package is made available on the web under http://www.kay-hamacher.de/Software/Resampling_Lisp.tar.gz.

5. ACKNOWLEDGMENTS

The author gratefully acknowledges financial support by the LOEWE projects iNAPO & compuGene of the Hessen State Ministry of Higher Education, Research and the Arts. This work was also supported by the German Federal Ministry of Education and Research (BMBF) within CRISP.

6. REFERENCES

[1] IEEE Transactions on Software Engineering (TSE), special issue, volume 25, number 3, may/june 1999.

[2] S. D. Anderson, A. Carlson, D. L. Westbrook, D. M. Hart, and P. R. cohen. Common lisp analytical statistics package: User manual. Technical report, Amherst, MA, USA, 1993.

[3] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. November 2014.

[4] J. de Leeuw. On abandoning XLISP-STAT. *Journal of Statistical Software*, 13(1):1–5, 2005.

[5] M. Dwass. Modified randomization tests for nonparametric hypotheses. *Ann. Math. Statist.*, 28(1):181–187, 03 1957.

[6] B. Efron. Bootstrap methods: Another look at the jackknife. *Ann. Statist.*, 7(1):1–26, 01 1979.

[7] P. Graham. *ANSI Common LISP*. Prentice Hall, Nov. 1995.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 1 edition, Mar. 2008.

[9] D. Hoyte. *Let Over Lambda*. 2008.

[10] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[11] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[12] T. Rossini. *Common Lisp Statistics*. https://github.com/blindglobe/common-lisp-stat, retrieved 03/15/2016.

[13] T. Rossini. *R-Common Lisp gateway*. https://github.com/blindglobe/rclg, retrieved 03/15/2016.

[14] P. Seibel. *Practical Common Lisp*. Apress, Sept. 2004.

[15] J. W. Tukey. Bias and confidence in not-quite large samples. *Ann. Math. Statist.*, 29(2):614–623, 06 1958. just the abstract to a talk.

[16] M. Waechter, K. Jaeger, D. Thuerck, S. Weissgraeber, S. Widmer, M. Goesele, and K. Hamacher. Using graphics processing units to investigate molecular coevolution. *Concurrency and Computation: Practice and Experience*, 26(6):1278–1296, 2014.

A High-Performance Image Processing DSL for Heterogeneous Architectures

Kai Selgrad* Alexander Lier* Jan Dörntlein* Oliver Reiche† Marc Stamminger*

*Computer Graphics Group †Hardware/Software Co-Design
Friedrich-Alexander University Erlangen-Nuremberg, Germany

{kai.selgrad, alexander.lier, jan.doerntlein, oliver.reiche, marc.stamminger}@fau.de

ABSTRACT

Over the last decade a number of high performance, domain-specific languages (DSLs) have started to grow and help tackle the problem of ever diversifying hard- and software employed in fields such as HPC (high performance computing), medical imaging, computer vision etc. Most of those approaches rely on frameworks such as LLVM for efficient code generation and, to reach a broader audience, take input in C-like form. In this paper we present a DSL for image processing that is on-par with competing methods, yet its design principles are in strong contrast to previous approaches. Our tool chain is much simpler, easing the burden on implementors and maintainers, while our output, C-family code, is both adaptable and shows high performance. We believe that our methodology provides a faster evaluation of language features and abstractions in the domains above.

CCS Concepts

•**Software and its engineering → Domain specific languages; Source code generation;** *Preprocessors; Macro languages;*

Keywords

Domain Specific Langauges, Generative Programming, Common Lisp, Meta Programming

1. INTRODUCTION

Image processing is a wide field with diverse applications ranging from high performance computing (e.g. fluid simulation) and computer vision to medical imaging and post-processing of computer generated imagery (in games and movies). For many of these applications efficient execution is crucial and the tools provided should be accessible to users that are not hardware experts. Therefore, approaches based on domain-specific languages are very popular in image processing (see Section 2), especially when the applications are intended to run on different target platforms, possibly at the same time.

To satisfy the performance demands of these applications, image processing DSLs usually generate output in the form of C or C++, or even lower level representations such as LLVM [10] bytecode. Targeting more closed systems even requires generating code in vendor-specific languages, such as CUDA [13], or the lower level PTX [14]. Additionally, many DSLs are embedded in low level host languages, such as C++, to exploit the already existing parser front-end and AST construction.

In this paper we propose to use a Lisp-based design approach to DSL construction in the high-performance domain and demonstrate our DSL, CHIPOTLE. CHIPOTLE is heavily inspired by HIPA^{cc}[1] [11], a CLANG-based, high-performance image processing DSL that targets heterogeneous applications. For instance, CHIPOTLE incorporates efficient execution-patterns for GPU kernels that were found to perform well in HIPA^{cc}.

In contrast to HIPA^{cc}, our DSL builds on C-MERA[2] [17], a lightweight S-Expression to C-style transcompiler embedded in COMMON LISP. The benefit of using C-MERA, especially as compared to CLANG, is that it carries much smaller overhead and is more easily extensible. This argument indicates that for CHIPOTLE it was a simple task to experiment with different notations, whereas HIPA^{cc} is strictly tied to valid C++ syntax. Providing more concise syntax would require changes to the C++ parser front-end, which is a highly non-trivial task. Using C-MERA's simple internal representation, extracting information from the input code is straightforward, as the input program itself is a (semantically annotated) syntax tree. With these two characteristics the task of expanding upon HIPA^{cc}'s feature-set (e.g. adding heterogeneous scheduling) is very problem-oriented and a correspondingly fast process. It should be noted, however, that CLANG is a very powerful tool and provides, amongst others, complete syntactic and semantic analysis and ensures type safety in the input program. While our goal is not to detract from such commodities, we rather propose that a more lightweight and flexible solution benefits research and exploration, and that using a full C++ compiler toolchain might be excessive for the rather limited code generation involved in our target domain. When fully type-checked and deeply implicit information is required, choosing CLANG in favour of C-MERA may prove reasonable.

9th ELS May 9–10, 2016, Kraków, Poland

[1] hipacc-lang.org

[2] github.com/kiselgra/c-mera

The remainder of this paper is structured as follows: In Section 2 we present an overview of the field of image processing DSLs and related approaches. Section 3 gives a short introduction to C-Mera, and our DSL is described in Section 4. In Section 5 we show two real-world examples of practical image processing operators, and compare our method's performance (in terms of executing time, application development time and DSL development effort) to a competing approach in Section 6. We conclude with a description of limitations and future work in Section 7.

The specific contributions of our work are:

- We present a new, high-performance image processing DSL targeting heterogeneous setups, Chipotle, that follows a Lisp-based methodology. The implementation is available and showcases an automatic generation of high-performance CUDA and AVX code from a single algorithm specification.
- We show how this language can be extended to automatically provide a schedule that employs GPUs and CPUs together, based on simple user input.
- We also show how easily these tasks are accomplished using powerful, but lightweight and flexible tools.
- Finally, we provide working examples that go beyond the simple filter setups commonly found in literature.

2. RELATED WORK

The feature set and optimization techniques of the presented Chipotle DSL are primarily inspired by HIPA^{cc} [11]. The HIPA^{cc} framework embodies a DSL for image processing, embedded into C++, and a source-to-source compiler, mainly focusing on point, local, and global operators. HIPA^{cc}'s compiler generates highly optimized code for different target architectures and languages, including CPUs, GPUs, and FPGAs through CUDA, OpenCL, Android's Renderscript, and Vivado HLS. HIPA^{cc} is based on the Clang/LLVM compiler infrastructure and performs AST-level optimizations based on domain and architecture knowledge. Architecture-specific optimizations include image padding for suitable memory alignment, the use of shared and texture memory, and thread-coarsening for GPUs. Furthermore, the automatic vectorization for common instruction sets of CPUs is supported, as well as the generation of a streaming pipeline for FPGAs. Optimizations based on domain knowledge, for instance, include the efficient handling of boundary conditions for local operators.

Halide [15] is also a DSL for image processing, based on the Clang/LLVM infrastructure, similar to HIPA^{cc}. Instead of imperatively describing image filters, a functional programming paradigm is applied, which enables additional sophisticated features, such as kernel fusion. Halide is capable of generating code for various target architectures, namely CUDA, OpenCL, PNaCL, as well as C++. Yet, this is not an entirely automatic process. The developer needs to specify a schedule that defines how the algorithm should be mapped onto the target architecture in order to obtain efficient code. Halide's schedule has to be manually specified and requires the developer to have a certain degree of architecture and domain knowledge. However, as the schedule is evaluated dynamically by the compiler, it can be altered or even entirely replaced at run time.

DeVito et al. [4] present Orion (and its source language, Terra), a stencil DSL for processing images, which is mainly inspired by Halide and makes use of mathematical operators that are implicitly evaluated on the whole image. This results in a very dense image processing pipeline representation. Although the DSL does not support the generation of code for different architectures, the vectorization module from its host language Terra can be used to map the stencil operations to vector instructions. Additionally, Orion adopted the ability to define a schedule for a filter pipeline from Halide, which led to very efficient results.

PolyMage [12] is a DSL for image processing where the image processing pipeline is represented as a directed, acyclic graph. Similar to our representation, this graph implicitly contains the data relationships between different operators and allows extracting this information to implement parallel scheduling for certain computations.

Patus [3] is a DSL and code generation framework for parallel stencil computations based on a C-like syntax. It follows a heterogeneous approach and is capable of generating code for CPU and GPU execution. Its auto-tuner is fed with a user-defined strategy to produce optimized code.

Native Common Lisp image processing libraries that target high-performance, such as Opticl[3], are, in contrast to the aforementioned methods, not DSLs themselves, but might be applicable as a basis for Lisp-only image processing DSL approaches. Targeting heterogeneous architectures might, however, prove problematic in such a setup.

A more general approach is proposed with frameworks for DSLs that can be used to create entirely new languages. Well-known representatives of this class are Delite [2], Asp [7], Terra [4] and AnyDSL [8]. Here, the framework performs generic, parallel and domain-specific optimizations for a new DSL without the necessity of starting development from scratch. Thereby, the effort to create DSLs can be drastically reduced. In a broader sense, C-Mera can also be attributed to this class of frameworks.

3. BRIEF REVIEW OF C-Mera

C-Mera is a simple transcompiler embedded in Common Lisp. It allows writing programs in an S-Expression syntax that is transformed to C-style code. This means that very simple extensions for languages with similar syntax are provided on top of the core C support. For example, the C-Mera distribution provides modules for C++, CUDA, GLSL and OpenCL. The main goal of providing an S-Expression syntax is to write the compiler such that it evaluates this syntax to construct a syntax tree when the input program is read, thereby allowing interoperability with the Common Lisp-system, most importantly by providing support for Lisp-style macros. To keep this part short we refer to the original C-Mera paper [17] for a more detailed description of the system and its implementation.

With the use of macros the input program no longer represents a plain syntax tree, but a semantically annotated tree that is transformed according to the implementation of the semantic nodes (macros). The utility of such a system ranges from simple, ad-hoc abstractions and programmer-centric simplifications [17] to providing otherwise hard to achieve programming paradigms for C-like languages [16] and even to fully fledged domain-specific languages.

The following example, taken from our domain, shows the definition of a simple image filter:

[3] github.com/slyrus/opticl

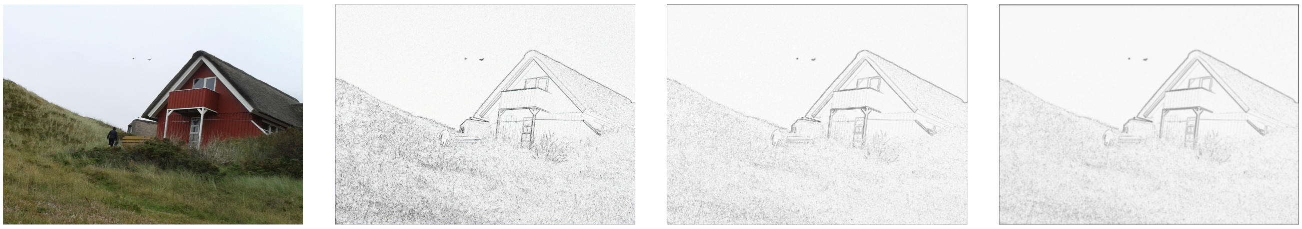

(a) Input image (b) Laplace filtered (c) Blur & Laplace (d) Blur, Laplace & blur

Figure 1: Input image and results from using edge detection via a Laplace operator.

```
(function filter ((float *data) (float *mask) (int filter-w)
                   (int filter-h) (int w) (int h)) -> void
  (for ((int y 0) (< y h) ++y)
    (for ((int x 0) (< x w) ++x)
      (decl ((float accum 0.0f))
        (for ((int dy 0) (< dy filter-h) ++dy)
          (for ((int dx 0) (< dx filter-w) ++dx)
            (set accum
                 (+ accum
                    (* (aref mask (+ (* filter-w dy) dx))
                       (aref data
                             (+ (* (+ y (- (/ filter-h 2)) dy)
                                   w)
                                x (- (/ filter-w 2)) dx)))))))
        (set (aref data (+ (* y filter-w) x))
             accum)))))
```

This describes a C function that takes an array as its input (e.g. a grayscale image) and applies a dimensional filter mask. Filtering proceeds by iterating over the input image. The weighted average is computed for each pixel using the provided filter mask (centered at the current pixel). Using C-Mera it is easy to reduce the code for this algorithm to a simplified description, as presented in the following listing.

```
(defilter filter (data mask (w h) (filter-w filter-h))
  (loop2d (x y w h)
    (decl ((float accum 0.0f))
      (loop2d (dx dy filter-w filter-h)
        (set accum (* (mask dx dy)
                      (cell (+ x (- (/ filter-w 2)) dx)
                            (+ y (- (/ filter-h 2)) dy)))))
      (set (cell x y) accum))))
```

This is achieved by a set of simple macrolets:

```
(defmacro defilter
    (name (data mask (w h) (filter-w filter-h)) &body body)
 `(function ,name
            ((float* ,data) (float* ,mask)
           ,@(loop for x in (list w h filter-w filter-h)
                  collect `(int ,x))) -> void
     (macrolet
       ((loop2d ((x y w h) &body body)
                `(for ((int ,y 0) (< ,y ,h) (+= ,y 1))
                   (for ((int ,x 0) (< ,x ,w) (+= x 1))
                     ,@body)))
        (mask (x y) `(aref ,',mask (+ (* ,',filter-w ,y) ,x)))
        (cell (x y) `(aref ,',data (+ (* ,',w ,y) ,x))))
       ,@body)))
```

Naturally, further shorthands and simplifications can be incorporated into this kind of macro. Section 4 shows the language that evolved from these considerations and Section 5 shows to more advanced examples.

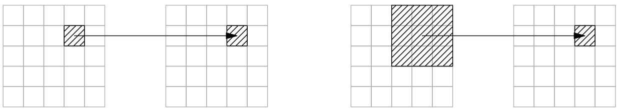


Figure 2: Illustration of point operators (left) and local filters (right).

```
(filter-graph laplacian
  (edge load-base (:output base)
    (load-image :file "test.jpg"))
  (edge laplacian (:input base :output lapla :arch cuda)
    (deflocal
       :mask ((-1 -1 -1)
              (-1  8 -1)
              (-1 -1 -1))
       :aggregator +
       :operator *
       :finally (set accum.x (- 255 (fabs accum.x))
                     accum.y (- 255 (fabs accum.y))
                     accum.z (- 255 (fabs accum.z)))))
  (edge store-ublapla (:input lapla)
    (store-image :file "out.png")))
```

Figure 3: A simple, but complete, Chipotle-program that filters an input image using a Laplace operator.

4. THE Chipotle-DSL

In this section we describe Chipotle, our domain-specific language for image processing. Chipotle provides a very concise notation, is easily extensible and expands into high-performance code for both GPUs (using CUDA) and CPUs (using SSE and AVX). It furthermore provides heterogeneous scheduling based on simple tagging.

As already mentioned, Chipotle is heavily inspired by HIPA^{cc}. HIPA^{cc} is based on the Clang/LLVM infrastructure, which introduces two major drawbacks. First of all the DSL syntax is restricted to its host language C++ and causes the DSL to be rather verbose. Secondly, extending the DSL with new language constructs is a very time consuming process. HIPA^{cc} uses the Clang-AST to substitute certain nodes with domain-specific variants. This AST, however, is generated after a complete semantic analysis of the input C++ code and is thus extremely detailed. Filtering the relevant nodes to extend and adapt the AST for a particular domain is a cumbersome task. Therefore, Chipotle was designed to counter these complications by providing a very concise and declarative style that aims to be easily extensible.

4.1 Notation

As a running example, we will consider the definition of a simple Laplace operator (see Figure 1 (b)). In image processing, the Laplace operator is a simple filter that can be used for edge detection. The code listed in Figure 3 shows how this operator can be expressed using Chipotle. The principal component of a Chipotle-program is the *filter graph*. The contents of a filter graph are nodes that represent (intermediate) images and *edges* that specify transformations on those images. In the code given in Figure 3 there are three image operations: loading an existing image from disk,

```
(edge box (:input base :output filtered :arch cuda)
  (deflocal
    :extent (5 5)
    :grayscale t
    :accum-name val
    :codelet (set val (+ val (local-ref base rel-x rel-y)))
    :finally (set val (/ val 25))))
```

Figure 4: A box-filter stage illustrating the use of `codelet`. The same operation can be implemented by using a filter mask of all ones.

filtering it with a local operator and finally storing it back to disk. The images themselves are implicit in the connection information given with the edges (e.g. `laplacian` takes input from `base` and stores its result in `lapla`).

The largest part of the `laplacian` graph is the description of the local operator. A local operator (see Figure 2), L, is a function on an image, I, mapping a neighborhood, N, of a given pixel of to an single output pixel value [1]:

$$L(x, y) = \oplus_{(i,j)\in N} f_I(x, y, i, j)$$

The most common case of this is discrete convolution using a convolution matrix M (as `mask` in the example above):

$$L(x, y) = \sum_{i,j=0}^{i,j<n} M_{ij} I_{x+i-\lfloor n/2\rfloor, y+j-\lfloor n/2\rfloor}$$

Inspired by HIPA^{cc}'s implementation we provide an implicit looping mechanism where only the operator and aggregator, $\otimes$ and $\oplus$, respectively, must be specified and we assume

$$f_I(x, y, i, j) = M_{ij} \otimes I_{x+i-\lfloor n/2\rfloor, y+j-\lfloor n/2\rfloor}.$$

The result of the local filter can be further adapted via the `:finally` clause. In the example above we use this clause to store the absolute value of the result. It is also possible to specify an arbitrary function for f by providing a `:codelet`, as with the simple box filter shown in Figure 4. There, the relative positions during the iteration are available in `rel-x` and `rel-y` (naturally, the names of such generator-defined variables, as found throughout this section, can be specified explicitly, too). For a more elaborate example see Section 5.1.

In accordance with Bankman [1] and HIPA^{cc} we also provide a simpler form, the point operator (see Figure 2). Instead of mapping a region of the input image to an output pixel, point operators map input pixels to output pixels without considering the pixel's neighborhood. The following fragment is part of the well-known Harris corner detector [5] that determines whether a pixel is part of a corner in the input. The computation depends on the two input images `xd` and `yd`, which hold the gradients of the original input in x and y direction.

```
(edge hcd (:input (xd yd) :output out :arch cuda)
  (defpoint (:grayscale t)
    (decl ((float xx (* xd xd))
           (float xy (* xd yd))
           (float yy (* yd yd))
           (float M (abs (- (- (* xx yy) (* xy xy))
                            (* 0.04 (+ xx yy) (+ xx yy)))))
           (float res 0))
      (if (< threshold M) (set res 255))
      (set out res))))
```

Since, for point operators, the neighborhood should not be available the names of the input images are mapped to reference the current pixel-location in the respective images.

```
1   (filter-graph laplacian
2     (edge load-base (:output base)
3       (load-image :file "test.jpg"))
4     (edge gauss (:input base :output blurred :arch cuda)
5       (deflocal
6          :mask #.(sample-filter #'gaussian 5 :sigma 1.5)
7          :aggregator +
8          :operator *))
9     (edge laplacian (:input blurred :output lapla :arch cuda)
10      (deflocal
11         :mask ((-1 -1 -1)
12                (-1  8 -1)
13                (-1 -1 -1))
14         :aggregator +
15         :operator *
16         :finally (set accum.x (- 255 (fabs accum.x))
17                       accum.y (- 255 (fabs accum.y))
18                       accum.z (- 255 (fabs accum.z)))))
19    (edge to-grayscale (:input lapla :output gray :arch cuda)
20      (defpoint ()
21         (decl ((float out (+ (* 0.2126 (lapla 0))
22                              (* 0.7152 (lapla 1))
23                              (* 0.0722 (lapla 2)))))
24           (set (gray 0) out
25                (gray 1) out
26                (gray 2) out))))
27    (edge gauss2 (:input gray :output output :arch cuda)
28      (deflocal
29         :mask #.(sample-filter #'gaussian 7 :sigma 4)
30         :grayscale t
31         :aggregator +
32         :operator *))
33    (edge store-out (:input output)
34      (store-image :file "out.png")))
```

Figure 5: The complete filter graph that transforms the image shown in Figure 1 (a) to that of (d).

4.2 Filter Graphs

As visible in the short graph given in Figure 3, the body of a filter graph consists of the edges that describe how images are transformed. Note that the body of the graph is evaluated and thus can also hold arbitrary forms, for example a set of user- or subdomain-specific macrolets (see below).

In our current implementation the roots of the graph (load operations) are found and used as a seed for topological sorting. Inconsistent graphs (e.g. containing unavailable input nodes) are rejected. Input and output is meant to be executed on the host-CPU, however this is by no means a systematic restriction and, in a future version, we plan on being able to connect Chipotle to hardware-rendered input images or interactive display using OpenGL.

Expanding on the example of using a Laplace operator at the outset of Section 4.1 the filtered version of Figure 1 (a), shown in (b), exhibits strong noise. This is due to the fact that the detection operator picks up very fine detail. Therefore it is common to pre-process images with a low-pass filter to remove small-scale detail prior to detection operators. Figure 1 shows further versions where the image (c) has been filtered with a Gaussian kernel before edge detection (and converted to grayscale) and (d) with an additional low-pass filter applied after edge detection to obtain a smoother image. Figure 5 shows the complete filter graph for this process. Note how the weights for the Gaussian kernels are computed beforehand (line 6 and 29). Figure 6 lists the same algorithm, but with a few convenience macros provided externally. For an example with proper macros in a filter graph see Section 5.

```
(filter-graph laplacian
  (edge load-base (:output base)
     (load-image :file "test.jpg"))
  (simple-filter base blurred
     (:arch cuda :mask #.(sample-filter #'gauss 5 :sigma 1.5)))
  (simple-abs-filter blurred lapla
     (:arch cuda :mask ((-1 -1 -1)
                        (-1  8 -1)
                        (-1 -1 -1))))
  (grayscale-conversion
      lapla gray :arch cuda)
  (simple-filter gray output
      (:arch cuda :mask #.(sample-filter #'gauss 7 :sigma 4))
      :grayscale t)
   (edge store-out (:input output)
      (store-image :file "out.png")))
```

Figure 6: The same graph as shown in Figure 5, however with a few convenience macros.

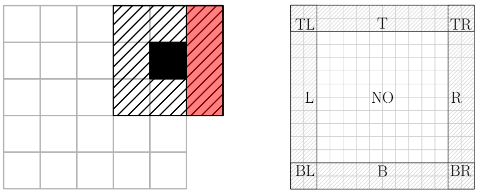


Figure 7: Boundary checks are required to ensure that only valid pixel locations are queried (left). To remove irrelevant bounds checks (e.g. for the inner part of the image, labelled NO) the image is partitioned (right).

4.3 Checked Memory Access

The local operators shown above are notationally very simple and do not contain explicit boundary checks. These checks are automatically introduced for the iteration over the filter mask such that, for example, when accessing pixels to the right of the target pixel's location CHIPOTLE only inserts checks for the right image border (see Figure 7).

For accessing locations outside the image $I(x, y)$ (i.e. $x \notin \{0, \ldots, w-1\} \vee y \notin \{0, \ldots, h-1\}$) different modes for changing the input coordinates, inspired by OPENGL's texture wrapping functions [19], are provided. The keyword parameter `:wrap` can be used to this end to compute x', y' with `mirror` ($x' = w - x$), `wrap` ($x' = x \mod w$) and `clamp` ($x' = \min(w-1, \max(0, x))$). Furthermore, `border` ($I(x, y) = \text{const}$) provides a constant border color.

In order to efficiently compute a filter over large images, it is also possible to partition the input image into areas requiring different boundary checks [11]. Since local filters usually employ a very small filter mask compared to the image size this ensures that no boundary checks are performed at all for the inner, and largest, part of the image. This combines well with generating border conditions based on the location in the filter mask: a check is only generated if the position in the filter mask is, for example, to the left and the active region after partitioning includes the left image edge. In the setting shown in Figure 7 (left), the local operator requires checked accesses only towards the right. Figure 7 (right) shows the different regions. Inspired by HIPA^{cc}'s implementation we provide a single function that checks for the appropriate boundary-handling scheme to be used and jumps to it. The benefit of this scheme is that it maps well to our target architectures (CUDA, SSE and AVX) as long as it does not introduce divergence, which is easily ensured (see Section 4.4).

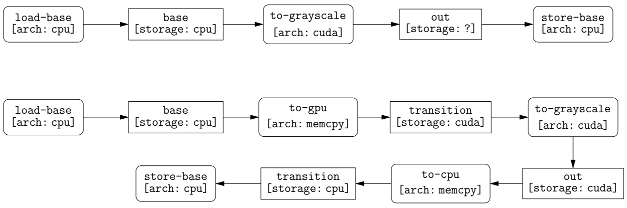


Figure 8: Top: Execution plan for the trivial graph shown in Figure 9 (top). Images are rectangular nodes, operators rounded. Bottom: Plan for the same graph, but with `:arch cuda` for the point operator, including transition edges and images.

4.4 Heterogeneous Image Processing

CHIPOTLE allows transforming its input programs to SSE and AVX with multi-threading as well as to CUDA. The target architecture of an image operation can be specified via the `:arch` parameter (see Figure 3). Note that different target architectures can be mixed freely in the same program.

For SSE and AVX the operations map to multi-threaded, two dimensional iterations over the input image. The specified operation (for local operators including the iteration over the filter mask) is then executed in groups of 4 (SSE) or 8 (AVX) pixels. This is achieved by automatically vectorizing of the provided code. To this end we map the content of declarations (`(decl (...)...)` to appropriate vectorized types and transform the user-provided arithmetic operations to corresponding vectorized versions. We also sequentialize conditional statements and track the masks of the true and false cases to ensure correct merging. Figure 9 shows an example of these operations for a very simple point operator, converting a color image to grayscale.

For operators to be instantiated for CUDA we generate a kernel function and a host-stub that addresses appropriate parameter forwarding. Transferring image data to the GPU and back to the host memory is implicit in the graph. Host-only operations such as loading and storing images (see Section 4.2 on this limitation) force the initial and final locations of image data. Furthermore, edges that operate on CUDA require that their input and output be present on the GPU. This is resolved by traversing the filter graph in order and introducing transition edges and images where appropriate. The location of images is propagated through the graph and only switches on the previously introduced transition nodes. For operators that do not specify a target architecture it is propagated similarly, to avoid expensive host/device transfers.

When targeting CPU vector instructions (SSE or AVX) or parallel GPU code (CUDA) we take care to incorporate the execution width in boundary handling conditions. For example, with AVX the generated code executes an 8 × 1 image region using a single control flow. If different paths of control flow are to be applied within such a group the code must be sequentialized. Therefore, bounds checks are conservatively clamped to multiples of 8 in the x direction for AVX. For CUDA, where the execution configuration is more flexible, borders are adjusted accordingly.

```
(filter-graph example
  (edge load-base (:output base)
    (load-image :file "test.jpg"))
  (edge gray (:input base :output out :arch sse)
    (defpoint ()
      (decl ((const float r (base 0))
             (const float g (base 1))
             (const float b (base 2))
             (const float luma (+ (* 0.2126 r) (* 0.7152 g)
                                  (* 0.0722 b)))
             (float res))
        (if (> luma .5)
            (set res luma)
            (set res 0))
        (set (out 0) res))))
  (edge store (:input out)
    (store-image :file "out.png")))
```

```
void gray(unsigned char *base, unsigned char *out, unsigned int w, unsigned int h)
{
    unsigned int vecLength = (w * h) - ((w * h) % 4);
    const __m128 xmm_constant_0_5__165    = _mm_set1_ps(5.00000000e-1);
    const __m128 xmm_constant_0_0722__164 = _mm_set1_ps(7.22000000e-2);
    const __m128 xmm_constant_0_7152__163 = _mm_set1_ps(7.15200000e-1);
    const __m128 xmm_constant_0_2126__162 = _mm_set1_ps(2.12600000e-1);
    const __m128 xmm_constant_1_0__161    = _mm_set1_ps(1.00000000e+0);
    for (unsigned int i = 0; i < vecLength; i += 4) {
        //Load: (base 2) to xmm290
        const __m128i xmm291 = _mm_cvtsi32_si128((*((const int*)
                                                   &base[(i + (2 * w * h))])));
        const __m128i xmm292 = _mm_unpacklo_epi8(xmm291, _mm_setzero_si128());
        const __m128i xmm293 = _mm_unpacklo_epi16(xmm292, _mm_setzero_si128());
        const __m128 xmm290 = _mm_cvtepi32_ps(xmm293);
        //Load: (base 1) to xmm289
        const __m128i xmm294 = _mm_cvtsi32_si128((*((const int*)
                                                   &base[(i + (1 * w * h))])));
        const __m128i xmm295 = _mm_unpacklo_epi8(xmm294, _mm_setzero_si128());
        const __m128i xmm296 = _mm_unpacklo_epi16(xmm295, _mm_setzero_si128());
        const __m128 xmm289 = _mm_cvtepi32_ps(xmm296);
        //Load: (base 0) to xmm288
        const __m128i xmm297 = _mm_cvtsi32_si128((*((const int*)
                                                   &base[(i + (0 * w * h))])));
        const __m128i xmm298 = _mm_unpacklo_epi8(xmm297, _mm_setzero_si128());
        const __m128i xmm299 = _mm_unpacklo_epi16(xmm298, _mm_setzero_si128());
        const __m128 xmm288 = _mm_cvtepi32_ps(xmm299);
        const __m128 r = xmm288;
        const __m128 g = xmm289;
        const __m128 b = xmm290;
        const __m128 luma = _mm_add_ps(
                              _mm_add_ps(_mm_mul_ps(xmm_constant_0_2126__162, r),
                                         _mm_mul_ps(xmm_constant_0_7152__163, g)),
                              _mm_mul_ps(xmm_constant_0_0722__164, b));
        __m128 res;
        const __m128 cond395 = _mm_cmpgt_ps(luma, xmm_constant_0_5__165);
        const __m128 mask396 = cond395;
        res = _mm_or_ps(_mm_and_ps(mask396, luma), _mm_andnot_ps(mask396, res));
        const __m128 mask397 = _mm_andnot_ps(cond395,
                                             _mm_set1_ps(xmm_constant_1_0__161));
        res = _mm_or_ps(_mm_and_ps(mask397, 0), _mm_andnot_ps(mask397, res));
        //Store: (out 0)
        const __m128i xmm1129 = _mm_cvtps_epi32(res);
        const __m128i xmm1130 = _mm_packs_epi32(xmm1129, xmm1129);
        const __m128i xmm1131 = _mm_packus_epi16(xmm1130, xmm1130);
        (*((int*)&out[(i + (0 * w * h))])) = _mm_cvtsi128_si32(xmm1131);
    }
    for (unsigned int i = vecLength; i < (w * h); ++i){
        const float r = base[i + (0 * w * h)];
        const float g = base[i + (1 * w * h)];
        const float b = base[i + (2 * w * h)];
        const float luma = (2.12600000e-1 * r) + (7.15200000e-1 * g)
                         + (7.22000000e-2 * b);
        float res;
        if (luma > 5.00000000e-1)
            res = luma;
        else
            res = 0;
        out[i + (0 * w * h)] = res;
    }
}
```

Figure 9: Top: Chipotle input graph for converting a color image to grayscale. Bottom: Generated SSE code for the `gray` edge.

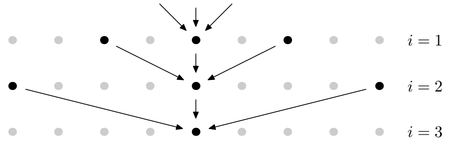


Figure 10: À-Trous filtering: Iterative application of a small filter with increasing gap between samples.

5. EXAMPLES

In this section we provide two examples of using Chipotle and describe further language features and practical considerations that arise.

5.1 Edge-Avoiding À-Trous Filter

Many image processing algorithms (e.g. the night filter following this example) require an image smoothing step that does not filter across edges, that is, a filter that smoothes regions that are similar, but maintains the sharpness of the image. This effect can be achieved by using a bilateral filter [21]. Such filters do not only weight pixel values by their spatial distance (such as the previous local fiters, e.g. the Gauss kernel), but also by difference in value. However, due to non-linearity, these filters are not separable and are thus very expensive to compute for large regions.

A common way to accelerate the computation of this filter is using the À-Trous (with holes) algorithm [18], where a filter with small support is iteratively applied while increasing the image-space gap between the sample locations in each iteration. Figure 10 illustrates this for a one-dimensional filter. Such a filter is easily constructed by filling in the appropriate number of zero-entries in the filter mask and it can then be used as a simple local operator. The large neighborhood introduced by this is easily reduced when checking for zero-coefficients while unrolling the loop over the local filter. Thus, for a Gaussian 3×3 À-Trous filter at iteration 3 the filter mask is 17×17 but the actual number of operations (and bounds checks, if appropriate) is still 9.

Our macro, which provides the code for the edge-avoiding À-Trous filter, is given in Figure 11. It shows how more complicated operators can be made available to various filter graphs (see the next example, for instance). As the bilateral filter is not a simple accumulation we exploit the flexibility of our `deflocal` implementation. After fixing parameter and filter names at the beginning, we set up our accumulators and reference values in line 12. There, `r0` references the red color component at the pixel of the filter's center. We accumulate color to `r`, `g` and `b` and also accumulate the weight, `W`, by which the result must normalized. The provided `:codelet` is then evaluated for the non-zero entries of the filter mask and combines the weights in the domain (i.e. the mask's value) and in the range (i.e. weighted by similarity, e.g. by an exponential term). Finally, we normalize the result and write it back to memory.

5.2 Night Tonemapping

The standard method of transforming images taken under daytime lighting conditions to look as if taken by night is by reducing brightness, blurring similar colors (reducing acuity), and shifting the colors towards more bluish tones [20]. In our implementation of such an algorithm we first blur similar regions in the input image by using the bilateral filter shown in the previous example. After a few iterations of this local operator we compute the actual scotopic image by applying a blue shift. To this end we follow Jensen et al. [6] and first convert the image to the XYZ color space, reduce the brightness, Y, compute the scotopic luminance [9], V, and use it to compute a darkened image that is shifted to a more bluish tone. Finally, we convert the image back to the RBG color space. Figure 12 shows the effect computed by the filter graph given in the lower part of Figure 11.

```
 1  (defmacro atrous-step (n &key (pre "atrous") input output arch)
 2    (let ((in  (cl:if input input
 3                        (cintern (format nil "~a~a" pre (cl:1- n)))))
 4          (out (cl:if output output
 5                        (cintern (format nil "~a~a" pre n)))))
 6      ‘(edge ,(cintern (format nil "compute-~a~a" pre n))
 7           (:input ,in :output ,out :arch ,arch)
 8         (deflocal
 9           :mask ,(atrous ’((0.057118 0.124758 0.057118)
10                             (0.124758 0.272496 0.124758)
11                             (0.057118 0.124758 0.057118)) n)
12           :initially ((float r0 (/ (,in 0 0 0) 255.0f))
13                        (float g0 (/ (,in 0 0 1) 255.0f))
14                        (float b0 (/ (,in 0 0 2) 255.0f))
15                        (float r 0) (float g 0) (float b 0)
16                        (float W 0))
17           :codelet
18               (decl ((float R (/ (,in rx ry 0) 255.0f))
19                       (float G (/ (,in rx ry 1) 255.0f))
20                       (float B (/ (,in rx ry 2) 255.0f))
21                       (float w0 (mask rx ry))
22                       (float rd (- R r0))
23                       (float gd (- G g0))
24                       (float bd (- B b0))
25                       (float w1 (+ (^2 rd) (^2 gd) (^2 bd))))
26                 (set w1 (* (fminf 1.0f (expf (- (* w1 1)))) w0))
27                 (set W (+ W w1))
28                 (set r (+ r (* R w1))
29                     g (+ g (* G w1))
30                     b (+ b (* B w1)))))
31           :finally (set (,out 0) (* 255.0f (/ r W))
32                          (,out 1) (* 255.0f (/ g W))
33                          (,out 2) (* 255.0f (/ b W)))))))
```

```
 1  (defmacro nightvision-filter (&key (iterations 3))
 2    ‘(filter-graph blub
 3       (edge load-base (:output base) (load-image :file "test.jpg"))
 4
 5       (atrous-step 0 :input base)
 6       (atrous-step 1)
 7       ,@(loop for i from 1 to (cl:1- iterations)
 8            collect ‘(atrous-step ,i))
 9       (atrous-step ,iterations :output prefiltered)
10
11       (edge scoto (:input prefiltered :output scotopic :arch cuda)
12         (defpoint ()
13           (decl ((float r (prefiltered 0))
14                  (float g (prefiltered 1))
15                  (float b (prefiltered 2))
16                  (float X (to-X r g b)
17                  (float Y (* (to-Y r g b) 0.33f))
18                  (float Z (to-Z r g b))
19                  (float V (scotopic-luminance X Y Z))
20                  (float W (+ X Y Z))
21                  (float s (* Y 0.2))
22                  (float xl (/ X W))
23                  (float yl (/ Y W))
24                  (const float xb 0.25)
25                  (const float yb 0.25))
26             (set xl (+ (* (- 1.0f s) xb) (* s xl))
27                  yl (+ (* (- 1.0f s) yb) (* s yl))
28                  Y  (+ (* V 0.4468f (- 1 s)) (* s Y))
29                  X  (/ (* xl Y) yl)
30                  Z  (- (/ X yl) X Y))
31             (decl ((float rgb_r (to-r X Y Z))
32                    (float rgb_g (to-g X Y Z))
33                    (float rgb_b (to-b X Y Z)))
34               (set (scotopic 0) (fminf 255.0f (fmaxf 0.0f rgb_r)))
35               (set (scotopic 1) (fminf 255.0f (fmaxf 0.0f rgb_g)))
36               (set (scotopic 2) (fminf 255.0f (fmaxf 0.0f rgb_b)))))))
37
38       (edge store-scotopic (:input scotopic)
39         (store-image :file "night.jpg"))))
40
41  (nightvision-filter :iterations 3)
```

Figure 11: Top: Our macro that generates different iterations of the edge-avoiding À-Trous blur filter. Bottom: The filter is used as a pre-process for the night filter.

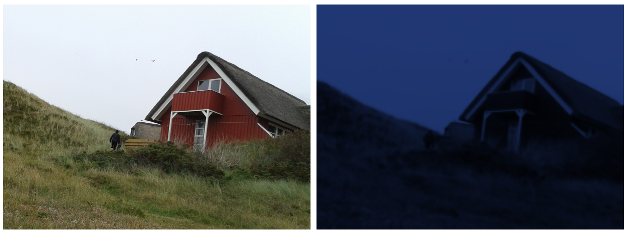


Figure 12: Input image (left) and night-filtered version (right). Note how not only the tone changed, but also many details are blurred out while edges (such as the roof, columns and the balcony) are still clearly visible.

6. EVALUATION

In the following we give a brief evaluation of implementing filter graphs using our DSL, Chipotle, and compare them to HIPA^{cc}. We focus on the night filter with three iterations of edge-avoiding À-Trous filtering (see Section 5.1) followed by night tonemapping (see Section 5.2). This corresponds to evaluating line 41 in the lower part of Figure 11.

In terms of filtering performance CUDA code generated by HIPA^{cc} takes 14.7 ms to filter the 1754×1280 image shown in Figure 12 (left) on a Nvidia Geforce GTX 680 graphics card. HIPA^{cc}'s SSE2 version takes 400 ms running on a Intel Xeon E5-1620 processor running at 3.50 GHz, using OpenMP for parallelization. Our code generated by Chipotle runs equally fast at 14.9 ms on CUDA, while our SSE2 version lags behind at 901 ms, with the same hardware. Due to the use of domain knowledge and the ability to generate code for special cases (e.g. for border handling) these computation times are hard to achieve with hand-written code [11].

Regarding code size there are two factors we consider: firstly the size of the code written in the DSL, and secondly the size of the DSL's code base itself, which is of particular importance when considering extensions and maintenance of the DSL. The HIPA^{cc}-version of the night-filter shown in Figure 11 (bottom) consists of 264 lines of code. The complete code for use with Chipotle, including the expansion of a simple input mask to an À-Trous filter (`atrous`, see line 9 in Figure 11 (top)) and the edge-avoiding filter, totals 85 lines. This is mainly due to the fact that our notation contains almost no boilerplate code.

The HIPA^{cc} distribution we used to take the above measurements consists of 48941 lines of code. As described earlier this includes a number of additional back-ends that are not available for Chipotle (most notably Renderscript and VivadoHLS for which there is no support in C-Mera). However, even with the additional back-ends the code base appears immense when compared to Chipotle's 978 lines of code (as-is, and including vectorization).

7. CONCLUSION

In this paper we presented our new image processing DSL, Chipotle, and showed how easily it is constructed using an existing, Common Lisp-based tool chain. At only 2% of the code size of competing methods our DSL yields highly optimized code that runs on-par with the state of the art on GPUs using CUDA. Our SSE2 version runs at around 50% the performance of HIPA^{cc}'s vectorized output. It should be noted that, even in this case, our performance is significantly faster than results from using a compiler's auto-vectorization routines as these cannot rely on domain knowledge. However, we believe that this performance gap is an artefact of our not yet fully matured vectorization routines and that CHIPOTLE will catch up with HIPA^{cc} shortly. We further believe that extensions to our DSL are much simpler as its implementation is very short and uses higher-level programming paradigms, most notably COMMON LISP macros and feature-oriented programming [16].

We also showed that the implementation of a filter graph using CHIPOTLE is only around 33% of the code size of the corresponding HIPA^{cc} implementation and that the DSL code itself is still amenable to further abstractions and simplifications using macros. Thus, we are confident that CHIPOTLE can compete with state of the art image processing DSLs while increasing productivity both on the level of the DSL users and implementors.

Acknowledgments

The authors gratefully acknowledge the generous funding by the German Research Foundation (GRK 1773).

8. REFERENCES

[1] I. N. Bankman. *Handbook of Medial Image Processing and Analysis*. Academic Press, Burlington, second edition edition, 2009.

[2] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 89–100, Oct 2011.

[3] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, 2011.

[4] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA, 2013. ACM.

[5] C. Harris and M. Stephens. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.

[6] H. W. Jensen, S. Premoze, P. Shirley, W. B. Thompson, J. A. Ferwerda, and M. M. Stark. Night rendering. Technical Report UUCS-00-016, Computer Science Department, University of Utah, Aug. 2000.

[7] S. Kamil, D. Coetzee, and A. Fox. Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2011.

[8] M. Köster, R. Leißa, S. Hack, R. Membarth, and P. Slusallek. Code Refinement of Stencil Codes. *Parallel Processing Letters (PPL)*, 24(3):1–16, Sept. 2014.

[9] G. W. Larson, H. Rushmeier, and C. Piatko. A visibility matching tone reproduction operator for high dynamic range scenes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):291–306, Oct. 1997.

[10] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[11] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. HIPAcc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):210–224, 2016.

[12] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, 2015.

[13] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, September 2015.

[14] NVIDIA Corporation. *Parallel Thread Execution ISA Version 4.3*, September 2015.

[15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[16] K. Selgrad, A. Lier, F. Köferl, M. Stamminger, and D. Lohmann. Lightweight, generative variant exploration for high-performance graphics applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 141–150, New York, NY, USA, 2015. ACM.

[17] K. Selgrad, A. Lier, M. Wittmann, D. Lohmann, and M. Stamminger. Defmacro for C: Lightweight, ad hoc code generation. In *Proceedings of ELS 2014 7rd European Lisp Symposium*, pages 80–87, 2014.

[18] M. J. Shensa. The discrete wavelet transform: wedding the a trous and mallat algorithms. *IEEE Transactions on Signal Processing*, 40(10):2464–2482, 1992.

[19] D. Shreiner and T. K. O. A. W. Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1.* Addison-Wesley Professional, 7th edition, 2009.

[20] W. B. Thompson, P. Shirley, and J. A. Ferwerda. A spatial post-processing algorithm for images of night scenes. *J. Graphics, GPU, & Game Tools*, 7(1):1–12, 2002.

[21] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, pages 839–, Washington, DC, USA, 1998. IEEE Computer Society.

Session III: Implementation

A modern implementation of the LOOP macro

Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
robert.strandh@u-bordeaux.fr

ABSTRACT

Most Common Lisp [1] implementations seem to use a derivative of MIT `loop` [4]. This implementation predates the Common Lisp standard, which means that it does not use some of the features of Common Lisp that were not part of the language before 1994. As a consequence, the `loop` implementation in all major Common Lisp implementation is *monolithic* and therefore hard to maintain and extend.

Furthermore, MIT `loop` is not a conforming `loop` implementation, in that it produces the wrong result for certain inputs. In addition, MIT `loop` accepts sequences of `loop` clauses with undefined behavior according to the standard, though whether such extended behavior is a problem is debatable.

We describe a modern implementation of the Common Lisp `loop` macro. This implementation is part of the SICL[1] project. To make this implementation of the macro modular, maintainable, and extensible, we use *combinator parsing* to recognize `loop` clauses, and we use CLOS generic functions for code generation.

CCS Concepts

•**Software and its engineering → Control structures;**

Keywords

CLOS, Common Lisp, Iteration, Combinator parsing

1. INTRODUCTION

The `loop` macro is part of the Common Lisp standard [1], so every conforming Common Lisp implementation contains an implementation of this macro. Appendix A contains the part of the syntax of the `loop` macro that is relevant for this paper.

The `loop` macro is frequently criticized as un-Lispy since it does not use S-expressions for its clauses, and for being impossible to extend, at least by using only features available in the Common Lisp standard. In addition, advocates of purely-functional programming also criticize it, along with all other iteration constructs that can not be explained in terms of recursion.

Despite all this criticism, the `loop` macro is an essential and widely used part of any non-trivial Common Lisp program. It is able to satisfy the vast majority of iteration needs. In addition, it is far easier to understand than equivalent loops using other iteration constructs such as `dotimes`, `dolist`, and `do`.

Most current implementations of Common Lisp seem to use an implementation of the `loop` macro that was largely written before the Common Lisp standard was adopted. Consequently, some of the interesting features of the standardized Common Lisp language are not used in the implementation of the `loop` macro in these implementations. In particular, the use of *generic functions* is typically minimal. As a result, the implementation of this macro is quite *monolithic*, making it hard to maintain, whether in order to remove defects or to extend it.

We present a modern implementation of the `loop` macro. This implementation was written as part of the SICL project, of which one of the explicit goals is to use improved coding techniques.

We are able to obtain a more modular `loop` implementation by using two key techniques. The first one is to parse the clauses using a parsing technique that allows for individual clause parsers to be *textually separated* according to clause type. The second modularity technique is to use *generic functions* for semantic analysis and code generation. By defining clause types as standard classes, we are able to textually separate processing according to clause type through the use of methods specialized to these clause classes.

2. PREVIOUS WORK

2.1 MIT LOOP with variations

One of the first implementations of the Common Lisp `loop` macro is the one that is often referred to as "MIT `loop`" [4]. A popular variation of this implementation includes modifications by Symbolics Inc.[2]

This implementation of the `loop` macro is sometimes more permissive than the Common Lisp standard. For example, the standard requires all *variable clauses* to precede all *main*

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

[2] Symbolics Inc was a company that sold Lisp machines. See https://en.wikipedia.org/wiki/Symbolics for a thorough description of the company and its products.

9th ELS May 9–10, 2016, Kraków, Poland

clauses. (See Appendix A.) Code such as the one in this example:

```
(loop until (> i 20)
      for i from 0
      do (print i))
```

is thus not conforming according to the standard, since `until` is a *main clause* whereas `for` is a *variable clause*. However, MIT `loop` and its variation accepts the code in the example.

Another example of non-conforming behavior is illustrated by the following code:

```
(loop for i from 0 below 10
      sum i
      finally (print i))
```

The Common Lisp standard clearly states that the loop variable does not take on the value of the upper limit, here 10, so the value printed in the `finally` clause should be 9. However, `loop` implementations derived from MIT `loop` print 10 instead.

Notice that the two examples above are non-conforming in two different ways, as explained in section 1.5 of the Common Lisp standard.

In the first case, we have an example of *a non-conforming program* as explained in section 1.5.2 for the simple reason that the standard does not specify what an implementation must do when the clause order is violated. By default, then, the behavior is said to be *undefined*, meaning that the implementation is free to reject the non-conforming program or to accept it and interpret it in some (perhaps unexpected) way. The MIT `loop` implementation is therefore conforming in this respect.

In the second case, we have an example of *a non-conforming implementation* as explained in section 1.5.1. The reason is that the standard clearly stipulates that every implementation must print 9, whereas MIT `loop` prints 10.

In addition to the non-conforming problems, MIT `loop` has issues with modularity, in that the implementation is *monolithic*, and that holds true for its variations too. The code is contained in a single file with around 2000 lines of code in it.

Code generation uses a significant number of special variables holding various pieces of information that are ultimately assembled into the final expansion of the macro.

2.2 ECL and Clasp

ECL[3] includes two implementations of the `loop` macro, namely the initial MIT `loop` with only minor modifications, and a variation of MIT `loop` that also includes code written by Symbolics Inc also with minor modifications.

Clasp[4] is a recent implementation of Common Lisp. It is derived from ECL in that the C [2] code of ECL has been translated to C++ [3] whereas most of the Common Lisp code has been included with no modification, including the code for the `loop` macro.

ECL `loop` being derived from MIT `loop`, the non-conforming examples shown in Section 2.1 are also accepted by ECL and Clasp.

2.3 SBCL

SBCL[5] includes an implementation of the `loop` macro that was originally derived from MIT `loop`, but that also includes code created by Symbolics Inc. Because of the way the code has evolved, it is hard to determine whether, at some point, the code of the `loop` macro of SBCL and that of ECL were the same, but a rough comparison suggests that this is the case.

The `loop` implementation of SBCL being derived from MIT `loop`, the non-conforming examples shown in Section 2.1 are also accepted by SBCL.

2.4 CLISP

CLISP has its own implementation of the `loop` macro. The bulk of the implementation can be found in a function named `expand-loop`. This function consists of more than 900 lines of code.

2.5 CCL

Like many other implementations, CCL[6] includes the variation of MIT `loop` containing modifications by Symbolics Inc from a brief inspection, we believe that the original code is the same as that of SBCL and ECL.

2.6 LispWorks

Evaluating the two examples in Section 2.1 using LispWorks[7] gives the same result as the implementations using MIT `loop`, suggesting that LispWorks also uses a derivative of that `loop` implementation.

3. OUR TECHNIQUE

3.1 Parsing clauses

In order to parse `loop` clauses, we use a simplified version of a parsing technique known as *combinator parsing* [10]. It is simplified in that we do not need the full backtracking power of this technique. It is fairly easy to structure the individual clause and sub-clause parsers so that a parser either succeeds or fails, simply because the `loop` grammar is unambiguous at this level. When a parser succeeds, the result is correct and unambiguous, and when it fails, other parsers are tried in sequence. Obtaining the correct result requires the individual parsers to be tried in a particular order, which is a disadvantage of the simplification. As discussed in Section 5.1, we plan to avoid this restriction by using an external parsing framework.

With this parsing technique, client code defines *elementary parsers* that are then combined using combinators such as *alternative* and *sequence*. The resulting parser code is *modular* in that individual parsers do not have to be listed in one single place. For the `loop` clauses, this modularity means that each type of clause can be defined in a different module.

In our parsing framework, an individual parser is an ordinary Common Lisp function that takes a list of Common Lisp expressions and that returns three values:

[3] ECL stands for “Embedded Common Lisp.
See: //https://gitlab.com/embeddable-common-lisp/ecl

[4] See: https://github.com/drmeister/clasp

[5] SBCL stands for Steel-Bank Common Lisp.
See: http://www.sbcl.org/

[6] CCL stands for Clozure Common Lisp.
See: http://ccl.clozure.com/

[7] See: http://www.lispworks.com/

1. A generalized Boolean[8] indicating whether the parse succeeded.

2. The result of the parse. If the parse does not succeed, then this value is unspecified.

3. A list of the tokens that remain after the parse. If the parse does not succeed, then this list contains the original list of tokens passed as an argument.

Consider the following example:

```
(define-parser arithmetic-up-1-parser
  (consecutive
   (lambda (var type-spec from to by)
     (make-instance 'for-as-arithmetic-up
       :order '(from to by)
       :var-spec var
       :type-spec type-spec
       :start-form from
       :end-form (cdr to)
       :by-form by
       :termination-test (car to)))
   'simple-var-parser
   'optional-type-spec-parser
   (alternative 'from-parser
                'upfrom-parser)
   (alternative 'to-parser
                'upto-parser
                'below-parser)
   'by-parser))
```

The macro `define-parser` defines a named parser. This parser consists of four consecutive parsers:

1. A parser that recognizes a simple variable. The result of this parser is the variable.

2. A parser that recognizes an optional type specifier. The result of this parser is the type specifier or `t` if the type specifier is absent.

3. A parser that recognizes one of the `loop` keywords `from` or `upfrom` followed by a form. The result of the parser is the form.

4. A parser that recognizes one of the `loop` keywords `to`, `upto`, or `below` followed by a form. The result of this parser is a `cons`, where the `car` is either the symbol < or the symbol <= depending on which keyword was recognized, and the `cdr` is the form.

The function defined by the `lambda` expression combines the results of those four parsers into a single result for the newly defined parser. In this example, the result of the new parser is an instance of the class `for-as-arithmetic-up`.

Initially, the `loop` body is parsed as a sequence of individual `loop` clauses, without any consideration for the order between those clauses. A failure to parse during this phase will manifest itself as an error relating to a particular clause, whether it is in a valid position or not. Furthermore, ignoring restrictions on clause ordering allows us to check the syntax of each clause. If order had been taken into account, we would either have to abandon the parsing phase when a syntactically correct clause were found in the wrong position and thereby being unable to verify subsequent clauses, or else we would have to implement some sophisticated error recovery, allowing the parsing process to continue after a failure.

Our technique for parsing clauses does not work very well for signaling useful errors when a clause fails to parse. In Section 5.2, we discuss our plans for improving the situation.

3.2 Representing parsed clauses

The result of the initial parsing process is a list of clauses, where each clause has been turned into an instance of (a subclass of) the class `clause`.

The classes representing different clauses are organized into a graph that mostly mirrors the names and descriptions of different clause types defined by the Common Lisp standard.

So for example, the class named `main-clause` is the root class of all clauses of that type mentioned in the standard. The same is true for `variable-clause`, `name-clause`, etc. (See Appendix A.)

Classes representing clauses that admit the `loop` keyword `and` also have a list of sub-clauses.

This organization allows us to capture commonalities between different clause types by defining methods on generic functions that are specialized to the appropriate class in this graph.

In addition to representing each clause as an instance of the `clause` class, we also represent the `loop` body itself as an instance of the class named `loop-body`. This instance contains a list of all the clauses, but also other information, in particular about default accumulation for this call to the `loop` macro.

3.3 Semantic analysis

We use generic functions to analyze the contents of the parsed clauses, and to generate code from them. The reason for using generic functions is again one of modularity. A method specialized to a particular clause type, represented by a particular standard class, can be textually close to other code related that clause type.

Checking the validity of the order between clauses is done in the first step of the *semantic analysis*, allowing us to signal pertinent error conditions if the restrictions concerning the order of clauses are not respected.

Next, we verify that the variables introduced by a clause are unique when it would not make sense to have multiple occurrences of the same variable. We also verify that there is at most one *default accumulation category*, i.e, one of the categories *list*, *min/max*, and *count/sum*.

3.4 Code generation

Our code generation consists of a direct expansion to lower-level Common Lisp code. We do not make any attempts to detect problems such as unused variables, type conflicts, etc. All these problems will be detected by the compiler when it processes the expanded code.

The main control structure for code generation consists of two steps:

- First, the `loop` prologue, the `loop` body, and the `loop` epilogue are constructed in the form of a `tagbody`[9] spe-

[8] The term *generalized Boolean* is part of the Common Lisp [1] terminology. It means any value where `nil` stands for *false* and any other value stands for *true*.

[9] For readers unfamiliar with Common Lisp, the `tagbody`

cial form.

- To the resulting `tagbody` form is then applied a set of nested *wrappers*, one for each clause. A wrapper for a clause typically contains `let` *bindings* required for the clause, but also iterator forms where such iterators are required by the clause type, for example `with-package-iterator`.

The `loop` body consists of three consecutive parts:

1. The *main* body, containing code for the `do` clauses and the accumulation clauses.
2. The *termination-test* part, containing code that checks whether iteration should terminate.
3. The *stepping* part, containing code that updates iteration variables in preparation for the next iteration.

For a small example of expanded code, consider the following `loop` form:

```
(loop for i from 2 to 20
      when (> i 10) do (print i))
```

It expands to[10] the following code:

```
(macrolet ((loop-finish ()
             '(go #:g956)))
  (block nil
    (let ((#:g957 2) (#:g958 20) (#:g959 1))
      (let ((#:g960 #:g957) (i #:g957))
        (tagbody
           (if (<= #:g960 #:g958)
               (incf #:g960 #:g959)
               (go #:g956))
         #:g963
           ;; main body
           (let ((#:g964 (> i 10)))
             (if #:g964
                 (print i)
                 (progn)))
           ;; termination test
           (unless (<= #:g960 #:g958)
             (go #:g956))
           ;; stepping
           (progn (setq i #:g960)
                  (incf #:g960 #:g959))
           (go #:g963)
         #:g956
           (return-from nil nil))))))
```

The essence of code generation is handled by a number of generic functions, each extracting different information from a clause:

- `accumulation-variables` extracts the accumulation variables of a clause, indicating also whether the `loop` keyword `into` is present.
- `declarations` extracts any declarations that result from the clause.
- `prologue-form` returns a form that should go in the `loop` prologue, or `nil` if no prologue form is required for the clause.
- `epilogue-form` returns a form that should go in the `loop` epilogue, or `nil` if no epilogue form is required for the clause.
- `termination-form` returns a form that should become a termination test, or `nil` if the clause does not result in a termination test.
- `step-form` returns a form that should be included in the stepping part of the `loop` body, for those clause types that define stepping. This generic function returns `nil` if the clause does not have any step forms associated with it.
- `body-form` returns a form that should be present in the main body of the expansion, or `nil` if the clause does not result in any form for the body.

The generic function `prologue-form` takes a clause argument and returns a form that should go in the `loop` prologue. The `initially` clause is an obvious candidate for such code. But the stepping clauses also have code that goes in the prologue, namely an initial termination test to determine whether any iterations at all should be executed.

Of the clause types defined by the Common Lisp standard, only the `finally` clause has a method that returns a value other than `nil` on the generic function `epilogue-form`.

The generic function `termination-form` takes a clause argument and returns a form for that clause that should go in the termination-test part of the body of the expanded code. Some of the `for/as` clauses and also the `repeat` clause have specialized methods on this generic function.

The generic function `step-form` takes a clause argument and returns a form for that clause that should go in the stepping part of the body of the expanded code. The `for/as` clauses and also the `repeat` clause have specialized methods on this generic function.

The generic function `body-form` takes a clause argument and returns a form for that clause that should go in the main body of the expanded code. The `do` and the accumulation clauses have specialized methods on this generic function.

3.5 Tests

Our code has been thoroughly tested. The code for testing contains almost 5000 lines. This code has been taken from Paul Dietz' ANSI test suite[11] and adapted to our needs. In particular, we had to remove some tests that did not conform to the standard, and we added tests where the test suite omitted to test potentially non-conforming behavior.

4. BENEFITS OF OUR METHOD

As already mentioned in Section 3.1, the main advantage of our technique is that it allows for a *modular* structure of the `loop` implementation.

MIT `loop` has extension capabilities as well, through the use of so-called *loop universes*. A loop universe is a structure instance that contains information on how to parse and translate all loop constructs. The ease with which an extension can be added depends on how well the extension fits

special form allows low-level constructs such as arbitrary control transfers to arbitrary statements through the use of labels and `go` forms that jump to such labels. This special form is mostly used in the expansion of high-level macros such as `loop` and other iteration constructs.

[10] We cleaned it up somewhat by removing unnecessary `progn` forms and we inserted the comments manually.

[11] See: https://gitlab.common-lisp.net/groups/ansi-test

into the framework provided. It is relatively easy to provide an additional keyword for a `for var being ...`, but it would be much more difficult to allow for a clause like `when form is-a type ...` to take but one example.

Furthermore, since MIT `loop` does not use generic functions, customizing existing behavior by extending existing methods for processing `loop` elements such as clauses is not an option.

The most immediate consequence of this improved modularity is that the code is easier to maintain than a monolithic code for the same purpose. A modification in one module is less likely to break other modules.

This modularity also makes it very simple for additional clause types to be added by the Common Lisp implementation, such as the extension for iterating over the user-extensible sequences described by Rhodes in his paper on user-extensible sequences [9]. This extension defines the new `loop` keywords `element` and `elements` for this purpose.

Furthermore, since the parsing technology we use does not require any costly pre-processing, extensions could be added by client code on a per-module basis, rather than as a permanent extension. Then, client code can maintain the capacity of detecting non-conforming constructs in most code, while allowing for selected extensions in specific modules.

As a consequence of this additional modularity, we think it is feasible to avoid the current problem of derivatives of MIT `loop`, namely that each implementation has had to introduce modifications to the single file that contains the code. With better modularity, we think it is possible to maintain the code for the `loop` macro as a separate entity, with each Common Lisp implementation supplying modifications in separate, small modules. Such a common code base would reduce the total maintenance cost for all Common Lisp implementations using this code base.

5. CONCLUSIONS AND FUTURE WORK

We have described a modern implementation of the Common Lisp `loop` macro. The main benefit of our method is better *modularity* compared to existing implementations, which makes maintenance easier, and also allows for more modular integration of client-defined extensions.

Our implementation contains significantly more code than, for instance, MIT `loop`; more than 5000 lines compared to 2000. There are several explanations for this discrepancy:

- Our code has more lines of comments; nearly 1500 compared to less than 200 for MIT `loop`.
- Our implementation contains more than 300 code lines dedicated to specific conditions and to condition reporters for those conditions, whereas MIT `loop` uses condition signaling very rarely, and then mostly using simple conditions with condition reporters in the form of literal strings.
- Our implementation is divided into nearly 50 files, or *modules*, and each new file represents some overhead in terms of code size.
- Our implementation contains more semantic verification as shown by the fact that it rejects the examples of non-conforming code shown in Section 2.1.
- Commonalities between clause types are captured as explicit class definitions which require additional code.
- We most likely have not identified all the instances where refactoring the code would be beneficial.

We believe that some ordinary code factoring will bring the difference in code size (not counting comments) down to a small difference that can be attributed to per-module overhead, and to the fact that we have more extensive semantic verification.

5.1 Use external parser framework

When we started the work on this library, we were unaware of any existing libraries for combinator parsing written in Common Lisp. Since then, we have been made aware of several libraries with such functionality, in particular:

- "cl-parser-combinators"[12] which is a library for combinator parsing inspired by Parsec [7]. Parsec was originally written in Haskell, and later re-implemented in other languages as well.
- "SMUG"[13] which seems to be more self contained than cl-parser-combinators, especially when it comes to the documentation.

We plan to evaluate cl-parser-combinators and SMUG to determine whether they provide the functionality required for parsing `loop` clauses, and if not, whether any of them can be extended to obtain this functionality.

A significant advantage of using one of these libraries over the existing technique is that they both have full support for the most general backtracking capabilities of combinator parsing. Using one of them rather than our current technique would make it unnecessary to consider careful ordering of clause parsers the way we currently need to do.

A possible disadvantage might be that full backtracking is potentially costly in terms of performance. However, we do not expect performance of clause parsers to be a determining factor for the overall performance of a Common Lisp compiler.

5.2 Second clause parser

As mentioned in Section 3, we are able to signal appropriate conditions in some cases when the initial attempt is made to parse the body of the `loop` form as individual clauses. However, when a syntax error is detected in some clause, all further analysis is abandoned. It would clearly be better if the analysis could continue with the remaining clauses, and if an appropriate error condition could be signaled for the faulty clause.

A simple way of improving error reporting would be to add more parsers for each clause type. These additional parsers would recognize incorrect clause syntax and ultimately result in an error being signaled, but more importantly, they would succeed so that parsing could continue with subsequent clauses. For example, a common mistake is to omit the `=` character in a `with` clause (See Appendix A.). A second parser would recognize this defective `with` clause as valid, but then either signal an appropriate error or issue a warning.

Unfortunately, however, while the parsing technique we use has many advantages as described in Section 4, it also

[12] https://github.com/Ramarren/cl-parser-combinators

[13] https://github.com/drewc/smug

has the main disadvantage that parsing gets slower as more parsers need to be tried, in particular if no care is taken to order the parsers with respect to probability of success.

We plan to avoid this conundrum by implementing a *second parser* for parsing individual clauses. This second parser would be invoked only when the first one fails. In that situation, we estimate that performance is of secondary importance and that emphasis should be on appropriate error signaling. Only this second parser would contain the additional clause parsers that recognize common incorrect clause syntax, and each such parser would be associated with an error or a warning appropriate for the situation. Existing clause parsers that recognize correct clause syntax would be re-used, and additional parsers for each clause type would be added to the set of parsers for that clause type.

At the moment, we have no quantifiable estimate of the cost of a fully backtracking parser, as compared to our simplified technique. At this point, we are also unable to quantify the additional performance penalty of having additional parsers for incorrect clause syntax when only correct `loop` syntax is parsed. Further work is required for a precise estimate.

5.3 Alternative parsing techniques

While a number of different parsing techniques such as LALR [5] or Pratt [8] would be feasible for the parsing the `loop` grammar as specified in the standard (See Appendix A.), an explicit goal of our work is to make it possible for client code to extend the grammar in a modular way. Most parsing techniques require the full grammar to be available a priori which is incompatible with this goal. There are parsing techniques other than combinator parsing that could be used, and we plan to investigate the feasibility of such techniques, in particular Earley [6] parsing.

5.4 Code refactoring

As suggested in the beginning of this section, there are very likely several remaining opportunities for code refactoring. Part of the plan for future work is to identify such opportunities and restructure the code accordingly, while respecting the existing modular structure of the code.

6. ACKNOWLEDGMENTS

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

APPENDIX

A. LOOP SYNTAX

In this appendix, we present some parts of the syntax of the `loop` macro that are relevant to the discussion in this paper. Parts that are not relevant to this paper have been left out.

```
loop [name-clause]
     {variable-clause}*
     {main-clause}*
=> result*

name-clause::= named name

variable-clause::= with-clause |
                   initial-final |
                   for-as-clause

with-clause::= with var1 [type-spec] [= form1]
               {and var2 [type-spec] [= form2]}*

main-clause::= unconditional |
               accumulation |
               conditional |
               termination-test |
               initial-final

initial-final::= initially compound-form+ |
                 finally compound-form+

unconditional::= {do | doing} compound-form+ |
                 return {form | it}

accumulation::= list-accumulation |
                numeric-accumulation

list-accumulation::= {collect | collecting |
                      append | appending |
                      nconc | nconcing}
                     {form | it}
                     [into simple-var]

numeric-accumulation::= {count | counting |
                         sum | summing | }
                        maximize | maximizing |
                        minimize | minimizing
                        {form | it}
                        [into simple-var]
                        [type-spec]

conditional::= {if | when | unless}
               form selectable-clause
               {and selectable-clause}*
               [else selectable-clause
               {and selectable-clause}*]
               [end]

selectable-clause::= unconditional |
                     accumulation |
                     conditional

termination-test::= while form |
                    until form |
                    repeat form |
                    always form |
                    never form |
                    thereis form

for-as-clause::= {for | as} for-as-subclause
                 {and for-as-subclause}*

for-as-subclause::= for-as-arithmetic |
                    for-as-in-list |
                    for-as-on-list |
                    for-as-equals-then |
                    for-as-across |
                    for-as-hash |
                    for-as-package
```

B. REFERENCES

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

[2] *ISO/IEC 9899:2011 Information Technology – Programming Language – C.* International Organization for Standardization, 2011.

[3] *ISO/IEC 14882:2014 Information Technology – Programming Language – C++.* International Organization for Standardization, 2014.

[4] G. Burke and D. Moon. Loop iteration macro. Technical Report MIT/LCS/TM-169, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, July 1980.

[5] F. L. DeRemer. *Practical Translators for LR(k) Languages.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, October 1969.

[6] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970.

[7] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, University of Utrecht, 2001.

[8] V. R. Pratt. Top down operator precedence. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 41–51, New York, NY, USA, 1973. ACM.

[9] C. 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.

[10] P. Wadler. How to replace failure by a list of successes. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

Source-to-Source Compilation via Submodules

Tero Hasu
BLDL and University of Bergen
tero@ii.uib.no

Matthew Flatt
PLT and University of Utah
mflatt@cs.utah.edu

ABSTRACT

Racket's macro system enables language extension and definition primarily for programs that are run on the Racket virtual machine, but macro facilities are also useful for implementing languages and compilers that target different platforms. Even when the core of a new language differs significantly from Racket's core, macros offer a maintainable approach to implementing a larger language by desugaring into the core. Users of the language gain the benefits of Racket's programming environment, its build management, and even its macro support (if macros are exposed to programmers of the new language), while Racket's syntax objects and submodules provide convenient mechanisms for recording and extracting program information for use by an external compiler. We illustrate this technique with Magnolisp, a programming language that runs within Racket for testing purposes, but that compiles to C++ (with no dependency on Racket) for deployment.

CCS Concepts

•Software and its engineering → Extensible languages; Translator writing systems and compiler generators;

Keywords

Language embedding, module systems, separate compilation

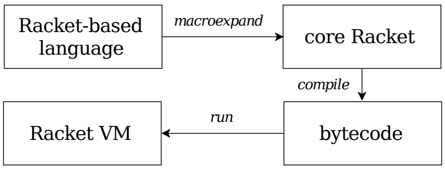
1. INTRODUCTION

A *macro expander* supports the extension of a programming language by translating extensions into a predefined core language. A *source-to-source compiler* (or *transcompiler* for short) is similar, in that it takes source code in one language and produces source code for another language. Since both macro expansion and source-to-source compilation entail translation between languages, and since individual translation steps can often be conveniently specified as macro transformations, a macro-enabled language can provide a convenient platform for implementing a transcompiler.

Racket's macro system, in particular, not only supports language extension—where the existing base language is enriched with new syntactic forms—but also language *definition*—where a completely new language is implemented though macros while hiding or adapting the syntactic forms of the base language. Racket's macro system is thus suitable for implementing a language with a different or constrained execution model relative to the core Racket language.

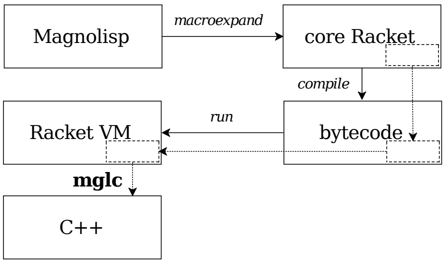
Magnolisp is a Racket-based language that targets embedded devices. Relative to Racket, Magnolisp is constrained in ways that make it more suitable for platforms with limited memory and processors. For deployment, the Magnolisp compiler transcompiles a core language to C++. For development, since cross-compilation and testing on embedded devices can be particularly time consuming, Magnolisp programs also run directly on the Racket virtual machine (VM) using libraries that simulate the target environment.

Racket-based languages normally target only the Racket VM, where macros expand to a core Racket language, core Racket is compiled into bytecode form, and then the bytecode form is run:



To instead transcompile a Racket-based language, Magnolisp could access the representation of a program after it has been macro-expanded to its core (via the `read` and `expand` functions). Fully expanding the program, however, would produce Racket's core language, instead of Magnolisp's core language. External expansion would also miss out on some strengths of the Racket environment, including automatic management of build dependencies.

Magnolisp demonstrates an alternative approach that takes full advantage of Racket mechanisms to assemble a "transcompile time" view of the program. The macros that implement Magnolisp arrange for a representation of the core program to be preserved in the Racket bytecode form of modules. That representation can be extracted as input to the `mglc` compiler to C++:



In this picture, the smaller boxes correspond to a core-form reconstruction that is only run in transcompile-time mode (as depicted by the longer arrow of the "run" step). The boxes are implemented as

9th ELS May 9–10, 2016, Kraków, Poland

submodules (Flatt 2013), and the core form is extracted by running the submodules instead of the main program modules.

By compiling a source program to one that constructs an AST for use by another compiler layer, our approach is similar to lightweight modular staging in Scala (Rompf and Odersky 2010) or strategies that exploit type classes in Haskell (Chakravarty et al. 2011). Magnolisp demonstrates how macros can achieve the same effect, but with the advantages of macros and submodules over type-directed overloading: more flexibility in defining the language syntax, support for static checking that is more precisely tailored to the language, and direct support for managing different instantiations of a program (i.e., direct evaluation versus transcompilation).

2. MAGNOLISP

Magnolisp[1] is statically typed, and all data types and function invocations are resolvable to specific implementations at compile time. Static typing for Magnolisp programs facilitates compilation to efficient C++, as the static types can be mapped directly to their C++ counterparts. To reduce syntactic clutter from annotations and to help retain untyped Racket's "look and feel," Magnolisp supports type inference à la Hindley-Milner.

Magnolisp's surface syntax is similar to Racket's for common constructs, but it also has language-specific constructs, including ones that do not directly map into Racket core language (e.g., `if-cxx` for conditional transcompilation). Magnolisp uses Racket's module system for managing bindings, both for run-time functions and for macros. An exported C++ interface is defined separately through `export` annotations on function definitions; only `export`ed functions are declared in the generated C++ header file.

A Magnolisp module starts with `#lang magnolisp`. The module's top-level can `define` functions, types, and so on. A function marked as `foreign` is assumed to be implemented in C++; it may also have a Racket implementation, given as the body expression, to allow it to be run in the Racket VM. Types are defined only in C++, so they are always `foreign`, and `typedef` can be used to give the corresponding Magnolisp declarations. The `type` annotation is used to specify types for functions and variables, and type expressions can refer to declared type names. The `#::` keyword is used to specify a set of annotations for a definition.

In the following example, `add` is a Magnolisp function of type `(-> Int Int Int)`, i.e., a binary function that computes with values of type `Int`. The `(rkt.+ x y)` expression in the function body is a call to a Racket function from the `racket/base` module to approximately simulate C++ `int`eger addition:

```
#lang magnolisp
(require magnolisp/std/list
         (prefix-in rkt. racket/base))

(typedef Int #:: ([foreign int]))
(define (add x y) ; integer primitive (implemented in C++)
  #:: (foreign [type (-> Int Int Int)])
  (rkt.+ x y))
```

No C++ code is generated for the above definitions, as they are both declared as `foreign`. As in Racket, it is possible to define macros; this pattern-based one defines a new conditional, which uses `magnolisp/std/list` module's `empty?` function:

```
(define-syntax-rule (if-empty lst thn els)
  (if (empty? lst) thn els))
```

For an example with a C++ translation, consider `sum-2`, a function that uses the above definitions to compute the sum of the first two elements of its list argument (or fewer for shorter lists):

```
(define (sum-2 lst) #:: (export)
  (if-empty lst 0
    (let ([t (tail lst)])
      (if-empty t (head lst)
        (add (head lst) (head t))))))
```

The transcompiler-generated C++ implementation for the `sum-2` function is the following (apart from minor reformatting):

```
MGL_API_FUNC int sum_2( List<int> const& lst ) {
  List<int> t;
  return is_empty(lst) ?
      0 :
      ((t = tail(lst)),
       (is_empty(t) ? head(lst) :
         add(head(lst), head(t))));
}
```

Figure 1 shows an overview of the Magnolisp architecture, including both the `magnolisp`-defined front end and the `mglc`-driven middle and back ends. Figure 2 illustrates the forms of data that flow through the compilation pipeline. Transcompilation triggers running of `"a.rkt"` module's transcompile-time code, through `magnolisp-s2s` submodule's instantiation by invoking `dynamic-require` to fetch values for certain variables (e.g., `def-lst`); the values describe the code of `"a.rkt"`, and are already in the compiler's internal data format. Any referenced dependencies of `"a.rkt"` (e.g., `"num-types.rkt"`, as indicated by `int`'s binding information) are processed in the same manner, and the relevant definitions are incorporated into the compilation result (i.e., `"a.cpp"` and `"a.hpp"`).

The middle and back ends are accessed either via the `mglc` command-line tool or via the underlying API as a Racket module. In either case, the expected input is a set of modules for transcompilation into C++. The compiler loads any transcompile-time code in the modules and their dependencies. Any module with a `magnolisp-s2s` submodule is assumed to be Magnolisp, but other Racket-based languages may also be used for macro programming or simulation. The Magnolisp compiler effectively ignores any code that is not run-time code in a Magnolisp module.

The program transformations performed by the compiler are generally expressed with term-rewriting strategies. These strategies are implemented by a custom combinator library[2] that is inspired by Stratego (Bravenboer et al. 2008). Syntax trees that are prepared for the transcompilation phase instantiate data types that support the primitive strategy combinators of the combinator library.

The compiler middle end implements whole-program optimization (by dropping unused definitions), type inference, and some simplifications (e.g., removal of condition checks where the condition is constant). The back end implements translation from Magnolisp core to C++ syntax (involving, e.g., lambda lifting), copy propagation, C++-compatible identifier renaming, splitting of code into sections (e.g.: public declarations, private declarations, and private implementations), and pretty printing.

3. TRANSLATED-LANGUAGE HOSTING

Magnolisp is an example of a general strategy for building a transcompiled language within Racket. In this section, we describe some details of that process for an arbitrary transcompiled language L. Where the distinction matters, we use L_R to denote a language that is intended to also run in the Racket VM (possibly with mock implementations of some primitives), and L_C to denote a language that only runs through compilation into a different language.

Building a language in Racket means defining a module or set of modules to implement the language. The language's modules

[1] Available from http://bldl.github.io/magnolisp-els16/

[2] http://bldl.github.io/illusyn/

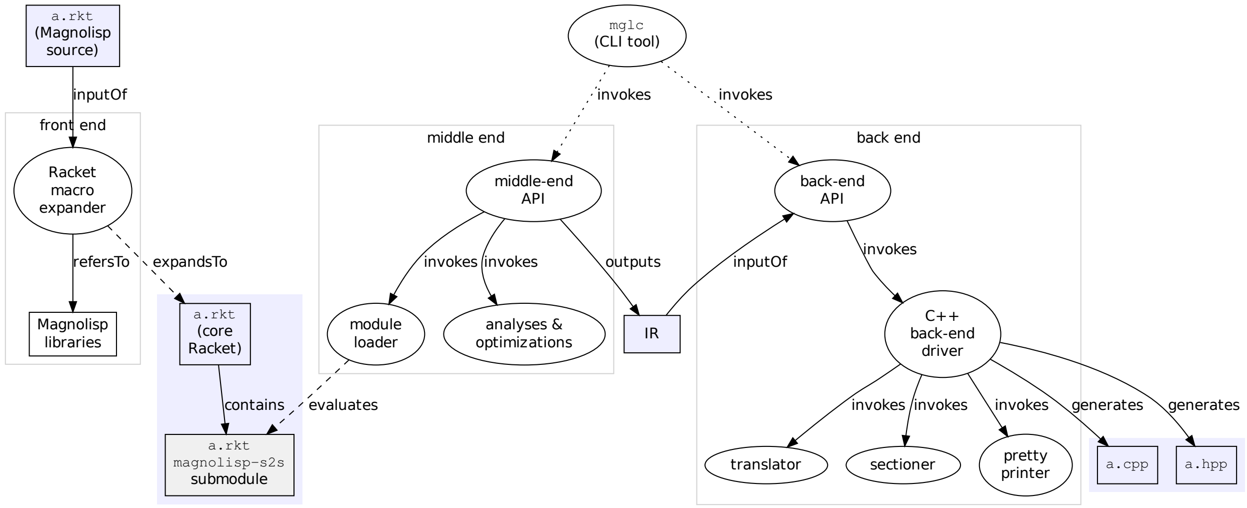
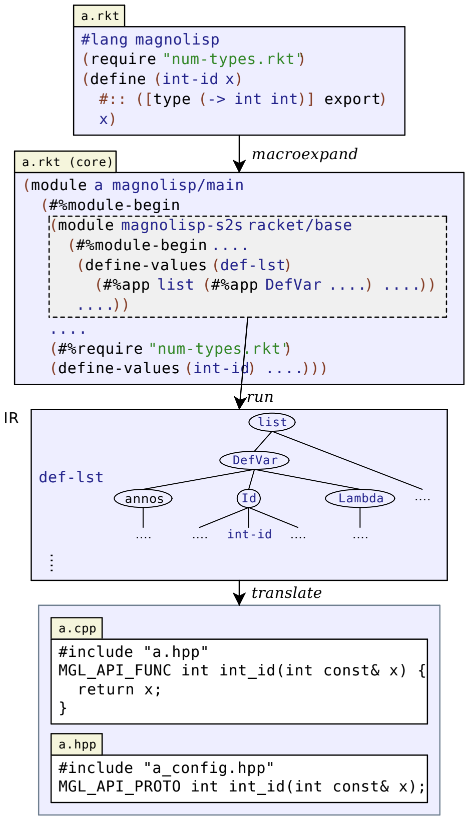


Figure 1: The overall architecture of the Magnolisp implementation, showing some of the components involved in compiling a Magnolisp source file "a.rkt" into a C++ implementation file "a.cpp" and a C++ header file "a.hpp". The dotted arrows indicate that the use of the mglc command-line tool is optional; the middle and back end APIs may also be invoked by other programs. The dashed "evaluates" arrow indicates a conditional connection between the left and right hand sides of the diagram; the magnolisp-s2s submodule is *only* loaded when transcompiling. The "expandsTo" connection is likewise conditional, as "a.rkt" may have been compiled to bytecode ahead of time, in which case the module is already available in a macro-expanded form; otherwise it is compiled on demand by Racket.



```
#lang magnolisp
(require "num-types.rkt")
(define (int-id x)
  #:: ([type (-> int int)] export)
  x)
```

```
(module a magnolisp/main
  (#%module-begin
   (module magnolisp-s2s racket/base
     (#%module-begin ....
      (define-values (def-lst)
        (#%app list (#%app DefVar ....) ....))
      ....))
   ....
   (#%require "num-types.rkt")
   (define-values (int-id) ....)))
```

```
#include "a.hpp"
MGL_API_FUNC int int_id(int const& x) {
  return x;
}
```

```
#include "a_config.hpp"
MGL_API_PROTO int int_id(int const& x);
```

Figure 2: Subset of Figure 1 showing file content: a Magnolisp module passing through the compilation pipeline.

define and export macros to compile the language's syntactic forms to core forms. In our strategy, furthermore, the expansion of the language's syntactic forms produces nested submodules to separate code than can be run directly in the Racket VM from information that is used to continue compilation to a different target.

3.1 Modules and #lang

All Racket code resides within some *module*, and each module starts with a declaration of its *language*. A module's language declaration has the form #lang *L* as the first line of the module. The remainder of the module can access only the syntactic forms and other bindings made available by the language *L*.

A language is itself implemented as a module.[3] In general, a language's module provides a *reader* that gets complete control over the module's text after the #lang line. A reader produces a *syntax object*, which is a kind of S-expression (that combines lists, symbols, etc.) that is enriched with source locations and other lexical context. We restrict our attention here to using the default reader, which parses module content directly as S-expressions, adding source locations and an initially empty lexical context.

For example, to start the implementation of *L* such that it uses the default reader, we might create a "main.rkt" module in an "L" directory, and add a reader submodule that points back to L/main as implementing the rest of *L*:

```
#lang racket
(module reader syntax/module-reader L/main)
```

The S-expression produced by a language's reader serves as input to the macro-expansion phase. A language's module provides syntactic forms and other bindings for use in the expansion phase by exporting macros and variables. A language *L* can re-export all of the bindings of some other language, in which case *L* acts as an

[3] Some language must be predefined, of course. For practical purposes, assume that the racket module is predefined.

extension of that language, or it can export an arbitrarily restrictive set of bindings.

A language must at least export a macro named `#%module-begin`, because that form implicitly wraps the body of a module. Most languages simply use `#%module-begin` from `racket`, which treats the module body as a sequence of `require` importing forms, `provide` exporting forms, definitions, expressions, and nested submodules, where a macro use in the module body can expand to any of the expected forms. A language might restrict the body of modules by either providing an alternative `#%module-begin` or by withholding other forms. A language might also provide a `#%module-begin` that explicitly expands all forms within the module body, and then applies constraints or collects information in terms of the core forms of the language.

As an example, the following `"main.rkt"` re-exports all of `racket` except `require` (and the related core language name `#%require`), which means that modules in the language *L* cannot import other modules. It also supplies an alternate `#%module-begin` macro to pre-process the module body in some way:

```
#lang racket
(module reader syntax/module-reader L/main)
(provide
  (except-out (all-from-out racket)
              require #%require #%module-begin)
  (rename-out [L-module-begin #%module-begin]))
(define-syntax L-module-begin ....)
```

For transcompilation, the `#%module-begin` macro plays a key role in our strategy. A Racket language *L* that is intended for transcompilation is defined as follows:

- *L*'s module exports bindings that define the language's surface syntax, and expand only to transcompiler-supported runtime forms. We describe this step further in section 3.2
- Macros record any additional metadata required for transcompilation. We describe this step further in section 3.3
- The `#%module-begin` macro expands all the macros in the module body. We describe this step further in section 3.4
- After full macro expansion, `#%module-begin` adds externally loadable information about the expanded module into the module. We describe this step further in section 3.5
- Any run-time support for running programs is provided alongside the macros that define the syntax of the language. We describe this step further in section 3.6

The export bindings of *L* may include variables, and the presence of transcompilation introduces some nuances into their meaning. When the meaning of a variable in *L* is defined in *L*, we say that it is a *non-primitive*. When its meaning is defined in the execution language, we say that it is a *primitive*. When the meaning of its appearances is defined by a compiler (or a macro) of *L*, we say that it is a *built-in*. As different execution targets may have different compilers, a built-in for one target may be a primitive for another.

3.2 Defining Surface Syntax

A module that implements the surface syntax of a language *L* exports a binding for each predefined entity of *L*, whether that entity is a built-in variable, a core-language construct, or a derived form. When the core language is a subset of Racket, derived forms obviously should expand to the subset. Where the core of *L* is a superset of Racket, additional constructs need an encoding in terms of Racket's core forms where the encoding is recognizable after expansion; possible encoding strategies include:

- **E1**. Use a variable binding to identify a core-language form. Use it in an application position to allow other forms to appear within the application form. Subexpressions within the form can be delayed with `lambda` wrappers, if necessary.
- **E2**. Attach information to a syntax object through its *syntax property* table; macros that manipulate syntax objects must then propagate properties correctly.
- **E3**. Store information about a form in a compile-time table that is external to the module's syntax objects.

A caveat for strategies **E2** and **E3** is that syntax properties and compile-time tables are transient, generally becoming unavailable after a module is fully expanded; any information to be preserved must be reflected as generated code in the module's expansion, as discussed in section 3.5. Another caveat of such "out-of-band" storage is that identifiers in the stored data must not be moved out of band too early; a binding form must be expanded before its references are moved so that each identifier properly refers to its binding.

In the case of L_R, the result of a macro-expansion should be compatible with both the transcompiler and the Racket evaluator. The necessary duality can be achieved if the surface syntax defining macros can adhere to these constraints: (**C1**) exclude Racket core form uses that are not supported by the compiler; (**C2**) add any compilation hints to Racket core forms in a way that does not affect evaluation (e.g., as custom syntax properties); and (**C3**) encode any compilation-specific syntax in terms of core forms that appear only in places where they do not affect Racket execution semantics.

Where constraints **C1**–**C3** cannot be satisfied, a fallback is to have `#%module-begin` rewrite the run- or transcompile-time code (or both) to make it conform to the expected core language. Rewriting may still be constrained by the presence of binding forms.

For cases where a language's forms do not map neatly to Racket binding constructs, Racket's macro API supports explicit *definition contexts* (Flatt et al. 2012), which enable the implementation of custom binding forms that cooperate with macro expansion.

For an example of foreign core form encoding strategy **E1**, consider an L_C with a `parallel` construct that evaluates two forms in parallel. This construct might be defined simply as a "dummy" constant, recognized by the transcompiler as a specific built-in by its identifier, translating any appearances of `(parallel e1 e2)` "function applications" appropriately:

```
(define parallel #f)
```

Alternatively, as an example of strategy **E2**, L_C's `(parallel e1 e2)` form might simply expand to `(list e1 e2)`, but with a `'parallel` syntax property on the `list` call to indicate that the argument expressions are intended to run in parallel:

```
(define-syntax (parallel stx)
  (syntax-case stx ()
    [(parallel e1 e2)
     (syntax-property #'(list e1 e2)
                      'parallel #t)]))
```

For L_R, `parallel` might instead be implemented as a simple pattern-based macro that wraps the two expressions in `lambda` and passes them to a `call-in-parallel` run-time function, again in accordance to strategy **E1**. The `call-in-parallel` variable could then be treated as a built-in by the transcompiler and implemented as a primitive for running in the Racket VM:

```
(define-syntax-rule (parallel e1 e2)
  (call-in-parallel (lambda () e1) (lambda () e2)))
```

For an example of adhering to constraint **C3**, we give a simplified definition of Magnolisp's typedef form. A declared type t is bound as a variable to allow Racket to resolve type references; these bindings also exist for evaluation as Racket, but they are never referenced at run time. The #%magnolisp built-in is used to encode the meaning of the variable, but as it has no useful definition in Racket, evaluation of any expressions involving it is prevented. The CORE macro is a convenience for wrapping (#%magnolisp ....) expressions in an (if #f .... #f) form to "short-circuit" the overall expression and make it obvious to the Racket bytecode optimizer that the enclosed expression is never evaluated. The annotate form is a macro that stores the annotations a ..., which might, for example, include t's C++ name.

```
(define #%magnolisp #f)
(define-syntax-rule (CORE kind arg ...)
  (if #f (#%magnolisp kind arg ...) #f))

(define-syntax-rule (typedef t #:: (a ...))
  (define t
    (annotate (a ...) (CORE 'foreign-type))))
```

3.3 Storing Metadata

A language implementation may involve *metadata* that describes a syntax object, but is not itself a core syntactic construct in the language. Such data may encode information (e.g., optimization hints) that is meaningful to a compiler or other kinds of external tools. Metadata might be collected automatically by the language infrastructure (e.g., source locations in Racket), it might be inferred by macros at expansion time, or it might be specified as explicit *annotations* in source code (e.g., Magnolisp functions' export).

Metadata differs from language constructs in that it does not tend to appear (or at least not remain) as a node of its own in a syntax tree. A workable strategy for retaining any necessary metadata is to have *L*'s syntactic forms store it during macro expansion. Encoding strategies **E1**–**E3** apply also for metadata, for which storage in syntax properties is a typical choice. Typed Racket, for example, stores its type annotations in a custom 'type-annotation syntax property (Tobin-Hochstadt et al. 2011).

Compile-time tables are another likely option for metadata storage. For storing data for a named definition, one might use an *identifier table*, which is a dictionary data structure where each entry is keyed by an identifier. An *identifier*, in turn, is a syntax object for a symbol. Such a table is suitable for both local and top-level bindings, because the syntax object's lexical context can distinguish different bindings that have the same symbolic name.

Recording metadata in compile-time state has the specific advantage of the data getting collated already during macro expansion which enables lookups across macro invocation sites, without any separate program analysis phase. One could, for example, keep track of variables annotated as #:mutable, perhaps to enforce legality of assignments already at macro-expansion time, or to declare immutable variables as const in C++ output:

```
(define-for-syntax mutables (make-free-id-table))

(define-syntax (my-define stx)
  (syntax-case stx ()
    [(_ x v)
     #'(define x v)]
    [(_ #:mut x v)
     (free-id-table-set! mutables #'x #t)
     #'(define x v)]))
```

It is also possible to encode annotations in the syntax tree proper, which has the advantage of fully subjecting annotations to macro expansion. Magnolisp adopts this approach for its annotation recording, using a special 'annotate-property-flagged let-values form to contain annotations. Each contained annotation expression a (e.g., [type ....]) has its Racket evaluation prevented by encoding it as a Magnolisp CORE form:

```
(define-syntax-rule (type t) (CORE 'anno 'type t))

(define-syntax (annotate stx)
  (syntax-case stx ()
    [(_ (a ...) e)
     (syntax-property
      (syntax/loc stx ; retain stx's source location
        (let-values ([() (begin a (values))] ...)
          e))
      'annotate #t)]))
```

The annotate-generated let-values forms introduce no bindings, and their right-hand-side expressions yield no values; only the expressions themselves matter. Where the annotated expression e is an initializer expression, the Magnolisp compiler decides which of the annotations to actually associate with the initialized variable.

3.4 Expanding Macros

One benefit of reusing the Racket macro system with *L* is to avoid having to implement an *L*-specific macro system. When the Racket macro expander takes care of macro expansion, the remaining transcompilation pipeline only needs to understand *L*'s core syntax (and any related metadata). Racket includes two features that make it possible to expand all the macros in a module body, and afterwards process the resulting syntax, all within the language.

The first of these features is the #%module-begin macro, which can transform the entire body of a module. The second is the local-expand (Flatt et al. 2012) function, which may be used to fully expand all the #%module-begin sub-forms.

The local-expand operation also supports *partial* sub-form expansion, as it takes a "stop list" of identifiers that prevent descending into sub-expressions with a listed name. At first glance, one might imagine exploiting this feature to allow foreign core syntax to appear in a syntax tree, and simply prevent Racket from proceeding into such forms. That strategy would mean, however, that foreign binding forms would not be accounted for in Racket's binding resolution. It would also be a problem if foreign syntactic forms could include Racket syntax sub-forms, as such sub-forms would need to be expanded along with enclosing binding forms.

3.5 Exporting Information to External Tools

After the #%module-begin macro has fully expanded the content of a module, it can gather information about the expanded content to make it available for transcompilation. The gathered information can be turned into an expression that reconstructs the information, and that expression can be added to the overall module body that is produced by #%module-begin.

The information-reconstructing expression should *not* be added to the module as a run-time expression, because extracting the information for transcompilation would then require running the program (in the Racket VM). Instead, the information is better added as compile-time code. The compile-time code is then available from the module while compiling other *L* modules, which might require extra compile-time information about a module that is imported into another *L* module. More generally, the information can be extracted by running only the compile-time portions of the module, instead of running the module normally.

As a further generalization of the compile versus run time split, the information can be placed into a separate *submodule* within the module. A submodule can have a dynamic extent (i.e., run time) that is unrelated to the dynamic extent of its enclosing module, and its bytecode may even be loaded separately from that of the enclosing module. As long as a compile-time connection is acceptable, a submodule can include syntax-quoted data that refers to bindings in the enclosing module, so that information can be easily correlated with bindings that are exported from the module.

For example, suppose that L implements definitions by producing a normal Racket definition for running within the Racket virtual machine, but it also needs a syntax-quoted version of the expanded definition to compile to a different target. The `module+` form can be used to incrementally build up a `to-compile` submodule that houses definitions of the syntax-quoted expressions:

```
(define-syntax (L-define stx)
  (syntax-case stx ()
   [(L-define id rhs)
    (with-syntax ([rhs2 (local-expand #'rhs
                         'expression null)])
     #'(begin
        (define id rhs2)
        (begin-for-syntax
          (module+ to-compile
            (define id #'rhs2)))))]))
```

The `begin-for-syntax` wrapping makes the `to-compile` submodule reside at compilation time relative to the enclosing module, so that loading the submodule will not run the enclosing module. Within `to-compile`, the expanded right-hand side is quoted as syntax using `#'`. Syntax-quoted code is often a good choice of representation for code to be compiled again to a different target language, because lexical-binding information is preserved in a syntax quote. Source locations are also preserved, so that a compiler can report errors or warnings in terms of a form's original location (`mglc` fetches original source text based on location).

Another natural representation choice is to use any custom intermediate representation (IR) of the compiler. Magnolisp, for example, processes Racket syntax trees already during macro expansion, turning them into its IR format, which also incorporates metadata. The IR uses Racket `struct` instances to represent AST nodes, while still retaining some of the original Racket syntax objects as metadata, for purposes of transcompile-time reporting of semantic errors. Magnolisp programs are parsed at least twice, first from text to Racket syntax objects by the reader, and then from syntax objects to the IR by `#%module-begin`; additionally, any macros effectively parse syntax objects to syntax objects. As parsing is completed already in `#%module-begin`, any Magnolisp syntax errors are discovered even when just evaluating programs as Racket.

The `#%module-begin` macro of `magnolisp` exports the IR via a submodule named `magnolisp-s2s`. The submodule contains an expression that reconstructs the IR, albeit in a somewhat lossy way, excluding details that are irrelevant for compilation. The IR is accompanied by a table of identifier binding information indexed by module-locally unique symbols, which the transcompiler uses for cross-module resolution of top-level bindings, to reconstruct the identifier binding relationships that would have been preserved by Racket if exported as syntax-quoted code. As `magnolisp-s2s` submodules do not refer to the bindings of the enclosing module, they are loadable independently from it.

3.6 Run-Time Support

The modules that implement a Racket language can also define run-time support for executing programs. For L, such support may be required for the compilation target environment; for L_R, any support would also be required for the Racket VM. Run-time support for L is required when L exports bindings to run-time variables, or when the macro expansion of L can produce code referring to run-time variables (even if such a variable's run-time existence is very limited, as it is for `#%magnolisp`).

Every run-time variable requires a run-time binding, to make it possible for Racket to resolve references to them. When binding built-ins and primitives of L_C, any initial value expression can be given, as the expressions are not evaluated. A literal constant expression is a suitable initializer for built-ins of L_R, which are initialized for Racket VM execution, but generally never referenced.

Each non-primitive is—by definition—implemented in L, with a single definition applicable for all targets. Strictly speaking, though, any non-primitive that is *exported* by a Racket module L *cannot* itself be implemented in L, but must use a smaller language; the Racket module system does not allow cyclic dependencies.

Defining a primitive of L involves specifying a translation for appearances of the variable into any target language. For a Racket VM target, the variable's value must specify its meaning. For other targets, it may be most convenient to specify the target language mapping in L, assuming that L includes specific language for that purpose. As the mappings are only needed during transcompilation, any metadata specifying them might be placed into a module that is only loaded on demand by the compiler.

The `magnolisp` language, for example, binds three run-time variables, all of which are built-ins. Of these, `#%magnolisp` is only used for its binding, and only during macro expansion. The compiler knows that conditional expressions must always be of type `Bool`, and that `Void` is *the* unit type of the language; this knowledge is useful during type checking and optimization. References to the Magnolisp built-ins may appear in code generated by `magnolisp`'s macros, and hence they must already be bound for the language implementation. Their metadata (specifying C++ translations) is not required by the macros, however, which makes it possible to `declare` that information separately, using Magnolisp's own syntax for storing metadata for an existing binding:

```
#lang magnolisp/base
(require "core.rkt" "surface.rkt")
(declare #:type Bool #:: ([foreign bool]))
(declare #:type Void #:: ([foreign void]))
```

4. EVALUATION

Our Racket-hosted transcompilation approach is generic—in theory capable of accommodating a large class of languages. In practice, we imagine that it is most useful for hosting newly developed languages (such as Magnolisp), where design choices can achieve a high degree of reuse of the Racket infrastructure. In particular, Racket's support for creating new, extensible languages could be a substantial motivation to follow our approach. Racket hosting is particularly appropriate for an evolving language, since macros facilitate quick experimentation with language features.

Another potential use of our strategy is to add transcompilation support for an existing Racket-based language. We have done so for Erda[4], creating $Erda_{C++}$ as its C++-translatable variant. Erda has Racket-like syntax, but its evaluation differs significantly from both Racket and Magnolisp. Erda_{C++} programs nonetheless compile to C++ using an unmodified Magnolisp compiler.

Erda_{C++} illustrates that Magnolisp is not only a language, but also infrastructure for making Racket-based languages translatable into C++. A Magnolisp-based language must be transformable into Magnolisp's core language, which is more limited than that of Racket (lacking first-class functions, escaping closures, etc.), but the language can have its own runtime libraries (whose names must be `magnolisp-s2s`-communicated to `mglc`). The Racket API of Magnolisp includes a `make-module-begin` function that makes

[4] http://bldl.github.io/erda/

it convenient for other languages to implement mglc-compatible #%module-begin macros—ones that communicate all the expected information.

A potential drawback of transcompilation is the disconnect between the original, unexpanded code and its corresponding generated source code, which can lead to difficulties in debugging. The problem is made worse by macros, and it can be particularly pressing when the output is hard for humans to read. As Racket's macro expansion preserves source locations, a transcompiler could at least emit the original locations via #line directives (as in C++) or source maps (as supported by some JavaScript environments).

4.1 Language Design Constraints

In our experience, two design constraints make Racket reuse especially effective: the hosted language's name resolution should be compatible with Racket's, and its syntax should use S-expressions.

Overloading as a language feature, for instance, appears a bad fit for Racket's name resolution. Instead of overloading, names in Racket programs are typically prefixed with a datatype name, as in string-length and vector-length. Constructs for renaming at module boundaries, such as prefix-in and prefix-out, help implement and manage name-prefixing conventions.

An S-expression syntax is not strictly necessary, but Racket's macro programming APIs work especially well with its default parsing machinery. The language implementor can then essentially use concrete syntax in patterns and templates for matching and generating code. This machinery is comparable to concrete-syntax support in program transformation toolkits such as Rascal (Klint et al. 2009) and Spoofax (Kats and Visser 2010). Still, other kinds of concrete syntaxes can be adopted for Racket languages, with or without support for expressing macro patterns in terms of concrete syntax, as demonstrated by implementations of Honu (Rafkind and Flatt 2012) and Python (Ramos and Leitão 2014).

5. RELATED WORK

Many language implementations run on Lisp dialects and also target other environments. Some languages, such as Linj (2013) or Clojure plus ClojureScript (2016), primarily provide a Lisp-like language in the target environment. Other languages, such as STELLA (Chalupsky and MacGregor 1999) and Parenscript (2016), primarily match the target environment's semantics but enable execution in a Lisp as well. Magnolisp is closer to the latter group, in that it primarily targets the target environment's semantics.

Most other languages previously implemented on Racket have been meant for execution only on the Racket virtual machine, but a notable exception is Dracula (Eastlund 2012), which compiles macro-expanded programs to ACL2. Its (so far largely undocumented) compilation strategy is to expand syntactic forms to a subset of Racket's core forms, and to specially recognize applications of certain functions (such as make-generic) for compilation to ACL2. The part of a Dracula program that runs in Racket is expanded normally, while the part to be translated to ACL2 is recorded in a submodule through a combination of structures and syntax objects, where binding information in syntax objects helps guide the translation.

Whalesong (Yoo and Krishnamurthi 2013) and Pycket (Bauman et al. 2015) are both implementations *of* Racket targeting foreign language environments. Their approaches to acquiring fully macro-expanded Racket core language differ from ours. Whalesong compiles to JavaScript via Racket bytecode, which is optimized for efficient execution (e.g., through inlining), but does not retain all of the original (core) syntax; thus, it is not the most semantics-rich starting point for translation into foreign languages. The Pycket compiler instead performs external expansion to get core Racket; it reads, expands, and JSON-serializes Racket syntax, in order to pass it over to the RPython meta-tracing framework.

Ziggurat (Fisher and Shivers 2008)—also built on Racket (then PLT Scheme)—is a meta-language system for implementing extensible languages. Its approach allows both for self-extension and transcompilation of languages, with different tradeoffs compared to ours. Ziggurat features hygienic macros that are Scheme-like, but have access to static semantics, as defined for a language through other provided mechanisms; Racket lacks specific support for interleaving macro expansion with custom analysis. Ziggurat's macros may be locally scoped, but not organized into separately loadable modules; Racket allows for both. There is basic safety of macro composition with respect to Ziggurat's own name resolution, but composability of custom static semantics depends on their implementation. Ziggurat includes constructs for defining new syntax object types, while our approach requires encoding "tricks."

Lightweight Modular Staging (LMS) (Rompf and Odersky 2010) is similar to our technique in goals and overall strategy, but leveraging Scala's type system and overload resolution instead of a macro system. With LMS, a programmer writes expressions that resemble Scala expressions, but the type expectations of surrounding code cause the expressions to be interpreted as AST constructions instead of expressions to evaluate. The constructed ASTs can then be compiled to C++, CUDA, JavaScript, other foreign targets, or to Scala after optimization. AST constructions with LMS benefit from the same type-checking infrastructure as normal expressions, so a language implemented with LMS gains the benefit of static typing in much the same way that a Racket-based language can gain macro extensibility. LMS has been used for languages with application to machine learning (Sujeeth et al. 2011), linear transformations (Ofenbeck et al. 2013), fast linear algebra and other data structure optimizations (Rompf et al. 2012), and more.

The Accelerate framework (Chakravarty et al. 2011; McDonell et al. 2013) is similar to LMS, but in Haskell with type classes and overloading. As with LMS, Accelerate programmers benefit from the use of higher-order features in Haskell to construct a program for a low-level target language with only first-order abstractions.

The Terra programming language (DeVito et al. 2013) takes an approach similar to ours, as it adopts an existing language (Lua) for compile-time manipulation of constructs in the run-time language (Terra). Like Racket, Terra allows compile-time code to refer to run-time names in a way that respects lexical scope. Terra is not designed to support transcompilation, and it compiles to binaries via Terra as a fixed core language. Another difference is Terra's emphasis on supporting code generation at run time, while our emphasis is on separation of compile and run times.

CGen (Selgrad et al. 2014) is a reformulation of C with an S-expression-based syntax, integrated into Common Lisp. An AST for source-to-source compilation is produced by evaluating the core forms of CGen; this differs from our approach, where run-time Racket core forms are not evaluated. Common Lisp's defmacro construct is available to CGen programs for defining language extensions; Racket's lexical-scope-respecting macros compose in a more robust manner. Racket's macro expansion also tracks source locations, which would be a useful feature for a CGen-like tool. CGen uses the Common Lisp package system to implement support for locally and explicitly switching between CGen and Lisp binding contexts, so that ambiguous names are shadowed; Racket does not include a similar facility, although approximations thereof should be implementable within Racket.

SC (Hiraishi et al. 2007) is another reformulation of C with an S-expression-based syntax. It supports language extensions defined by transformation rules written in a separate, Common Lisp based domain-specific language (DSL). The rules treat SC programs as data, and thus SC code is not subject to Lisp macro expansion (as in our solution) or Lisp evaluation (as in CGen). Fully transformed programs (in the base SC-0 language) are compiled to C source code. SC programs themselves have access to a C-preprocessor-style extension mechanism via which there is limited access to Common Lisp macro functionality.

6. CONCLUSION

We have described a generic approach for having Racket host the front end of a source-to-source compiler. It involves a proper embedding of the hosted language into Racket, so that Racket's usual language definition facilities are exploited rather than bypassed. Notably, the macro and module systems are still available and, if exposed to the hosted language, provide a way to implement and manage language extensions within the language. Furthermore, tools such as the DrRacket IDE work with the hosted language, recognize the binding structure of programs written in the language, and can usually trace the origins of macro-transformed code.

Among the various ways to arrange for a source-to-source compiler to gain access to information about a program, our approach is most appropriate when the language's macros target a specific foreign core language and runtime library and when it is useful to avoid "extra-linguistic mechanisms" (Felleisen et al. 2015) by having the language itself communicate its execution requirements to the outside world. Such communications may be prepared as submodules, which can also contain an AST in the appropriate core language and representation, allowing one source language to support multiple different targets. Racket's separate compilation and build management help limit preparation work to modules whose source files or dependencies have changed.

Racket's macro system is expressive enough that the syntax and semantics of a variety of language constructs can be specified in a robust way. Given that typical macros compose safely, and given that hygiene reduces the likelihood of name clashes and allows macros to be defined privately, pervasive use of syntactic abstraction becomes a realistic alternative to manual or tools-assisted writing of repetitive code. Such abstraction can benefit both the codebase implementing a Racket-based language, as well as programs written in a macro-enabled Racket-based language.

Acknowledgements Carl Eastlund provided information about the implementation of Dracula. Magne Haveraaen, Anya Helene Bagge, and anonymous referees provided useful comments on drafts of this paper. This research has in part been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language.

Bibliography

Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: A Tracing JIT For a Functional Language. In *Proc. ACM Intl. Conf. Functional Programming*, 2015.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming* 72(1-2), pp. 52–70, 2008.

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proc. Wksp. Declarative Aspects of Multicore Programming*, 2011.

Hans Chalupsky and Robert M. MacGregor. STELLA - a Lisp-like language for symbolic programming with delivery in Common Lisp, C++ and Java. In *Proc. Lisp User Group Meeting*, 1999.

ClojureScript. 2016. https://github.com/clojure/clojurescript

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A Multi-Stage Language for High-Performance Computing. *ACM SIGPLAN Notices* 48(6), pp. 105–116, 2013.

Carl Eastlund. Modular Proof Development in ACL2. PhD dissertation, Northeastern University, 2012.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *Proc. Summit Advances in Programming Languages*, 2015.

David Fisher and Olin Shivers. Building Language Towers with Ziggurat. *J. Functional Programming* 18(5-6), pp. 707–780, 2008.

Matthew Flatt. Submodules in Racket: You Want it *When*, Again? In *Proc. Generative Programming and Component Engineering*, 2013.

Matthew Flatt, Ryan Culpepper, Robert Bruce Findler, and David Darais. Macros that Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. *J. Functional Programming* 22(2), pp. 181–216, 2012.

Tasuku Hiraishi, Masahiro Yasugi, and Taiichi Yuasa. Experience with SC: Transformation-based Implementation of Various Language Extensions to C. In *Proc. Intl. Lisp Conference*, pp. 103–113, 2007.

Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 444–463, 2010.

Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proc. IEEE Intl. Working Conf. Source Code Analysis and Manipulation*, pp. 168–177, 2009.

Linj. 2013. https://github.com/xach/linj

Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs. In *Proc. ACM Intl. Conf. Functional Programming*, 2013.

Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *Proc. Generative Programming and Component Engineering*, 2013.

Parenscript. 2016. https://common-lisp.net/project/parenscript/

Jon Rafkind and Matthew Flatt. Honu: Syntactic Extension for Algebraic Notation Through Enforestation. In *Proc. Generative Programming and Component Engineering*, pp. 122–131, 2012.

Pedro Ramos and António Menezes Leitão. An Implementation of Python for Racket. In *Proc. European Lisp Symposium*, 2014.

Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proc. Generative Programming and Component Engineering*, pp. 127–136, 2010.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing Data Structures in High-Level Programs: New Directions for Extensible Compilers Based on Staging. In *Proc. ACM Sym. Principles of Programming Languages*, 2012.

Kai Selgrad, Alexander Lier, Markus Wittmann, Daniel Lohmann, and Marc Stamminger. Defmacro for C: Lightweight, Ad Hoc Code Generation. In *Proc. European Lisp Symposium*, 2014.

Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proc. Intl. Conf. Machine Learning*, 2011.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. *SIGPLAN Not.* 47(6), pp. 132–141, 2011.

Danny Yoo and Shriram Krishnamurthi. Whalesong: Running Racket in the Browser. In *Proc. Dynamic Languages Symposium*, 2013.

Extending Software Transactional Memory in Clojure with Side-Effects and Transaction Control

Søren Kejser Jensen
Department of Computer Science
Aalborg University, Denmark
skj@cs.aau.dk

Lone Leth Thomsen
Department of Computer Science
Aalborg University, Denmark
lone@cs.aau.dk

ABSTRACT

In conjunction with the increase of multi-core processors the use of functional programming languages has increased in recent years. The functional language Clojure has concurrency as a core feature, and provides Software Transactional Memory (STM) as a substitute for locks. Transactions in Clojure do however not support execution of code with side-effects and provide no methods for synchronising threads. Additional concurrency mechanisms on top of STM are needed for these use cases, increasing the complexity of the code. We present multiple constructs allowing functions with side-effects to be executed as part of a transaction. These constructs are made accessible through a uniform interface, making them interchangeable. We present constructs providing explicit control of transactions. Through these constructs transactions can synchronise threads based on the current program state, abort a transaction, and provide an alternative path of execution if a transaction is forced to restart. eClojure is the implementation of these constructs in Clojure 1.8, and it is tested using an additional set of unit tests. With a series of use cases we show that the addition of these constructs provides additional capabilities to Clojure's STM implementation without violating its ACI properties. Through a usability evaluation with a small sample size we show that using these constructs reduces both the number of lines of code and the development time compared to Clojure 1.8. Last we demonstrate through a preliminary performance benchmark that the addition of these constructs adds only a constant overhead to the STM implementation.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.4.1 [**Operating Systems**]: Process Management—*Concurrency; Synchronization; Threads*

Keywords

Clojure; MVCC; STM; Side-Effects; Transaction Control

9th ELS May 9–10, 2016, Kraków, Poland

1. INTRODUCTION

In recent years processors have gained more computational threads instead of an increase in clock frequency [17]. This has increased the complexity of developing software that utilises the full computational power of these multi-core processors due to the need for managing multiple threads of execution with non-deterministic interleaving [12]. To ease development of multi-threaded software, multiple concurrency methods which provide an abstraction on top of threads and locks have been proposed. One of these is STM where a critical section is executed as a transaction known from database management systems, alleviating the need for locks. Clojure by default provides immutable data structures making it safe to share data between threads without any synchronisation, and implements STM for synchronising access to mutable data between multiple threads [11].

While the STM implementation in Clojure simplifies synchronisation of access to shared mutable data, it only has limited support for synchronising Lisp forms with side-effects. As STM transactions can restart a mechanism must be provided so side-effects are not repeated, while still allowing the transaction itself to restart. Additionally, Clojure does not provide any method for synchronising threads, but relies on its interop with Java to provide constructs such as `Barriers` and `Semaphores`. The use of Java's concurrency API can add unnecessary complexity to a Clojure program, due to the API being developed in an imperative object oriented style. Examples will be presented in Section 3.

Based on these shortcomings of the STM implementation in Clojure 1.8, we make the following contributions:

- A uniform interface to methods for synchronising forms with side-effects in multiple transactions in parallel.
- A `ref` type for storing a reference to a Java object and a set of functions for executing methods on the object.
- Multiple novel extensions that enhance the capability of an existing method for controlling transactions [8].
- The eClojure implementation with a set of unit tests.[1]
- An initial evaluation of eClojure in regards to functionality, usability and added performance overhead.

To our knowledge this is the first interface to multiple methods for executing forms with side-effects in a transaction that has been implemented in a Lisp based programming language. While similar transaction control constructs have

[1] https://github.com/skejserjensen/eclojure

been implemented in the Common Lisp STM implementation STMX [6], we provide extensions to these constructs that allow for additional functionality and more readable code due to less reliance on implicit behaviour. Last, Clojure's Lisp heritage and dynamic type system allow for a high degree of expressiveness. This makes it impossible for the runtime to determine if a form contains data shared between threads or what side-effects it will perform, a problem that is minimised by a strong static type system [8, 4].

This paper is structured as follows; related work is summarized in Section 2. A short introduction to the STM implementation in Clojure followed by an in-depth discussion of its limitation is presented in 3. The functionality added in eClojure is described in Section 4, followed by a preliminary evaluation of the extensions and a discussion of the results in Section 5. We present our conclusion in Section 6, and we describe possible future work in Section 7.

2. RELATED WORK

An analysis of the problems with STM executing functions with side-effects was presented by Baugh and Zilles [2]. The following naming conventions and descriptions are based on their summary, with the exception of irrevocable as it in general is the term used in the literature for that method.

Outlaw The combination of STM and side-effects is not allowed, enforced as a recommendation as in Clojure, or forced as Haskell does using its type system [8].

Defer The transaction waits until it is guaranteed to commit before the side-effects are performed. This adds complexity as the execution no longer is sequential.

Irrevocable Guarantees that a transaction does not restart, i.e. the use of side-effects is the same as outside transactions. The complexity of irrevocable comes from executing multiple irrevocable transactions in parallel.

Compensate If a transaction restarts after a side-effect has occurred, code to compensate for said side-effect is executed. Not all side-effects can be compensated for, e.g. a message printed to the screen.

In general two approaches have been proposed for combining these methods, a generic interface or a set of system calls specific for STM. McDonald et al. [15] propose the use of a commit, a violation, and an abort handle, each allowing a function to be executed at that phase of the transaction. The implementation uses Hardware Transactional Memory (HTM) making it less portable than an STM based approach. The other approach is used by Volos et al. [19] to develop an API proving OS functionality such as file management, threading and network communication in transactions. Demsky and Tehrany [5] present a method for integrating file operations into STM using a defer based method, where transactions can operate on the same file as operations on each file are buffered until the transaction commits.

Other uses for the methods have also been proposed. In addition to displaying how irrevocable can be used for executing side-effects, Welc et al. [20] show the benefits of having irrevocable transactions as part of an STM implementation. They present benchmarks showing that hash tables performing the rehashing step in an irrevocable transaction increase performance, as the rehashing step cannot restart.

Harris et al. [8] present an STM implementation in Haskell which allows developers to control transactions through the function `retry`, allowing the transaction to restart and block. `orElse` is used for composing alternative transactions and execute the first able to commit. Harries and Jones [7] extended this implementation with the function `check`. `check` allows developers to specify an invariance that must hold for all transactions. Bieniusa et. al. [3] extend Haskell's STM implementation with a pre-commit step, splitting the commit process into `stm_prepare` and `stm_finalize`. Code can be executed between these stages, providing developers with the means to remove conflicts with other transactions and then allowing a transaction to commit, in addition to execution of code with side-effects.

The constructs `retry` and `orElse` have been reused in several other STM implementations. Bronson et al. [4] propose an implementation of STM for Scala that included the constructs, while Ghilardi [6] provides an implementation in Common Lisp through the library STMX which combines STM with HTM to increase performance.

3. CLOJURE STM

As Clojure's use of the term retry overlaps with its existing use in the literature [8], the following terms are used:

Abort the transaction stops and discards all changes.

Terminate the transaction aborts and execution continues as if the transaction succeeded.

Restart the transaction aborts and starts execution of the transaction from the start.

Retry the transaction restarts, but execution is blocked at the start of the transaction.

3.1 Interface

The STM implementation in Clojure is based on Multi Version Concurrency Control (MVCC). MVCC provides each transaction with a snapshot of all mutable values from when the transaction started executing. As each transaction has a separate snapshot of the mutable values, a write by one transaction does not require another transaction reading the same value in parallel to restart. Use of MVCC does not prevent a transaction from restarting in general as multiple writes executed in parallel must be synchronised.

```
(dosync & exprs)

(ref x)
(ref x & options)

(deref ref)

(ref-set ref val)
(alter ref fun & args)
(commute ref fun & args)

(ensure ref)
(io! & body)
```

Listing 1: Interface for use of STM

The interface for the STM implementation in Clojure is shown in Listing 1. A transaction is created by the function `dosync` which takes forms as argument and executes them in a transaction. For a transaction to synchronise reading and writing through MVCC a set of metadata must be attributed to a mutable value. In Clojure this is represented by the type `Ref`, that combines a mutable value with the necessary metadata. A value stored in a `Ref` can be read using `deref` and updated inside a transaction by one of three functions `ref-set`, `alter` or `commute` depending on how it should be updated. `ref-set` writes a new value to the `Ref` overwriting the existing value. `alter` updates the existing value by executing a function passed as the argument `fun`, the function must have at least one parameter as the value of the `Ref` is given as the first argument and the returned value is written to the `Ref`. `commute` provides the same functionality as `alter` but assumes that the update to the `Ref` is commutative, i.e. can be performed by transactions in parallel.

Using MVCC, writing and reading of a `Ref` can be performed in parallel. However if a computation depends on multiple `Ref`s being kept in a consistent state but only read by the transaction, the function `ensure` can be used to ensure that the `Ref`s are not updated by another transaction until the transaction calling `ensure` has ended. Last, in contrary to Haskell, Clojure can not use types to represent functions with effects. `io!` provides a means for annotating forms that are incompatible with transactions, forcing an exception to be thrown if the form is executed by a transaction.

```
(agent state & options)
(send a f & args)
(await & agents)
```

Listing 2: Interface for use of agents

As forms executed by a transaction cannot contain side-effects, interoperability with another concurrency construct, `Agent`, is provided. `Agent`s are superficially similar to actors as they encapsulate state and update this state in a separate thread based on a sequence of instructions it receives. However an actor implements its behaviour internally and receives messages about which operation to perform, but `Agent`s receive functions which are then executed with the `Agent`'s current state as input and the returned value of the function written as the `Agent`'s new state.

Clojure's interface for working with `Agent` can be seen in Listing 2. `agent` creates a new `Agent` with an initial state. `send` sends a function to an `Agent`, the function takes the `Agent`'s current state as argument and its return value is set as the `Agent`'s new state. A thread executing `await` with a sequence of `Agent`s as argument is blocked until all functions received by the `Agent`s are executed. `Agent` provides a limited means for transactions to execute arbitrary side-effects as calls to `send` are deferred until a transaction commits.

```
(add-watch reference key fn)
(remove-watch reference key)

(set-validator! iref validator-fn)
(get-validator iref)
```

Listing 3: Interface for validators and watchers

As both `Ref`s and `Agent`s update internal state, Clojure provides two event based mechanisms for executing code when an update is performed, the interface can be seen in Listing 3. A watch function, added by `add-watch` and removed by `remove-watch`, is executed when a new value is written to the `Ref` or `Agent` it is attached to, providing a means for reacting to the new value. A validator function is set by `set-validator!` and retrieved by `get-validator`, one validator function can be set per `Ref` or `Agent`, so `set-validator!` overwrites the existing function when a new is set, `nil` is used for no validator. A validator function is executed before a new value is written to a `Ref` or `Agent`, and if it returns `false` the update is discarded.

3.2 Limitations

Clojure's STM implementation lacks constructs to execute side-effects such as system calls like printing to the screen.

```
(def counter-ref (ref 0))

(dosync
  (println (deref counter-ref))
  (if (< (deref counter-ref) 10)
    (alter counter-ref inc)
    (ref-set counter-ref 0)))
```

Listing 4: Print a mutable value inside a transaction

An example combining side-effects with STM can be seen in Listing 4. The example prints the value of a `Ref` before updating it. As updating the `Ref` restarts the transaction if the operation conflicts with another transaction, there are no guarantees for the number of times the value is printed. Restructuring this example to either use an `agent` for printing or returning the original value and then printing after the transaction is possible. The example however shows why combining side-effects and transactions is problematic. A similar problem occurs if a function with side-effects must be synchronised and its return value written to a `Ref`.

```
(def keys-ref (ref []))
(def rows-ref (ref vector-of-rows))

(dosync
  (let [row (first (deref rows-ref))
        next-key (database-insert row)]
    (alter keys-ref conj next-key)
    (alter rows-ref rest)))
```

Listing 5: Inserting rows and returning keys

Listing 5 shows a transaction performing an operation on a database synchronised using STM. The transaction starts by inserting a row into a database and then appending a key to an empty `vector`, before removing the inserted row from the `vector-of-rows`. Having multiple threads executing the transaction concurrently would not guarantee the correct result, as a transaction could restart due to conflicts with the other transactions. As a consequence a row might be inserted into the database multiple times as the transaction cannot ensure that the operation is only executed once.

Creating another transaction for updating `keys-ref` with the keys would allow the operation to be performed, but would not ensure that the two vectors are updated atomically. Restructuring this example to use `agent`s to perform the side-effect would not be possible, as performing the database operation inside an `agent` would not allow the transaction to access the resulting key. Since `agent`s execute asynchronously it is necessary to wait for the `agent` to finish executing using the function `await` before reading the resulting value. This causes the thread to block until all functions sent to the `agent` at that point have been executed, allowing other threads to overwrite the result before it could be read.

The STM implementation in Clojure also lacks constructs for transaction control [8]. Controlling when transactions abort, providing alternative paths of execution or synchronising threads, and not just reads and writes, is not possible.

Clojure depends on its interoperability with Java to use the monitors on Java `Objects` and the constructs from the package `java.util.concurrent` to synchronise threads. These constructs nearly all depend on side-effects to operate, making them incompatible with Clojure's implementation of STM and forcing the use of exceptions to make transactions abort. Blocking a thread currently executing a transaction does not relinquish ownership of `Refs`. This prevents other transactions from using these `Refs` leading to starvation.

```
(try
  (dosync
    (throw TerminateException))
  (catch TerminateException te))
```

Listing 6: Aborting a transaction

The simplest aspect of transaction control, aborting a transaction, must in Clojure be emulated using exceptions as shown in Listing 6. A `try/catch` block around the `dosync` block will catch the exception `TerminateException`. Code inside the transaction can throw this exception which then aborts the transaction. A unique exception is needed to avoid aborting the transaction silently if an error occurs.

While the example shows that aborting a transaction is possible in Clojure, such a simple operation should require only one function call. Performing more advanced operations such as blocking a thread would require use of a blocking concurrency construct in addition to the exception, adding additional complexity.

4. CLOJURE STM EXTENSIONS

Our first extension removes the requirement that forms executed by a transaction must be without side-effects. The second provides manual control over a transaction primarily for synchronisation of threads. Adding these extensions the problems shown in Section 3.2 can be solved without Java.

4.1 STM and Side-Effects

Synchronisation of forms with side-effects using STM can be performed using one of the three methods, defer, irrevocable or compensate, described in Section 2.

4.1.1 Event Management

These methods all depend on being executed at a specific point in a transaction, and that the transaction then is guaranteed not to be restarted when passed this point. To provide this functionality it was necessary to extend Clojure's runtime with an event management system, as the event management systems needed to interact with parts of the STM implementation not accessible through its interface. The event manager interface can be seen in Listing 7.

```
(listen event-key event-fn & event-args)
(listen-with-params event-key thread-local
        delete-after-run event-fn & event-args)

(dismiss event-key event-fn dismiss-from)

(notify event-key)
(notify event-key context)

(context)
```

Listing 7: Interface for event management

An event listener can be registered by either `listen` or `listen-with-params` with a `Keyword` as a key identifying the listener, a function to serve as the listener function and any arguments for that function. `listen-with-params` takes two extra parameters, these indicate if the event should only be registered for the thread calling the function or all threads, and if the event listener should be dismissed after it has been executed once. Both functions return an object used to remove the event listener function through `dismiss`.

Registered listeners are executed when a `notify` function is called with the same `Keyword` used to register the listener. An overload of `notify` takes an extra argument named `context`. The arguments, given through the `context` parameter of `notify`, are made accessible through the function `context` inside any listener executed due to the call to `notify`. This allows a developer to provide data to an event listener both from where the listener is created using arguments to the event listener function, and from where the listeners were notified through the `context` argument. All functions in Listing 7 apart from `context` will throw an `IllegalStateException` if used inside a transaction to prevent them being executed multiple times if a transaction restarts.

4.1.2 Transactional Events

Using the event management infrastructure three events are added to Clojure's STM implementation, `after-commit`, `on-abort` and `on-commit`, as seen in Listing 8.

```
(after-commit & body)
(after-commit-fn event-fn & event-args)

(on-abort & body)
(on-abort-fn event-fn & event-args)

(on-commit & body)
(on-commit-fn event-fn & event-args)

(lock-refs func & body)

(java-ref x)
(java-ref x & options)

(alter-run input-ref func & args)
(commute-run input-ref func & args)
```

Listing 8: Interface for transactional events

Events enable use of side-effects by removing the problem of not knowing if a transaction will restart. Two overloads are created for each STM event. The first takes forms as argument, making it simple to use when the computation has no parameters. The second takes a function and a list of parameters as arguments, hence adding the need to specify a function if arguments are required. The events are executed as follows:

- `after-commit:` after a transaction has committed, providing a synchronise alternative to `agents`.
- `on-abort:` if a transaction aborts, allowing the developer to compensate for executed side-effects.
- `on-commit:` after a transaction is guaranteed not to conflict but before ownership of `Refs` is released, providing a fixed point for executing side-effects and writing the result to `Refs`.

Defer is provided through on-commit as it suspends executing the form until the transaction is guaranteed not to conflict. Irrevocable is provided by on-commit as a transaction, containing only a call to on-commit, is guaranteed to commit after the event listener function is registered. Compensate is provided through on-abort allowing code to undo side-effects. after-commit provides a simpler method compared to Agent for executing side-effects synchronously.

To ensure side-effects are only executed once, the implementation guarantees that the transaction cannot restart when executing an event listener. In after-commit no abort is possible due to these events being executed after a transaction has committed. As side-effects are enabled by on-abort by compensating for updates to Refs, additional updates to these will never make a transaction abort. Restarts due to other Refs being updated will force an STMEventException to be thrown. For on-commit events any Refs written to by the event listener must be locked manually using lock-refs. As only the Refs either read from or written to by the event listener function are locked, multiple events can be executed in parallel if their sets of Refs do not overlap.

lock-refs uses the existing locking mechanisms in Clojures's STM implementation. lock-refs takes as argument either the function ensure if only reading is necessary, commute if shared writes until the event listener function starts executing are necessary, or alter if exclusive writes from when lock-refs are executed are necessary. The other argument can be any sequence of symbols and is intended to be the source code of an event listener function which lock-refs will search through recursively for Refs and lock them according to the first argument passed to lock-refs. lock-refs is implemented to operate like the identity function, allowing it to parse the source code of an event listener function without the developer having to define it twice. As the source code of compiled functions is not accessible programmatically, any Refs written to by compiled functions executed as part of the on-commit event listener must be added explicitly by the developer. If the Refs written to by on-commit are not locked using lock-ref and the transaction is forced to restart, an STMEventException is thrown.

java-ref constructs a JavaRef type, a Ref optimised for use with references to mutable Java objects. Two changes are made compared to Ref. The deref function when used inside a transaction sets the value of the JavaRef to nil to discourage aliasing, making it less trivial to copy a reference to multiple JavaRefs. Secondly, to support MVCC each Ref stores a history of values. For references only a shallow copy is performed, making the stored history a copy of the same reference. As MVCC allows for each transaction to read any value written before that transaction started, a reference to an object can be retrieved despite the object being changed after the transaction started. To prevent this, history is disabled for JavaRefs. Deep copying could be performed but would be computationally expensive for data structures not designed with multiple versions in mind.

alter-run and commute-run perform the same operations as alter and commute but do not set the Ref to the return value. This allows method calls on objects stored in Refs when the object does not return itself as the result, a common occurrence in Java's standard library.

4.2 Transaction Control

The interface for transaction control can be seen in Listing 9, and is based on Harris et. al. [8] with some additions. The additions were added to alleviate problems identified when developing programs in Haskell. retry can only block a thread in a transaction until any Ref read by the transaction is updated, not all Refs. Also, in general a call to retry will be based on an invariant, as the transaction would never complete otherwise. As any update to a Ref will unblock a transaction the invariant might still be true, making the transaction continuously unblock and block. In addition retry provides no mechanisms for explicitly defining a subset of Refs to block on. orElse only allows for developers to define alternative code to execute when a transaction is explicitly forced to retry but not due to conflicts [8, 7].

```
(retry [])
(retry [refs])
(retry [refs func & args])

(retry-all [])
(retry-all [refs])
(retry-all [refs func & args])

(or-else & funcs)
(or-else-all & funcs)

(terminate)
```

Listing 9: Interface for transaction control

retry and retry-all both block until a set of Refs is written to by another transaction, retry blocks until any of the Refs blocked on are written to, while retry-all blocks until all Refs in the set are updated. When no arguments are given, the functions will block based on the Refs read in the transaction. When one argument is given it must be a Ref or a sequence of Refs and the function will block on the specified Refs, allowing the developer to specify an explicit set of Refs instead of relying on what has been read. The last overload takes both Refs, a function returning a boolean and arguments for that function. The function is executed when either one or all of the Refs are written to, depending on whether retry or retry-all is used. If the function returns true the transaction is unblocked. This allows the transaction to be blocked until an invariance is true, and as the function must be provided to retry, all symbols used to determine if the transaction should block are also available for use as arguments to the function.

or-else and or-else-all allow for a transaction to execute alternative functions if one fails. Both take as argument a list of functions and try to execute them in the given order until one of them executes without forcing the transaction to abort, if none of the functions given as arguments could be executed successfully the transaction will abort. The function or-else follows the semantics of Haskell's orElse construct [8, 7], or-else only executes the next function if the first function was forcefully stopped by the execution of retry or retry-all. If the transaction is forced to restart due to conflicts with another transaction the entire transaction will restart. or-else-all will execute the next function if the current one fails either due to a conflict with another transaction or due to the execution of retry or retry-all. This allows or-else-all to use another resource if one is used by another transaction without the developer making additional checks, while or-else provides developers with control over when another function should be executed.

terminate terminates a transaction, meaning that it will abort and continue execution of the next form after the `dosync` that defined the transaction. One use of this function is to combine `terminate` with `or-else` or `or-else-all` to skip a transaction if a resource is unavailable. `or-else` or `or-else-all` can be used to select the first available resource and then have `terminate` be the last function to ensure the transaction terminate if no resources are available.

5. PRELIMINARY EVALUATION

As preliminary evaluation of our extensions we detail how they maintain the ACI properties of Clojure's STM implementation. We also present the findings of a usability evaluation using a small sample size, in addition to the initial performance results. Both are presented to demonstrate to what degree eClojure makes development of concurrent programs simpler and what overhead is added [16].

The transaction control methods replace traditional structures such as `Semaphore`s, constructs which exist in Clojure through its Java interop, so the usability evaluation will focus on development of programs with a high degree of thread synchronisation to compare these two thread synchronisation methods. The extension for executing forms with side-effects in a transaction has no equivalent constructs in Clojure, and is not part of our usability evaluation.

As changes were only made to the STM implementation and the interface between Clojure's Java runtime and standard library written in Clojure, eClojure demonstrates that the extension can be implemented with changes only to a small part of Clojure's runtime. The focus of the implementation has been to keep it simple so we could reason about its behaviour, hence optimisation has not been a priority.

5.1 Clojure STM Properties

The ACI properties guarantee that a transaction is processed reliably, and that updates to mutable data are synchronised correctly. Due to the capability of executing forms with side-effects in transactions, eClojure does not fully fulfil these properties. As the transaction control constructs are without any changes to how a transaction synchronises reads and writes of data, these constructs do not affect the ACI properties. The same holds for `after-commit` as any form executed by `after-commit` is run after a transaction has committed. In this section we will detail the compromises we had to make in our implementation. Clojure defines the ACI properties as follows [1]:

Atomic means that every change to Refs made within a transaction occurs or none do.

Consistent means that each new value can be checked with a validator function before allowing the transaction to commit.

Isolation means that no transaction sees the effects of any other transaction while it is running.

The atomicity property is enhanced to include mutable values and side-effects. To ensure this property is fulfilled the constructs must be used correctly. For `on-commit` `lock-refs` must be used to indicate what `Ref`s will be updated so the transaction is guaranteed not to restart. Having to manually add `lock-refs` was a compromise made to only lock the set of `Ref`s a transaction with side-effects will use so multiple transactions with side-effects can run in parallel, and not automatically parse the forms executed by all `on-commit` functions for `Ref`s. For `on-abort` it is the developers responsibility to remove all side-effects executed by the transaction until it was aborted.

eClojure required no changes to Clojure's validator functionality. Each validator is executed the same for both `Ref`s and `JavaRef`s ensuring consistency. An exception from this property must be made for the `on-commit` functions as they are executed after the validators to ensure that the transactions do not restart after forms with side-effects are executed. Use of `on-abort` allows the property to be fulfilled as compensation code is executed if the transaction aborts and validators execute right before the transaction commits.

Isolation guarantees are unchanged for immutable values stored in `Ref`s and `JavaRef`s as is the norm in Clojure. Due to its use of MVCC and Java objects being represented as references, isolation can only be guaranteed for mutable data stored as references if the data is accessed through a single `JavaRef` only. If an object reference is aliased and stored in multiple `JavaRef`s, a transaction will operate on these as if they were different values. This breaks isolation as writes synchronised through one `JavaRef` can be read through another `JavaRef`. The problem is that `Ref`s store a history of values to support MVCC. However for a Java object the reference is stored in a `Ref`, not the object itself. This leads to multiple references to the same object being stored in the same `Ref`. As a transaction can read an older value from a `Ref` despite another transaction updating the `Ref`, multiple transactions can have an unsynchronised reference to the same mutable object which breaks consistency. To remedy this `JavaRef` was developed. `JavaRef` only stores one version of the reference, so a transaction must restart if it reads data from an object that was changed after the transaction started executing. An alternative would be to perform a deep copy of the object each time an object is updated. By using a `JavaRef` to hold a reference to mutable data, isolation can be guaranteed between transactions if the reference is not aliased. Side-effects outside the program, such as updates to a database, can be observed by the database. However if the database connection object is stored in a `JavaRef`, access will be synchronised between transactions.

5.2 Usability

The usability evaluation was conducted to determine if eClojure simplifies development of concurrent programs compared to Clojure. The evaluation consists of two implementations of the Santa Claus Problem [18]. It is a substitute for a real world program because, while small, it nearly exclusively consists of synchronisation of threads, and larger programs often only contain a very small percentage of code performing synchronisation [16]. The two implementations will be compared based on the following metrics:

- **Lines of code (LOC)** How many LOC the implementation consists of, ignoring empty lines and comments.
- **Development time** How much time was spent on that implementation of the Santa Claus Problem.

LOC will be used as a measure as it makes the resulting code comparable to other studies [14, 13, 16]. Though LOC in itself is not that informative, it gives an indication of how much code is needed to use the constructs of the language. A short program on its own is not guaranteed to have been simple to write, therefore development time is included as it shows how efficiently the developer can create solutions with the constructs available [16]. Based on experience with Haskell, we anticipate that eClojure will improve on Clojure. A short subjective discussion of the two implementations will provide additional context regarding the usability of the added language constructs [13]. Both programs were developed by the same developer who had experience with similar transaction control constructs from Haskell and concurrent programming with traditional concurrency control structures from Java. The eClojure version was developed first to ensure that all knowledge gained from developing the first version, would be counted against eClojure for a fair comparison.

The implementation in eClojure consisted of 22.22 % fewer LOC and took half the time to implement compared to the version in Clojure. The reduced number of LOC is the result of adding the `retry` constructs in eClojure, instead of throwing exceptions and synchronising using a `Semaphore` and a `CyclicBarrier`. The overall structure of the two versions turned out to be very similar, and the time used for each was initially similar. However more time was spent ensuring that the Clojure implementation did not cause a deadlock. This is due to synchronisation with a `Semaphore` and a `CyclicBarrier` being more complicated then using our extensions.

5.3 Performance Overhead

A benchmark was performed to determine if the additions in eClojure added a performance overhead compared to Clojure. We are aware that some benchmark suites for STM implementations exist, but as documented by Massimiliano Ghilardi [6] they either consist of a set of micro benchmarks or larger programs where only a small part of the program performs synchronisation, making the difference in performance of the STM implementation negligible compared to the overall execution time of the program.

As our evaluation focuses on the overhead added to an existing STM implementation, we use three micro benchmarks which only execute code that was changed between the two versions of the implementation. This approach was chosen to ensure that as few factors as possible could influence the experiments, and that any added overhead was not hidden by a long running program. The three benchmarks were:

Empty A `dosync` function without any arguments.

Deref A `dosync` function executing a `deref` function.

Database The database example shown in Section 3.2 with `database-insert` returning a random number as key.

An empty transaction will show the overhead the extensions have added to `dosync`, `deref` will show if the instrumentation needed for `retry` adds overhead, and the database example will indicate if any added overhead is still measurable in a real transaction. The experiments were executed on a MacBook Pro with a 2.8 GHz Intel Core i7 and 16GB DDR3-1600 MHZ ram running Mac OS X 10.11.3 and the Oracle JDK 8 Update 71. Criterium 0.4.3 was used to ensure that the Java virtual machine was initialised, garbage collection was taken into account and enough samples were collected to eliminate outliers. A graph of the results was created using matplotlib [9], and can be seen in Figure 1.

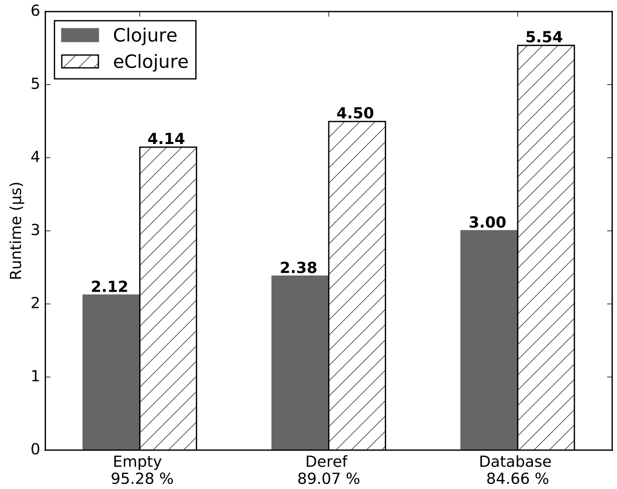


Figure 1: Execution time for each benchmark and the percentage overhead added by eClojure.

The results show that the additions in eClojure have increased the execution time of a transaction. However, there is a tendency that when operations are performed in a transaction the overhead decreases, indicating that the overhead is constant and could become insignificant when executing larger transactions. We hypothesise that the overhead could be reduced even further if the implementation is optimised based on the need of real world applications with uses of STM beyond what is needed for the micro benchmarks.

6. CONCLUSION

In this paper we demonstrate how the expressiveness of the STM implementation in Clojure is limited by the lack of methods for synchronising functions with side-effects and constructs for manually controlling how a transaction must be executed. We discuss alternative solutions for the presented limitations and the alternative concurrency constructs a developer is forced to use both in terms of Clojure and Java. To demonstrate that these limitations are unnecessary given the existing STM implementation, we present two sets of extensions that remove these limitations and provide the complete source code in addition to unit tests.

The first set implements methods for synchronising forms with side-effects through STM, while allowing transactions to be executed in parallel given no overlap in their write sets. Constructs are added for synchronising access to Java objects, allowing STM to be used for both synchronising access to mutable data structures and for execution of external effects by multiple transactions in parallel.

The second set of extensions provides control over a transaction. A transaction in eClojure can block threads until a write to `Ref` is performed. If the program must be in a specific state for the thread to unblock, it can be checked using a function returning a boolean. Constructs were implemented that allow a transaction to select a different path of execution if the transaction is forced to restart due to a conflict with another transaction, or manually made to retry by the developer. Through these methods the STM implementation in eClojure can be used for synchronising threads based on state, and developers can prevent transactions from restarting by defining an alternative path of execution in case of conflicts.

We evaluate eClojure based on its compliance with the ACI properties of Clojure's STM implementation. In addition we demonstrate through a preliminary performance benchmark and a usability evaluation with a small sample size that the added constructs simplify concurrent programming by reducing the need for complementing STM with other concurrency constructs. eClojure adds only a constant performance overhead to the implementation when executing transactions, as the relative overhead is decreased by the execution of increasingly larger transactions.

7. FUTURE WORK

The event handling system of eClojure introduced the use of contexts in event handling functions. This functionality is used to indicate which `Refs` can be changed during the `on-commit` event. We see the possibility of adding more detailed control over the transaction [3], by creating an API for interacting with the transaction itself and providing it as an object through the `context` function to both `on-commit` and `on-abort`. Providing `on-abort` with information about what side-effects were performed would make `on-abort` able to only compensate for side-effects already executed, instead of forcing the developer to define an `on-abort` event listener function for each side-effect executed.

The need for manually indicating what `Refs` an `on-commit` event will modify, using the function `lock-refs`, adds complexity both due to the extra code and overhead as `lock-refs` is forced to check if each symbol is a `Ref` to the implementation. Removing the need for `lock-refs` and extending `Ref` to provide the functionality of `JavaRef` based on the type of its input, would simplify the use of side-effects in transactions as `Refs` would be locked automatically.

Additional evaluation of performance and usability for eClojure should be performed based on a larger set of concurrent programs and additional developers to determine how the implementations can be better optimised, both in terms of performance and usability. An interesting area for performance evaluation is parallel versions of data structures that use a reorganisation step such as hash tables [20].

8. ACKNOWLEDGMENTS

We thank Thomas Stig Jacobsen and Daniel Rune Jensen for their contributions in the master thesis [10], producing the initial ideas build upon in this paper.

9. REFERENCES

[1] Refs and Transactions. http://clojure.org/reference/refs. Accessed: 2016/02/13.

[2] L. Baugh and C. Zilles. An Analysis of I/O And Syscalls In Critical Sections And Their Implications For Transactional Memory. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2008.

[3] A. Bieniusa, A. Middelkoop, and P. Thiemann. Actions in the Twilight: Concurrent Irrevocable Transactions and Inconsistency Repair (extended version). Technical report, 257, Institut für Informatik, Universität Freiburg, 2010. Accessed: 2016/02/13.

[4] N. G. Bronson, H. Chafi, and K. Olukotun. CCSTM: A library-based STM for Scala. *The First Annual Scala Workshop at Scala Days*, 2010.

[5] B. Demsky and N. F. Tehrany. Integrating file operations into transactional memory. *Journal of Parallel and Distributed Computing*, 2011.

[6] M. Ghilardi. High performance concurrency in Common Lisp-hybrid transactional memory with STMX. In *7th European Lisp Symposium*, 2014.

[7] T. Harris and S. P. Jones. Transactional memory with data invariants. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[8] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[9] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science and Engineering*, 2007.

[10] D. R. Jensen, S. K. Jensen, and T. S. Jacobsen. Unifying STM and Side Effects in Clojure. http://projekter.aau.dk/projekter/files/213827517/p10DOC.pdf. Accessed: 2016/02/13.

[11] M. Kalin and D. Miller. Clojure for Number Crunching on Multicore Machines. *Computing in Science and Engineering*, 2012.

[12] E. A. Lee. The Problem With Threads. *IEEE Computer*, 2006.

[13] M. Luff. Empirically investigating parallel programming paradigms: A null result. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2009.

[14] S. Markstrum. Staking claims: a history of programming language design claims and evidence: a positional work in progress. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2010.

[15] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. *ACM SIGARCH Computer Architecture News*, 2006.

[16] S. Nanz, S. West, K. Soares da Silveira, and B. Meyer. Benchmarking usability and performance of multicore languages. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013.

[17] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm. Accessed: 2016/02/13.

[18] J. A. Trono. A new exercise in concurrency. *ACM SIGCSE Bulletin*, 1994.

[19] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc. xCalls: safe I/O in memory transactions. In *Proceedings of the 4th ACM European conference on Computer systems*, 2009.

[20] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, 2008.

Session IV: Applications

CANDO - A Common Lisp based Programming Language for Computer-Aided Nanomaterial Design and Optimization

Christian E. Schafmeister
Chemistry Department
Temple University
1901 N. 13th Street
Philadelphia, PA
meister@temple.edu

ABSTRACT

CANDO is a compiled programming language designed for rapid prototyping and design of macromolecules and nanometer-scale materials. CANDO provides functionality to write programs that assemble atoms and residues into new molecules and construct three-dimensional coordinates for them. CANDO also provides functionality for searching molecules for substructures, automatically assigning atom types, identifying rings, carrying out conformational searching, and automatically determining stereochemistry, among other things. CANDO extends the Clasp implementation of the dynamic language Common Lisp. CANDO provides classes for representing atoms, residues, molecules and aggregates (collections of molecules) as primitive objects that are implemented in C++ and subject to automatic memory management, like every other object within the language. CANDO inherits all of the capabilities of Clasp, including the easy incorporation of C++ libraries using a C++ template programming library. This automatically builds wrapper code to expose the C++ functionality to the CANDO Common Lisp environment and the use of the LLVM library[1] to generate fast native code. A version of CANDO can be built that incorporates the Open Message Passing Interface C++ library[2], which allows CANDO to be run on supercomputers, in order to automatically setup, start, and analyze molecular mechanics simulations on large parallel computers. CANDO is currently available under the LGPL 2.0 license.

Categories and Subject Descriptors

D.2.12 [**Software and its engineering**]: Interoperability; D.3.4 [**Software and Programming Languages**]: Incremental compilers

Keywords

Computational chemistry, Common Lisp, LLVM, C++, interoperation

9th ELS May 9–10, 2016, Kraków, Poland

1. INTRODUCTION

Using computers to design molecules and materials presents many challenges. There are almost no software tools that allow a chemist to write efficient programs that automatically construct molecules according to a design and optimize those designs based on their ability to organize functional groups in desired three-dimensional constellations. Software like this would greatly facilitate the creation of designer macromolecules and materials.

My laboratory is a synthetic organic chemistry group that has developed a unique approach to synthesizing large molecules with well defined three-dimensional structures[3]. These molecules can be designed to act as new therapies, to create new catalysts, and to create channels that purify water and separate small molecules from each other. These molecules are assembled like peptides, DNA and carbohydrates, using a common set of interchangeable building blocks linked through common linkage chemistry. By assembling them in different sequences an enormous number of different well defined, three-dimensional structures are created, some of which could have extremely valuable properties. The challenge is to find those active molecules and with the right software, together with large, parallel, modern supercomputers we may be able to find them.

We synthesize stereochemically pure bis-amino acids that we connect together through pairs of amide bonds. These create large molecules with programmable three-dimensional shapes and functional groups called "spiroligomers". In the last several years, we have demonstrated that we can design small spiroligomers that bind a protein[4], and we have demonstrated three spiroligomer based catalysts that accelerate chemical reactions[5, 6, 7]. Recently we have demonstrated that we can connect multiple spiroligomers together to create macromolecules that present large surfaces with complex constellations of functional groups[8]. These molecules can now be synthesized to a size where it is impossible to design them using existing tools and so I have developed CANDO, a molecular programming environment that will enable the design of large, functional spiroligomers as well as any other foldamers[9] such as peptides, peptiods, betapeptides and any other large molecules assembled from modular building blocks.

CANDO is a compiled, dynamic programming language that implements objects essential to molecular design including atoms, residues, molecules and aggregates as memory-efficient C++ classes. CANDO manages these objects using modern memory management (garbage collection), freeing the programmer from having to deal with memory management for exploratory code. CANDO is written in C++ and makes it easy to integrate existing C++, C, and Fortran libraries that efficiently carry out molecular simulations and tightly integrate them with high-level code that sets up calculations and carries out analyses of results. The idea is to do everything within CANDO: build new molecules, assign force-field atom types and atomic charges, run molecular dynamics simulations, and analyze the results of the simulations, all within a single compute node, without having to leave the CANDO programming environment. This is done using external C++ libraries like OpenMM from within CANDO, using the exposed C++ application programming interface (API) provided by the library. CANDO is designed to be for chemistry what Mathematica is for symbolic mathematics or R is for statistial analysis - a general purpose programming language, tailored for the specific problem domain of computational chemistry and molecular and nanomaterials design. CANDO is an implementation of the Common Lisp dynamic programming language, a compiled, industrial strength language with a well-defined standard that includes incremental typing and can interactively compile code in a way that is both highly interactive and dynamic while generating fast native code.

2. RELATED WORK

CANDO is not the only programming language that specifically targets chemistry. The Scientific Vector Language (SVL) is embedded within the Molecular Operating Environment (MOE) software package developed by the Chemical Computing Group Inc[10]. SVL contains primitive objects for atoms, residues, molecules and systems of molecules. SVL is touted as being both a compiled and interpreted language. MOE is a commercial product and the source code to the SVL compiler is not open source[11]. SVL uses a syntax that could be termed "C-like". More generally, Scientific Python (SciPy)[12], and scientific programming languages like Julia could be used to implement chemistry objects, but they do not implement the classes required to represent molecular systems[13]. There are significant differences between CANDO and SVL. CANDO builds on and extends the existing language Clasp Common Lisp, therefore CANDO is able to use the large library of existing software that has been developed in Common Lisp[14]. CANDO is open source, and is freely available to the scientific community under the LGPL 2.1 license. This is to facilitate its use in research and allow it to be adapted to run on large academic parallel supercomputers without paying a per-processor fee. SVL and MOE are closed source and commercial products and licensing must be negotiated with the Chemical Computing Group.

In the realm of molecular design, CANDO is related to the software package Rosetta[15]. Rosetta is a suite of command line applications and bindings for languages like Python, that carry out different operations such as protein folding prediction and protein/protein interface design. Rosetta is highly tuned to work with natural proteins and uses the Brookhaven Protein Database to construct rotamer libraries that describe the preferred conformations of amino acids. This structural information is not available for unnatural building blocks like spiroligomers and other foldamers and non-proteogenic building blocks require a great deal of work to incorporate into Rosetta. CANDO provides tools for carrying out conformational analysis on molecules constructed from unnatural building blocks and enables the construction of conformational databases for building blocks. CANDO is more general than Rosetta with respect to the chemistry that it can deal with and while it was designed with spiroligomers in mind, it supports molecular design with any molecular building blocks.

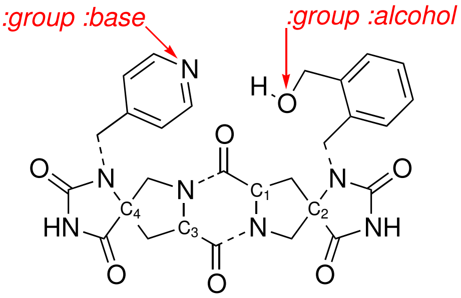


Figure 1: A bond line drawing of a molecule that can form an internal hydrogen bond between the base and alcohol if it has the correct stereochemistry at C_1, C_2, C_3 and C_4. The annotations in red add properties to the atom's property list.

3. BUILDING MOLECULAR STRUCTURES

One of the ways that chemists communicate with each other is using "bond line formalism", a graphical notation that describes the chemical structure of molecules. A molecule is represented with a structure like Figure 1. Structures like this can be drawn using commercial packages such as ChemDraw[16] and ChemDoodle[17]. These programs are able to save these structures in an XML-based format called cdxml. CANDO can read cdxml files directly and construct the molecular graph of the molecule with atom names, atom elements, bond orders, connectivity and other annotations attached. CANDO interprets some of the bond types as codes to annotate atom, residue, and molecule properties with Common Lisp symbols and value pairs so that programs can easily identify features within the molecules. The red keyword symbols and arrows in Figure 1 are examples of this. The red arrows indicate that the property/value pairs (:group :base) and (:group :alcohol) are attached to the pyridine nitrogen and benzyl alcohol oxygen respectively.

The code to read a cdxml file into CANDO is shown in listing 1. In bond line formalism (and thus cdxml), hydrogen atoms are not drawn on carbon atoms because their number can be inferred from the structure. Carbon atoms are represented by vertices and ends of lines and other atoms are labeled with their element name or a name that starts with the element name (eg: C1). CANDO automatically identifies the number of implicit hydrogens and adds them to the molecules graph. The resulting structure obtained by loading the cdxml file is put in the dynamic variable *agg*, which is an instance of the CHEM:AGGREGATE class containing one instance of the CHEM:MOLECULE class, which contains four instances of the CHEM:RESIDUE class. Each residue contains a collection of CHEM:ATOM instances that contain lists of pointers to CHEM:BOND objects. The CHEM:BOND objects create a bidirectional graph by pointing to pairs of CHEM:ATOM objects. A CHEM:RESIDUE object contains multiple atoms that are bonded to each other and form a common unit of molecular structure like an amino acid, DNA nucleotide, or monosaccharide. A collection of CHEM:RESIDUE objects can be bonded to each other to form a CHEM:MOLECULE object. A collection of CHEM:MOLECULEs can be grouped together into a CHEM:AGGREGATE instance, used to describe a collection of molecules that interact with each other.

```
(defparameter *cd*
  (with-open-file
    (fin #P"base-alcohol.cdxml"
      :direction :input)
    (chem:make-chem-draw fin)))
(defparameter *agg* (chem:as-aggregate *cd*))
```

Listing 1: Load cdxml file

Carbon atoms that have sp^3 hybridization are bonded to four other atoms that roughly point to the corners of a tetrahedron with the sp^3 carbon atom at its center. If the four bonded groups are all unique then there are two possible three-dimensional arrangements of those four groups, termed the "stereochemical configuration" of the central carbon. CANDO can automatically identify the carbons that can have such stereochemical configurations, and allows the programmer to define their configurations using the keyword symbols `:S` or `:R`. In the structure above, the four stereocenters were labeled C_1, C_2, C_3, and C_4 and in Listing 2 the four stereocenters are all set to the `:S` configuration. The assignment of stereochemical configuration sets a field in the carbon's CHEM:ATOM instance which will be used later to add a stereochemical configuration restraint when three-dimensional coordinates of the molecule are constructed.

```
(defparameter *stereocenters*
  (sort (cando:gather-stereocenters *agg*)
        #'string< :key #'chem:get-name))
(cando:set-stereoisomer-func
  *stereocenters*
  (constantly :S) :show t)
```

Listing 2: Set stereocenter configurations

CANDO can load files of various formats that describe the three-dimensional structure of molecules (PDB, MOL2). However, CANDO's purpose is to build three-dimensional structures from a simple graph description of the molecule with no prior knowledge of the three-dimensional structure. To do this CANDO uses a very robust approach to building chemically reasonable three-dimensional structures for molecules from first principles. Quantum mechanics and the Schroedinger equation can be used to build chemically reasonable three-dimensional structures of molecules, however the calculations for even simple molecules are extremely time consuming. A less expensive approximation is to use "molecular mechanics," which approximates bonds as mechanical springs and describes molecules using a "molecular force-field function" that contains many empirically-determined parameters including ideal bond lengths, angles, torsion angles, and through-space interactions describing the electrostatic and van Der Waals interactions between non-bonded atoms. CANDO has built in the popular AMBER force-field,[18] which has the form (1). CANDO adds several additional terms to the force-field equation that restrain atoms in different ways. For instance, the stereochemical configuration defined above is maintained with a special stereochemical configuration restraint term that penalizes molecular structures where the four groups attached to the stereochemical carbon are in the wrong tetrahedral arrangement.

$$
\begin{aligned}
E_{\text{total}} = &\sum_{\text{bonds}} K_r(r - r_{eq})^2 + \sum_{\text{angles}} K_\theta(\theta - \theta_{eq})^2 \\
&+ \sum_{\text{dihedrals}} \frac{V_n}{2}[1 + \cos(n\phi - \gamma)] \qquad (1) \\
&+ \sum_{i<j}\left[\frac{A_{ij}}{R_{ij}^{12}} - \frac{B_{ij}}{R_{ij}^{6}} + \frac{q_i q_j}{\epsilon R_{ij}}\right] + Restraints
\end{aligned}
$$

To build the three-dimensional structure of a molecule from first principles, the programmer first instructs CANDO to assign random coordinates to every atom within a 20x20x20 Angstrom box using the `(cando:jostle *agg* 20)` command (Figure 2). Then CANDO is instructed to minimize the force-field energy using non-linear optimization. CANDO implements three non-linear optimizers in C++ (a steepest descent, a conjugate gradient, and a truncated Newton-Raphson[21] optimizer), switching between them as the optimization proceeds. Prior to carrying out the non-linear optimization, CANDO assigns force-field atom types using substructure pattern recognition and assigns force-field parameters. Currently CANDO assigns all atomic charges to zero for model building. In the future, atomic-charge approximation algorithms will be implemented in CANDO Common Lisp. The non-linear-optimization is then initated using the `(my-minimize *agg* *ff*)` function (Listing 3).

```
(cando:jostle *agg* 20)
(defparameter *ff* (energy:setup-amber))
(my-minimize *agg* *ff*)
(cando:chimera *agg*)
```

Listing 3: Energy minimization

The energy minimization function first minimizes the energy with non-bonded interactions turned off, and then with non-bonded interactions turned on. This is to allow atoms and bonds to initially pass through each other so that rings do not get tangled up with each other. The resulting structure is generated in a few seconds and shown in Figure 3. The structure can then be displayed using the molecular visualization program Chimera[20] by calling the `(cando:chimera *agg*)` function, which writes the structure to a temporary file and calls the Unix `system` function to open it (Figure 3).

```
(defun my-minimize (agg force-field)
  "Minimize the conformational energy"
  (let* ((energy-func
          (chem:make-energy-function
           agg force-field))
         (minimizer
          (chem:make-minimizer
           :energy-function energy-func)))
    (cando:configure-minimizer
     minimizer
     :max-steepest-descent-steps 1000
     :max-conjugate-gradient-steps 5000
     :max-truncated-newton-steps 0)
    (chem:enable-print-intermediate-results
     minimizer)
    (chem:set-option energy-func
                     'chem:nonbond-term nil)
    (cando:minimize-no-fail minimizer)
    (chem:set-option energy-func
                     'chem:nonbond-term t)
```

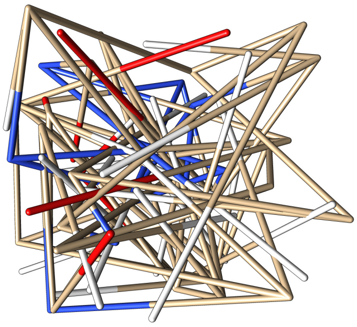


Figure 2: A molecule with random coordinates.

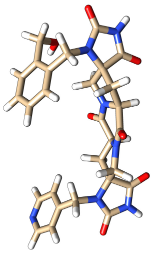


Figure 3: A molecule after energy minimization.

```
    (cando:minimize-no-fail minimizer)))
```

Listing 4: Definition of my-minimizer function

CANDO is capable of many other manipulations to prepare structures for calculations such as molecular dynamics simulations. For example, Figure 4 shows a model of a potential membrane channel that was designed using CANDO.

```
(defparameter *channel*
  (cando:load-mol2 "trichannel.mol2"))
(cando:chimera *channel*)
(defparameter *membrane* (load-membrane))
(cando:chimera *membrane*)
```

Listing 5: Loading a channel from a mol2 file and a model membrane

This molecule is designed to create a pore in a cell membrane. To prepare it for a simulation, it needs to be embedded within a model of a fragment of cell membrane in water. To do this, the channel needs to be superimposed onto the membrane and the lipid and water molecules that overlap with the channel atoms need to be removed. A problem is that lipid molecules are long and almost always overlap the channel while water molecules are small and so different criteria for overlap are needed for the two types of molecules. If the same distance criterion was used for each, then there would be large gaps where too many lipids were removed or there would be lots of close contacts between many water molecules and the channel where not enough water molecules were removed. This is a problem that is easily solved by writing a bit of code. The function `load-membrane` (Listing 6), loads the membrane and then uses the `chem:map-molecules` function to count the number of atoms in each molecule of the membrane and assign the symbol `:solvent` or `:lipid` to each molecule based on the number of atoms it contains.

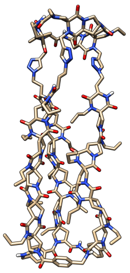


Figure 4: A potential membrane channel designed using CANDO.

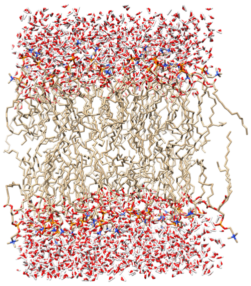


Figure 5: A model of a small segment of cell membrane.

```
(defun load-membrane ()
  "Load the membrane from the psf/pdb files
and name each of the molecules
within it based on whether they
are solvent (<= 3 atoms) or lipid."
  (let ((agg-membrane
    (cando:load-psf-pdb "POPC36.psf"
                        "POPC36.pdb")))
    (chem:map-molecules
     nil
     (lambda (m)
       (if (<= (chem:number-of-atoms m) 3)
           (chem:set-name m :solvent)
           (chem:set-name m :lipid)))
     agg-membrane)
    agg-membrane))
```

Listing 6: Load a membrane from a pair of psf/pdb files and assign each molecule in the membrane to be :solvent or :lipid.

Removing the various molecules from the membrane model that overlap with the channel is carried out using the function `cando:remove-overlaps`. This function accepts two AGGREGATE objects and a Common Lisp function object that returns the minimim overlap distance depending on the type of molecule that is overlapping with an atom in the channel (Listing 7). The ability to pass functions and closures to other functions is a very useful feature of the Common Lisp language that CANDO makes extensive use of. The `cando:remove-overlaps function` is written in CANDO Common Lisp and it carries out a very common but time-consuming operation in programming for chemistry. It needs to compare every atom in the first aggregate to every atom in the second aggregate, an expensive NxM calculation that would normally be implemented in C++. Since CANDO Common Lisp is compiled, the algorithm can be implemented in Common Lisp and the calculation is fast.

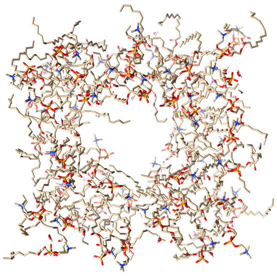


Figure 6: The top view of the cell membrane after overlaps with the channel are removed. Water molecules have been hidden for clarity.

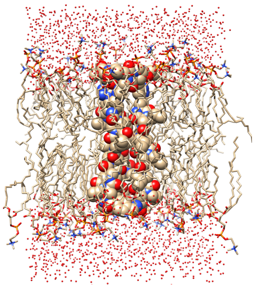


Figure 7: The side view of the channel embedded within the cell membrane. The channel is rendered using spheres for atoms and waters as dots for clarity.

After the molecules are removed from the membrane model, a new aggregate, *merged*, is created. The molecules from the membrane and the molecule of the channel are added to it and the result is saved to a mol2 file (Listing 8).

```
(defun overlap-func (molecule)
  "Small waters (:solvent) overlap
if within 3A of a channel atom while
:lipid overlaps if within 0.6A"
  (if (eq (chem:get-name molecule) :solvent)
      3.0
      0.6))
(cando:remove-overlaps
 *membrane*
 *channel*
 :distance-function #'overlap-func)
(cando:chimera *membrane*)
```

Listing 7: Removing membrane molecules that overlap with the channel.

```
(let ((agg (chem:make-aggregate)))
  (chem:map-molecules
   nil
   (lambda (m) (chem:add-molecule agg m))
   *membrane*)
  (chem:map-molecules
   nil
   (lambda (c) (chem:add-molecule agg c))
   *channel*)
  (defparameter *merged* agg))
(cando:save-mol2 *merged* "merged.mol2")
(cando:chimera *merged*)
```

Listing 8: Create a new aggregate and add the membrane and channel molecules to it.

At this point the structure in *merged* in the file merged.mol2 can be used to construct the input files for a molecular dynamics simulation using AMBER[19]. Molecular dynamics simulates the motions of molecules and is used by chemistry researchers to understand many molecular phenomena. CANDO does not currently generate the input files for a molecular dynamics simulation directly but relies on an external program called LEaP (written by this author), which is the primary tool used by computational chemists to set up their calculations for AMBER[19]. Future plans are to write the code to generate AMBER input files in CANDO Common Lisp and to incorporate molecular dynamics packages directly into CANDO and communicate with them through an application programming interface (API) using CANDO's built-in abilities to interface with external libraries.

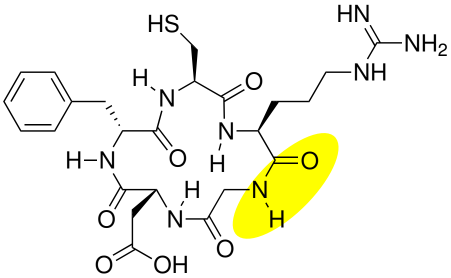


Figure 8: A cyclic peptide with one secondary amide bond highlighted in yellow.

4. CHEMICAL PATTERN RECOGNITION

CANDO incorporates a modified version of SMARTS,[22] which is a language that allows a programmer to specify a chemical substructure and then search for that substructure within the graph of a molecule. SMARTS searches chemical structures for substructures in the same way that regular expressions are used to search text. With SMARTS, a string such as "CCCC" will match a continuous chain of carbon atoms connected by single bonds. The SMARTS code: "$(N1(C2)(~[#1]3)~C4(=O5)C6)" will match a secondary amide bond and also bind the six atoms that form the amide bond to numerical keys so that the atoms can be recovered from the match. This capability has myriad uses and is the basis of the automatic atom-type assignment required for structure building. In the example below, a cyclic peptide is searched using a substructure matcher that recognizes secondary amide bonds (Listing 9). All five of the amide bonds are found and the nitrogen atoms of the amides are displayed (Listing 10).

```
(defparameter *cyclorgd*
  (load-cdxml #P"cyclorgd.cdxml"))
(defparameter *smarts*
  (make-cxx-object 'chem:chem-info))
(chem:compile-smarts
 *smarts*
 "$(N1(C2)(~[#1]3)~C4(=O5)C6)")
(chem:map-atoms
 nil
 (lambda (a)
   (let ((match (chem:matches *smarts* a)))
     (when match
       (format t "Amide nitrogen: ~a~%" a))))
 *cyclorgd*)
```

Listing 9: Find all secondary amide bonds within a cyclic peptide.

```
Amide nitrogen: #<ATOM :N/:N@0x118032610>
Amide nitrogen: #<ATOM :N/:N@0x118032190>
Amide nitrogen: #<ATOM :N/:N@0x118032c10>
Amide nitrogen: #<ATOM :N/:N@0x117991d90>
Amide nitrogen: #<ATOM :N/:N@0x117991790>
```

Listing 10: Found five amide bonds within the cyclic peptide.

5. MOLECULAR MECHANICS ENERGY FUNCTIONS

CANDO uses a very flexible approach to implementing non-linear optimization for structure building with molecular mechanics. CANDO allows the programmer to define new energy function terms by providing the functional form of the energy term and then automatically generates optimized computer code to evaluate the function and its analytical first and second derivatives. CANDO uses automatic symbolic differentiation and a compiler that optimizes the resulting symbolic expressions to remove common subexpressions and minimize the number of expensive reciprocal and square-root operations. The energy terms, the analytical gradient terms, and the analytical Hessian terms of the AMBER force field and the additional restraint terms are all implemented within CANDO in this way. Currently this facility is implemented in Mathematica code[23] and it generates efficient C code. The Mathematica and resulting C code is included in the CANDO source code. In the future this facility will be implemented in CANDO Common Lisp and generate efficient static single assignment (SSA) LLVM-IR. This will allow users to add new energy terms to the molecular mechanics force-field of CANDO to support other force fields and add new restraint terms and to use CANDO's non-linear optimizers to optimize things like partial charges in residues.

6. SERIALIZATION OF OBJECTS

CANDO has a built-in serialization format based on Common Lisp S-expressions. It uses the Common Lisp printer and reader facilities to serialize any collection of objects, including those containing internal cross-referencing pointers, into a compact, human-readable format. This serialization facility is used within CANDO to store and retrieve objects to files and to communicate data and code between processing nodes on large parallel clusters. The serialization facility is extended to CANDO C++ classes by adding one C++ method to the definition of the C++ class.

7. MEMORY MANAGEMENT OF CHEMISTRY OBJECTS

Programs that implement molecular design algorithms need to repeatedly allocate memory to represent atoms, residues, molecules, and other objects. They then need to allocate other objects to generate three-dimensional coordinates for molecular designs, predict the molecular properties *in silico*, and then release those memory resources to advance to the next design. Molecules are most conveniently represented as bi-directional graphs of atoms connected to each other through bonds. These form enormous numbers of reference loops and primitive reference-counting techniques are inadequate to manage memory. Robust garbage collection is therefore necessary to avoid crippling memory leaks.

Table 1: Calculating the 78th Fibonacci number 10^7 times

Language	Seconds	Factor	Stdev(sec)	Rel. cclasp
clang C++	0.76	1	0.016	0.3
sbcl 1.2.11	0.63	1	0.008	0.2
cclasp	2.9	4	0.022	1.0
Python 2.7	77.7	102	0.36	26.8
bclasp	639	839	n/c	221

CANDO is based on the Common Lisp implementation Clasp[14] and it uses the memory management facilities that Clasp provides to manage all chemistry objects. Clasp supports both the Boehm[24] and the Memory Pool System (MPS)[25] garbage collectors with the MPS garbage collector intended for use in production and the Boehm garbage collector used for bootstrapping and building the CANDO executable. The additional 239 C++ classes that CANDO adds to Clasp are all managed by the garbage collectors in the same way that the standard Common Lisp objects are managed. This means that objects that are no longer used are automatically discarded and objects that remain are continuously compacted in memory to release memory for further use. For algorithms where the overhead of memory management is considered to be too high for performance code, the algorithms can be implemented in C++ and the responsibility for memory management is taken over by the programmer.

In order to allow the MPS library to work with CANDO's C++ core, every pointer to every object that will move in memory needs to be updated whenever that object is moved. CANDO fully automates the identification of C++ pointers that need to be updated by MPS, using a static analyzer written in Clasp Common Lisp. The static analyzer uses the Clang C++ compiler front end to parse the more than 360 C++ source files of CANDO and uses the Clang AST-Matcher library to search the C++ Abstract Syntax Tree, in order to identify every class and global pointer that needs to be managed by MPS.

8. CANDO, CLASP AND COMMON LISP

CANDO is a full implementation of the Common Lisp language including the Common Lisp Object System. It supports the Common Lisp software packages: ASDF, the Superior Lisp Interaction Mode for Emacs (SLIME), and Quicklisp. It is a superset of the Clasp implementation of Common Lisp and has every feature that Clasp has, including the C++ template programming library "clbind", which makes it easy to integrate C++ libraries with CANDO[14]. In the past year, Clasp (and CANDO) have been enhanced with tagged pointers, and immediate fixnums, characters, and single-float types. Additionally, the Cleavir compiler[26] has been fully integrated into Clasp and the speed of the most optimized code generated by Clasp is within a few factors of C++ (Table 1). The "cclasp" compiler is the Cleavir compiler within Clasp. The "bclasp" compiler is a less sophisticated compiler within Clasp that bootstraps cclasp and was Clasp's only compiler when reported last year[14]. So, by one measure, the performance of Clasp has improved by a factor of 221 in the past year.

9. THE CANDO SOURCE CODE

CANDO adds 155,000 logical source lines of C++ code to the 186,000 logical source lines that makes up Clasp (version 0.4). CANDO extends Clasp with 239 additional C++ classes, 948 additional C++ instance methods and 66 C++ functions that implement objects and algorithms related to chemistry. In addition, CANDO adds a growing body of CANDO Common Lisp code that implements algorithms important to chemistry and molecular design. CANDO is freely available at http://github.com/drmeister/cando.

10. CONCLUSIONS AND FUTURE WORK

CANDO is a general, compiled programming language designed for rapid prototyping and design of macromolecules and nanometer-scale materials. Future work will include developing bindings external molecular modeling packages and the OpenGL graphics library and developing a rich library of molecular modeling tools within CANDO Common Lisp.

11. LICENSE

Clasp is currently licensed under the GNU Library General Public License version 2.

12. ACKNOWLEDGMENTS

Thanks to Robert Strandh for providing Cleavir and for many fruitful discussions. Thanks to Göran Rydqvist for helpful discussions and support to present this work at ELS 2016. Thanks to Stephanie Schneider for helpful feedback.

13. REFERENCES

[1] *Lattner, C. (2005)* “Masters Thesis: LLVM: An Infrastructure for Multi-Stage Optimization”, Computer Science Dept., University of Illinois at Urbana-Champaign, http://llvm.cs.uiuc.edu

[2] *Richard L. Graham and Timothy S. Woodall and Jeffrey M. Squyres. (2005)* “Open MPI: A Flexible High Performance MPI”; Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics

[3] *Schafmeister, C. E., Brown, Z. Z., and Gupta, S.* “Shape-Programmable Macromolecules” Accounts Of Chemical Research 41, no. 10 (2008): 1387–1398. doi:10.1021/ar700283y

[4] *Brown, Z. Z., Akula, K., Arzumanyan, A., Alleva, J., Jackson, M., Bichenkov, E., Sheffield, J. B., Feitelson, M. A., and Schafmeister, C. E.* “A Spiroligomer alpha-Helix Mimic That Binds HDM2, Penetrates Human Cells and Stabilizes HDM2 in Cell Culture.” Plos One 7, no. 10 (2012): e45948. doi:10.1371/journal.pone.0045948

[5] *Zhao, Q., Lam, Y.-H., Kheirabadi, M., Xu, C., Houk, K. N., and Schafmeister, C. E.* “Hydrophobic Substituent Effects on Proline Catalysis of Aldol Reactions in Water.” The Journal of Organic Chemistry 77, no. 10 (2012): 4784–4792. doi:10.1021/jo300569c

[6] *Kheirabadi, M., Celebi-Olcum, N., Parker, M. F. L., Zhao, Q., Kiss, G., Houk, K. N., and Schafmeister, C. E.* “Spiroligozymes for Transesterifications: Design and Relationship of Structure to Activity.” Journal Of The American Chemical Society 134, no. 44 (2012): 18345–18353. doi:10.1021/ja3069648

[7] *Parker, M. F. L., Osuna, S., Bollot, G., Vaddypally, S., Zdilla, M. J., Houk, K. N., and Schafmeister, C. E.* “Acceleration of an Aromatic Claisen Rearrangement via a Designed Spiroligozyme Catalyst That Mimics the Ketosteroid Isomerase Catalytic Dyad.” Journal Of The American Chemical Society 136, no. 10 (2014): 3817–3827. doi:10.1021/ja409214c

[8] *Zhao, Q. and Schafmeister, C. E.* “Synthesis of Spiroligomer-Containing Macrocycles” Journal Of Organic Chemistry 80, no. 18 (2015): 8968-8978. doi:10.1021/acs.joc.5b01109

[9] *Hill, D. J., Mio, M. J., Prince, R. B., Hughes, T. S., and Moore, J. S.* “A Field Guide to Foldamers” Chemical Reviews 101, no. 12 (2001): 3893–4012. doi:10.1021/cr990120t

[10] *Santavy, M., Labute, P.*“SVL: The Scientific Vector Language”, https://www.chemcomp.com/journal/svl.htm

[11] *Molecular Operating Environment (MOE), 2013.08*; Chemical Computing Group Inc., 1010 Sherbooke St. West, Suite #910, Montreal, QC, Canada, H3A 2R7, 2016.

[12] *Jones E, Oliphant E, Peterson P, et al.* “SciPy: Open Source Scientific Tools for Python”, 2001–, http://www.scipy.org/

[13] *Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah (2014)*, “Julia: A fresh approach to numerical computing”. http://arxiv.org/abs/1411.1607

[14] *Christian E. Schafmeister, (2015)* “Clasp - A Common Lisp that Interoperates with C++ and Uses the LLVM Backend”; Proceedings of the 8th European Lisp Symposium, pg 90. http://github.com/drmeister/clasp

[15] *Leaver-Fay, A., Tyka, M., Lewis, S. M., Lange, O. F., Thompson, J., Jacak, R., Kaufman, K., Renfrew, P. D., Smith, C. A., Sheffler, W., Davis, I. W., Cooper, S., Treuille, A., Mandell, D. J., Richter, F., Ban, Y.-E. A., Fleishman, S. J., Corn, J. E., Kim, D. E., Lyskov, S., Berrondo, M., Mentzer, S., Popović, Z., Havranek, J. J., Karanicolas, J., Das, R., Meiler, J., Kortemme, T., Gray, J. J., Kuhlman, B., Baker, D., and Bradley, P. (2011)* “ROSETTA3: an Object-Oriented Software Suite for the Simulation and Design of Macromolecules.” Methods in enzymology 487, (2011): 545–574. doi:10.1016/B978–0–12–381270–4.00019–6

[16] *Mills, N. (2006).* “ChemDraw Ultra 10.0”. J. Am. Chem. Soc. 128 (41): 13649–13650. doi:10.1021/ja0697875

[17] *Todsen, W.L. (2014).* “ChemDoodle 6.0”; J. Chem. Inf. Model., 2014, 54 (8), pp 2391–2393 doi: 10.1021/ci500438j

[18] *Cornell, W. D., Cieplak, P., Bayly, C. I., Gould, I. R., Merz, K. M., Ferguson, D. M., Spellmeyer, D. C., Fox, T., Caldwell, J. W., and Kollman, P. A.* “A Second Generation Force Field for the Simulation of Proteins, Nucleic Acids, and Organic

Molecules" J. Am. Chem. Soc. 118, no. 9 (1996): 2309–2309. doi:10.1021/ja955032e

[19] *D.A. Case, et. al and P.A. Kollman (2015),* "AMBER 2015", University of California, San Francisco.

[20] *Pettersen EF, Goddard TD, Huang CC, Couch GS, Greenblatt DM, Meng EC, Ferrin TE.* "UCSF Chimera–a visualization system for exploratory research and analysis." J. Comput. Chem. 2004 Oct;25(13):1605–1612.

[21] *Nash, Stephen G. (2000).* "A survey of truncated-Newton methods". Journal of Computational and Applied Mathematics 124 (1–2): 45–59. doi:10.1016/S0377–0427(00)00426–X

[22] "SMARTS Theory Manual", Daylight Chemical Information Systems, Santa Fe, New Mexico. http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html

[23] Wolfram Research, Inc., Mathematica, Version 10.3, Champaign, IL (2015).

[24] *Boehm, H.,* "Simple Garbage-Collector-Safety", Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.

[25] *Richard Brooksby. (2002)* The Memory Pool System: Thirty person-years of memory management development goes Open Source. ISMM02.

[26] https://github.com/robert–strandh/SICL

A Case Study in Implementation-Space Exploration

Alexander Lier   Linus Franke   Marc Stamminger   Kai Selgrad

Computer Graphics Group, Friedrich-Alexander University Erlangen-Nuremberg, Germany

{alexander.lier, linus.franke, marc.stamminger, kai.selgrad}@fau.de

ABSTRACT

In this paper we show how a feature-oriented development methodology can be exploited to investigate a large set of possible implementations for a real-time rendering algorithm. We rely on previously published work to explore potential dimensions of the implementation space of an algorithm to be run on a graphics processing unit (GPU) using CUDA. The main contribution of our paper is to provide a clear example of the benefit to be gained from existing methods in a domain that only slowly moves toward higher level abstractions. Our method employs a generative approach and makes heavy use of Common Lisp-macros before the code is ultimately transformed to CUDA.

1. INTRODUCTION

When developing algorithms to be used in time-critical application domains, such as real-time rendering, many different implementation variants need to be evaluated to arrive at the method that best exploits the available resources. This is even more true in research where the spectrum of solutions to investigate is potentially larger. Higher-level languages such as Common Lisp, can ease this process of finding the best solution considerably. However, while providing great flexibility and productivity, they are not available in all domains and for all applications. This can be due to technical limitations (vendor-specific languages, compatibility) or policies (certified processes, coherent working environment) as well as due to resistance from developers unwilling to embrace change.

In earlier work we proposed formulating C and similar languages in an S-Expression syntax and only transforming them to their native notation as late in the process as possible [22] using C-Mera[1]. This allows employing a macro-heavy methodology to generate many different variants of an input program, and to express it on a very high level. We believe that this scheme, even if very unfamiliar to programmers used to C, can help make higher level paradigms available to that domain (also by supporting guerrilla adoption [22, 21]).

For domain exploration we recently proposed employing a feature-oriented programming paradigm to implement the exploration process [21]. To this end we use cm-fop[2], a very lightweight library that provides this feature-oriented programming model to C-Mera.

In our case study we show how these methods can be applied to find highly efficient implementations of a post-processing depth of field effect for real-time rendering.

The following section, related work, is divided into two parts. At first we cover generative meta-programming techniques followed by rendering methods for depth of field. After a short review of C-Mera in a dedicated section, we depict several aspects and details of our depth of field algorithm. Details and examples of the actual meta programming approach are given in the implementation section. Thereafter, the generated resulting code is discussed and evaluated, and our findings are summarized in a short conclusion.

2. RELATED WORK

In this paper we propose employing a generative methodology to explore the space of possible implementations of real-time depth of field rendering algorithms. Section 2.1 gives a short review of generative programming techniques while Section 2.2 provides an introduction to depth of field rendering. The latter is divided further, whereby the first part discusses depth of field in general. Afterwards, the difference between gathering and scattering methods is elucidated, followed by a description of how the scattering algorithm can be harnessed for GPU application.

2.1 Generative Meta-Programming

In the following we will focus on related work concerned with general purpose methods providing domain-specific abstractions. For a comprehensive summary of general generative programming methods see Czarnecki and Eisenecker [4].

The most ubiquitous approach to generative programming is C++ template meta programming (TMP) [26, 4]. It has been applied to a wide range of problems, also in graphics (for example with RTfact [23], a ray tracing library). We believe that, while certainly convenient and unobtrusive to use when working in a C++ environment, exploration is seriously impaired by this approach as the maintenance overhead of the meta code becomes a burden in itself [7, 13, 21].

[1] github.com/kiselgra/c-mera

[2] github.com/kiselgra/cm-fop

9th ELS May 9–10, 2016, Kraków, Poland

Also note that TMP is only available in C++, only partially in CUDA [15] and not in other, very similar languages such as OpenCL [8] and GLSL [11].

C-Mera is a multi-stage programming language [24], that is, a language that is embedded in a host language (in our case Common Lisp). Embedded languages are compiled into the host language and the resulting program is then further compiled or interpreted. In the case of C-Mera the evaluation of the input program constructs the internal Common Lisp representation that is then pretty printed to C-style code.

Examples of other multi-stage programming languages are MetaOCaml [3], an extension for staging OCaml, Terra [6], a low-level language embedded in Lua, and AnyDSL [14], which gives the programmer explicit control over when certain parts of the program are to be evaluated.

Feature-oriented programming [17, 2] offers additional degrees of freedom regarding programming versatility. A *feature* is a unit of functionality that provides an interface for configuration. A feature-oriented program is then composed of features that together provide its implementation. This scheme aims to provide well-structured programs that can be configured to provide different incarnations by varying the implementations of the underlying features [1]. Our work relies heavily on feature-oriented programming, as it is demonstrated in more detail in Section 5.

2.2 Depth of Field Rendering

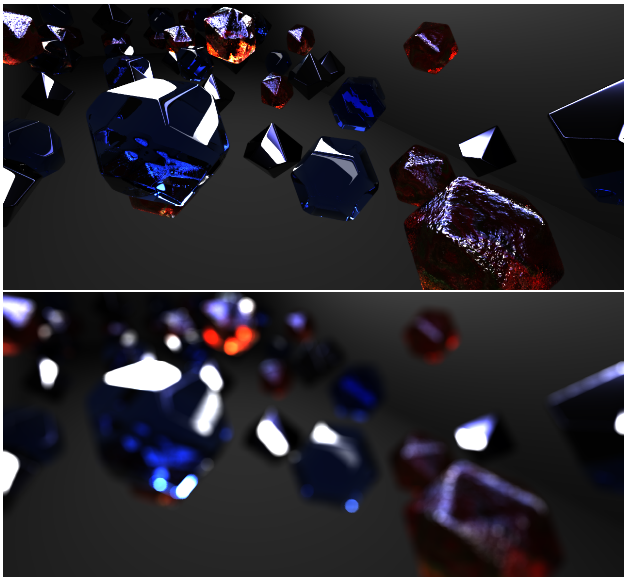


Figure 1: Rendered image (top) and post processed image showing depth of field (bottom).

Depth of field is a physical effect caused by the finite size of the lens in an imaging system (e.g. in a camera or in the eye). Rays of light are collected through the aperture of the system (the pupil of the eye) and focused by the lens on the image sensor (the retina). As the refractive power of the lens can only be in one state at any given time only light arriving from a given distance is actually in focus and mapped to a single point on the sensor. Light from a different distance is mapped to a circular region (or, in case of a camera, to a region in the shape of the camera's aperture, known from polygonal shapes of out-of-focus lights in movies and artistic photographs). This *circle of confusion* is the reason out-of-focus objects appear blurred. Figure 1 shows an example computed using our implementation of a post-processing method.

When synthesizing images this is an important effect for generating more plausible results, as the depth perception is skewed for scenes that are shown completely in-focus. Demers [5] describes this effect in great detail and provides an overview of methods of generation. In the following we will only touch on a few methods and refer to Demers [5] for more details.

Gathering vs Scattering. The most common method to implement depth of field is by adding it to an previously generated image. This can be done in two ways, namely via *gathering* or *scattering*. Gathering approaches apply a (bilateral) image-space blur filter where the filter's size depends on the current pixel's circle of confusion [5, 18]. The main limitation of these methods is that the ordering of the participating pixels is not considered correctly and only a weighted average is computed [19]. With scattering methods the problem is approached differently: the rendered image is interpreted as a point-sampled scene representation and the individual points are scaled to form circles according to the original pixel's circle of confusion. These scaled points ("splats") are then sorted and accumulated (in order) as semi-transparent objects, thus ensuring correct weighting [16, 12]. The limitation of these methods is that they require a global ordering of the splats be established, which is an expensive operation.

Recent work on tiled shading [9] and particle accumulation [25] can be applied to remove this limitation of scattering depth of field algorithms in a straightforward fashion [19].

Fast GPU Particle Accumulation. High-performance particle accumulation can be implemented by employing a tiling-based scheme that maps well to graphics hardware. Instead of globally sorting the particles, they are binned into screen-space tiles (e.g. of 16×16 pixels, potentially a $1 : n$ mapping) and the tiles are then sorted independently and in parallel [25]. The computation of the contribution for each pixel is then a simple process that traverses the list of particles in the pixel's associated tile. This process, too, maps very well to hardware as (in CUDA terms) the execution can be set up such that each thread-block traverses a single tile, and thus the threads in a warp run with great coherency.

In Section 4 we list choices of how to implement the accumulation phase of this technique for depth of field rendering. We also show how C-Mera, together with cm-fop, can be used to explore the space of possible solutions with feature-oriented programming.

3. BRIEF REVIEW OF C-Mera

C-Mera is a simple transcompiler embedded in Common Lisp. It allows writing programs in an S-Expression syntax that is transformed to C-style code providing simple extension for languages with similar syntax on top of the core C support. For example, the C-Mera distribution provides modules for C++, CUDA, GLSL and OpenCL. The main goal of providing an S-Expression syntax is to write the compiler such that it evaluates the syntax to construct a syntax tree when the input program is read, thereby allowing interoperability with the Common Lisp-system, most importantly by providing support for Lisp-style macros. To keep this part short we refer to the original C-Mera paper [22] for a more detailed description of the system and its implementation.

With the use of macros the input program no longer represents a plain syntax tree, but a semantically annotated tree that is transformed according to the implementation of the semantic nodes (macros). The utility of such a system ranges from simple, ad-hoc abstractions and programmer-centric simplifications [22] to providing otherwise hard to achieve programming paradigms for C-like languages [21] and even to fully fledged domain specific languages [20]. In this paper we provide a case study of applying C-Mera to provide a higher level programming paradigm for GPU algorithm development.

Figure 2 shows an example from our application domain, which is processed by C-Mera, and C-style code that is generated by it. The overall appearances of the input (top) and the generated code (bottom) resemble each other but the former applies S-Expressions, whereas C-style syntax and infix expressions are required in the latter. As can be seen, C-Mera supports convenient and essential language elements from C, for example the member accessor *color.x* (line 10,11,12), infix increment *++i* (line 1, top), type suffix *3.1415f* (line 7, top), and further notations that are not shown here. Additionally, C-Mera renames variable names that are restricted in C, for example *dist-x* (line 6, top).

```
1   (for ((int i 0) (< i num-per-tile) ++i)
2     (decl ((dataSpl elem (aref lists tile-index i))
3            (int dist-x (- (funcall elem.x) gid.x))
4            (int dist-y (- (funcall elem.y) gid.y))
5            (float coc-sq (funcall elem.sq-coc)))
6       (if (<= (+ (* dist-x dist-x) (* dist-y
            dist-y)) coc-sq)
7         (decl ((float area (* coc-sq 3.1415f))
8                (float alpha (/ 1.0f area)))
9           (+= color.x (* alpha-dest alpha elem.r))
10          (+= color.y (* alpha-dest alpha elem.g))
11          (+= color.z (* alpha-dest alpha elem.b))
12          (*= alpha-dest (- 1.0f alpha))))))
```

```
1   for (int i = 0; i < num_per_tile; ++i){
2       dataSpl elem = lists[tile_index][i];
3       int dist_x = elem.x() - gid.x;
4       int dist_y = elem.y() - gid.y;
5       float coc_sq = elem.sq_coc();
6       if (((dist_x * dist_x) + (dist_y * dist_y))
            <= coc_sq) {
7           float area = coc_sq * 3.1415f;
8           float alpha = 1.0f / area;
9           color.x += (alpha_dest * alpha * elem.r);
10          color.y += (alpha_dest * alpha * elem.g);
11          color.z += (alpha_dest * alpha * elem.b);
12          alpha_dest *= (1.0f - alpha);
13      }
14  }
```

Figure 2: The most basic particle accumulation loop as used as input for C-Mera (top) and the resulting generated C-style code (bottom).

4. IMPLEMENTATION-SPACE

We focus our analysis on the accumulation of particles, which can be described briefly as follows: Every pixel of the resulting image must be synthesized form a number of particles that might affect it. To do so, the program has to iterate through a previously sorted list with possible candidates in reach that might affect the resulting pixel. Every tile, a group of 16 × 16 pixels, has one associated list, therefore a list is shared by 256 pixels.

The process of sorting the entries in the tile lists, as well as the subsequent accumulation benefit greatly from a compact memory layout in which single entries can be transferred en-bloc. For our CUDA implementation this means that the entries should be no more than 16 bytes, such that they can be encoded as `uint4`. However, the required fields are the pixel's (high resolution) color, screen-space position and camera distance. This data can be stored compactly, but the best choice not only depends on the number of bits reserved for each entry, but also on the effort to unpack the respective fields and how how likely they will be accessed. In our implementation we tested two different basic node layouts.

The most obvious point to evaluate different implementations is how the CUDA warp and block configuration is set up and employed during traversal. Here we evaluated two different approaches: Loading large chunks (256 elements) of the tile-lists (with an average length of 1300) to a buffer of shared memory, and then processing the list chunk-by-chunk with all threads working in parallel on the same data. This, however, requires synchronization after the data has been loaded to ensure that it is available to all threads. The second approach is to only load blocks of warp size (i.e. 32) to shared memory and process smaller chunks. The benefit of this approach is that no synchronization is necessary, however, at the cost of smaller batches in the shared cache and redundant loads on the block level.

To gain greater insight into the effect of blending front-to-back vs back-to-front, both approaches have been analyzed. Naturally, the front-to-back method performs better [25] as it offers the option to terminate early when the pixel is saturated. This leads to the question of how finely checking for early termination is advisable: after each accumulation step, or only after each chunk, which interacts with the aforementioned chunk size?

We also evaluated many small-scale optimizations such as explicitly enabling caching to L1, storing vs computing certain values and peeling off parts of loops to remove conditionals from the inner loop. In the end we arrived at 320 different (and meaningful) *combinations* that are implemented using cm-fop in 300 lines of feature definitions, feature implementations and the algorithm-template that expand to more than 16000 lines of CUDA code.

5. IMPLEMENTATION

As the previous section shows, our implementation space expands into multiple dimensions; thus we must consider and evaluate many versions of the accumulation loop, which might differ heavily from each other and from the non optimized version shown in Figure 2. Starting from the basic implementation and with a rough draft of the desired variations, we can incrementally extend the existing solution with functionnlities, also in reaction to the results from previously tested variants. This leads to an iterative and especially exploratory programming methodology. In this section we will discuss our meta implementation, starting with an explanation of why we have chosen features over plain macros. Given examples will at first focus on feature usage and later examine target and feature interaction in more detail.

From Macros to Features. Relying solely on Common Lisp's macro system to facilitate previously mentioned extensions has the risk of becoming tedious, since particular variation points can mutually exclude each other and macros do not support convenient configurability of their implementation. Therefore, if wanting to write a macro that combines multiple varying results, one has to implement each of its possible expansions inside a single block of conditionals. An example of such a macro is given in Figure 3. Therefore, we employ cm-fop, C-Mera's library for feature-oriented programming to ease this procedure. This library enables using features that are essentially macros with a built-in system that automatically implements and resolves conditional expansion. A feature-oriented definition equivalent to the macro from Figure 3 is shown in Figure 4. In contrast to plain macros the feature system is able to recognize the desired expansion code by means of a *configuration variable*, which can be defined and stays valid within a lexical scope and thus supports nesting of different *configurations.* Additionally, the feature system decouples the feature definition from its implementations, with the result that the definition of a feature is only required once and its likely multiple and divergent implementations can be written individually. Furthermore, writing a feature implementation does not require to manually declare its dependencies, nor define conditional expansion. Thus, unlike a macro that incorporates such a behaviour, feature implementations only require minimal boiler-plate code. Nevertheless, it should be mentioned that cm-fop's feature system relies on Common Lisp's macro and object system, but without the need to manually define the conditional expansions, cm-fop improves the handling of multiple implementations over plain macros considerably.

```
(defmacro early-out (target condition &body body)
  (cond ((eql target 'no-early-out)
         `(progn ,@body))
        ((eql target 'blockwise-early-out)
         `(if ,condition
            (progn ,@body)
            (break)))
        (t
          `(error "The target ~a is not
               specified" ',target))))
```

Figure 3: Example macro-implementation with multiple expansion possibilities.

Feature Setup. As a first example we introduce a simple feature to the algorithm from Figure 2. The most basic part of extending the particle accumulation is to exit the loop when the pixel is opaque because collecting further elements will not change the resulting color. To do so we construct a feature equal to the macro shown in Figure 3 that wraps the body with an if-statement and uses a break operation to exit the loop. The corresponding feature setup is shown in Figure 4.

Before we implement features, we define their possible targets (lines 1 and 2). Features are defined once (line 4) and support one implementation per target combination (lines 6 to 12). Targets can be derived from each other, as it is the case here, thus the combination of the currently used most specific target and its available implementations determine the expanded code. For example, if the implementation for *blockwise-early-out* is not given, but that target is used, the more general *no-early-out* will be used. However, if implementations for more specific targets are given, but a less specific target is used, the best fitting implementation (according to CLOS's [10] method lookup) is chosen. As can be seen in Figure 4, the body for a feature implementation is similar to the body of a standard macro, but there are multiple implementations for the same feature. The upper implementation (line 6 and 7) returns the body passed in without modifications. The other one (line 9 to 12) splices the body inside an if-statement, places the condition in the designated position, and introduces a break-statement as the else case.

```
1  (define-target no-early-out)
2  (define-target blockwise-early-out no-early-out)
3
4  (define-feature early-out (condition &body body))
5
6  (implement early-out (no-early-out)
7    `(progn ,@body))
8
9  (implement early-out (blockwise-early-out)
10   `(if ,condition
11      (progn ,@body)
12      (break)))
```

Figure 4: Construction of a feature for a conditional break

Elementary Feature Utilization. The example application shown in Figure 5 depicts the use of features (top) and their resulting code (bottom), whereby feature applications are highlighted in orange and targets in green. Multiple targets can be combined with *make-config* into one single configuration (top, line 1 and 6). The *with-config* form takes a configuration as an argument and declares it as locally valid for features used within its lexical scope. Depending to the configuration used, each *early-out* feature expands into different code.

The *no-early-out* target adds no further code nor changes the body, whereas the *blockwise-early-out* target adds an if-clause and a break-statement. This behaviour is the essential element that we want to utilize. Normally if we introduce additional variations, we must clone and partially rewrite every version for every additional divergence. Thus, the number of possible implementations to write grows exponentially. Our solution for this problem, as we already proposed in previous work [21], is to use a single, general implementation that handles each diverging point individually by applying the proper feature expansion.

Based on the examples from Figure 4 and Figure 5 an applicable multi-variant-aware implementation is shown in Figure 6. As can be seen, we now only require one implementation of the accumulation-loop that can expand into multiple versions depending on the configuration passed in.

Pinpoint Implementation. By adding an increasing amount of expansion possibilities to the unified implementation, we

```
(with-config (make-config no-early-out)
  (for ((int i 0) (< i num-per-tile) ++i)
       (early-out (> alpha-dest 0.01)
        ....

(with-config (make-config blockwise-early-out)
  (for ((int i 0) (< i num-per-tile) ++i)
       (early-out (> alpha-dest 0.01)
        ....
```

```
for (int i = 0; i < num_per_tile; ++i){
    dataSpl elem = lists[tile_index][i];
    ...
}

for (int i = 0; i < num_per_tile; ++i){
    if (> alpha_dest 0.01) {
       dataSpl elem = lists[tile_index][i];
       ...
       }
    }
    else
        break
}
```

Figure 5: Feature evaluation: Depending on the configuration or targets used, highlighted in green (top), the same features, highlighted in orange, expand into different resulting code (bottom)

arrive at the most general implementation of the accumulation loop, which is shown in Figure 7. This algorithm template is capable of expanding into 320 versions of the accumulation loop. Most of the features used are implemented similarly to previous examples and are mapped straightforwardly to one single element previously described in the implementation space in Section 4. More sophisticated features are shown in Figure 8. These features are used together to assemble one single variation inside the implementation space. The given example generates code that iterates chunk-wise over an input list (*loop-over-blocks*, lines 1 and 11)), after which the inner loop processes the single chunk elements (*process-blocks*, lines 6, 22, 26). *Process-blocks* is not used directly within the feature-implementation of *loop-over-blocks*, but appears later on, inside its body as seen in line 3 in Figure 7. Processing a list chunk-wise requires a special case handling, since the last iteration only processes the residual list elements.

One possibility to manage this is to check each iteration step whether the currently processed chunk is the last one and to set the upper limit of the inner loop according to the number of remaining elements. A respective implementation is shown in the upper part (line 1 to 9) of Figure 8. In this case, *loop-over-blocks* does not distinguish between the first blocks ($0 <= i < iterations$) and the last one ($i == iterations$), thus iterates over all chunks. The associated feature (*process-blocks*, line 6), then limits the upper bound for the inner loop (N) by setting it either to the standard block-width (*elems-per-iter*) or, if the outer loop reached the last element, to the size of the last chunk (*last-iter-width*). In short, *loop-over-blocks* iterates over the complete range of chunks and *process-blocks* tests whether the last chunk is to be processed to set suitable limits for the inner loop.

A different approach handling the last chunk is shown in the lower part (line 11 to 27) of Figure 8. The underlying concept is to process the last chunk separately. Consequently, the outer loop iterates over every chunk except for the last one, which must be handled individually outside the loop. The benefit of this method is that the conditional assignment (line 7) can be omitted and the necessary *process-blocks* features (line 21 to 27) reduce their complexity. Yet, as it can be seen, we now employ two, slightly different *process-blocks* features, which depend on different targets. One deploys a loop, which iterates over the full range of the block size, and the other one processes the width of the final chunk.

To fuse the looped code with that of the last chunk, having been processed separately, both of the described *process-blocks* features need to be used. Since the algorithmic procedure for every block is the same, the body of the *loop-over-blocks* feature can be duplicated and used for both parts sequentially. The corresponding implementation (lines 11 to 19) simply duplicates the forwarded body, whereby one is placed within the for-loop (line 16) and the other is appended afterwards.

Both bodies are implemented at the same and cannot be changed usefully by means of list modification at this point. However, they still must generate different code. Therefore, we substitute the currently valid configuration individually for each body. Since it is not desirable to overwrite the configuration completely, the newly introduced configurations are composed of the targets previously passed in (*config*, line 15 and 18). These are already being used for the global configuration, and extended by the respective, locally required targets (*full-block*, line 15 and *peel-block* line 18). Eventually we have two nearly equal sections, which expand into two different specific implementations.

These examples were aimed at providing further insight into how we approached the design of a merged implementation for all meaningful variants previously described in Section 4.

In the following section we will evaluate each instance in terms of their performance in order to identify the optimal combination.

```
(defmacro instantiate (config)
  `(with-config ,config
     (for ((int i 0) (< i num-per-tile) ++i)
       (early-out (> alpha-dest 0.01)
         ...

(instantiate (make-config no-early-out <do-this>
      ...))
(instantiate (make-config blockwise-early-out
      <skip-that> ...))
```

Figure 6: Single implementation for multiple variants

6. EVALUATION AND RESULTS

In this section we briefly evaluate the generator and resulting code of our generic implementation on a Nvidia Geforce GTX Titan, 980, and 980Ti graphics card. We strive to analyze each possible combination of aspects and GPU architecture to identify substantial coherences and special cases. However, to cover every generated accumulation loop, we have to consider 320 versions and testing all of them on three GPUs results in 960 individual measurements.

All time measurements in the context of implementation aspects and architectures are shown in Figure 9. It should

```
(with-iteration-bounds
  (loop-over-blocks ,config
    (early-out (> alpha-dest 0.01f)
      (process-blocks
        (load-current-element elem
          (decl (((dist-type) dist-x (dist x))
                 ((dist-type) dist-y (dist y))
                 ((coc-type-in-node) coc-sq
                                     (funcall
                                       elem.sq-coc)))
            (if (<= (+ (* dist-x dist-x)
                       (* dist-y dist-y)) coc-sq)
              (with-alpha
                (set-color x r)
                (set-color y g)
                (set-color z b)
                (set-sample)))))))
    (sync-after-iteration)))
```

Figure 7: Multidimensional implementation with features

be noted that the visualization is normalized, thus the lower bound is representing the shortest processing time and the upper bound the longest. In addition, for each graphics card different values for upper and lower bounds were applied. This representation was chosen due to better exposition of specific patterns.

Performance Evaluation. The most efficient combination for the Maxwell architecture (980 and 980Ti) is highlighted in green and the equivalent set for the Kepler architecture (Titan), which is highlighted in blue and interestingly, as well suitable for the Maxwell architecture but not vice versa. Despite the fact that the best Titan measurement is only 0.1 ms faster than the slowest GTX 980Ti measurement, all GPUs share favorable implementation aspects. The resulting code of the best implementation for the Maxwell architecture can be found in the appendix in Figure 10.

A surprising insight is that it is not always the best choice, as initially assumed, to implement an early-out behaviour. As can be seen, all version highlighted in violet, the most aggressive early-out method, are in most cases slower than versions highlighted in yellow, which are implemented with only one exit attempt per processed block, and versions highlighted in red, which are implemented completely without early-out mechanism. This being said, applying a Maxwell GPU, a moderate early-out technique seems to be a better choice over omitting early-out completely. Employing a Kepler GPU, early-out techniques should be avoided, at least in our use case.

Further discoveries are the superior performance for the Maxwell architecture, when the last work unit is processed separately, and for both architectures, when using float instead of integer values to compute the distance, even though this requires additional conversions with a *__half2float()* call. The remaining aspects seem to have only a minor or no impact at all.

In contrast to the plot of the GTX Titan, it is striking how similar the measurement charts of the GTX 980 and the GTX 980Ti appear even though they represent different performance ranges. Although noticeable, it is not very surprising, since both GTX 980 and 980Ti share the same architecture.

```
(implement loop-over-blocks (check-residual-block)
  `(for ((int i 0) (<= i iterations) ++i)
             (load-threads-local-data i)
      ,@body))

(implement process-blocks (check-residual-block)
  `(decl ((int N (? (== i iterations)
      last-iter-width elems-per-iter)))
     (loop-over-loaded-block (j N)
     ,@body)))

(implement loop-over-blocks (peel-residual-block)
  `(progn
     (for ((int i 0) (< i iterations) ++i)
       (load-threads-local-data i)
       (with-config (make-config ,@config
           full-block)
         (progn ,@body)))
     (load-threads-local-data iterations)
       (with-config (make-config ,@config
           peel-block)
         (progn ,@body))))

(implement process-blocks (full-block)
  `(loop-over-loaded-block (j elems-per-iter)
     ,@body))

(implement process-blocks (peel-block)
  `(loop-over-loaded-block (j last-iter-width)
     ,@body))
```

Figure 8: Implementation for checking (top) and peeling (bottom) residual list elements.

Code Evaluation. Our preliminary objective was a general implementation of the accumulation loop of our depth of field algorithm [19]. At first we extended the initial concept with simple features, followed by a few iterations of including and testing newly emerging variation possibilities leading to an implementation that provides 320 versions of our algorithm.

Extending, debugging, and maintaining has been done with ease, since, per variant only one location of the program code has to be considered and modifications affect generated instances of features globally. This behaviour is an additional benefit in itself, precisely because we now can guarantee that each specific instance of the generated feature aspect is identical, in contrast to manually copied and and modified code. With this assurance we do not risk to draw the wrong conclusion comparing unequal or faulty code instances and are able to keep many variations, which, as the comparison across the architecture generations shows, can be beneficial in the long run.

The final C-Mera code consists of 300 lines (275 without comments) of feature implementations and expands into over 19000 lines (16000 without comments) of CUDA code that provides 320 distinct kernels.

7. CONCLUSION

In this paper, we showed how we were able to discard redundant instances of essentially similar code fragments by merging them into general structures. Instead of implementing the whole algorithm for each of its possible variants. the algorithm is implemented only once and every ambiguity is replaced with its respective feature.

This approach enabled us to simply unfold, examine, and maintain 320 different versions of the same algorithm to eventually determine the best fitting feature-set in terms

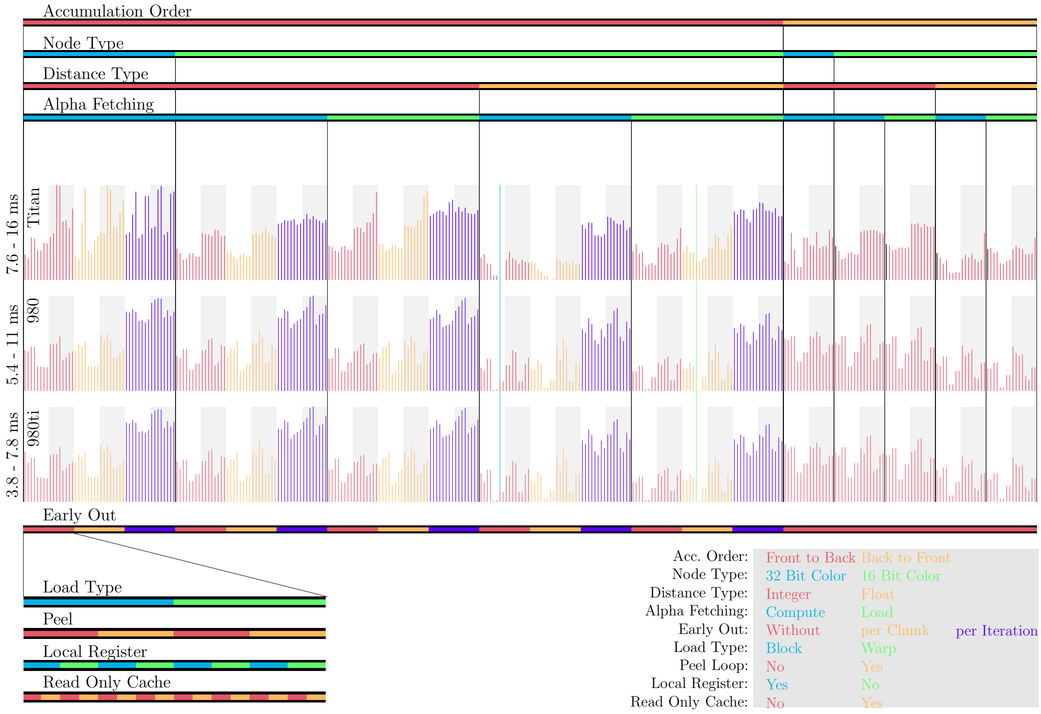


Figure 9: Individual measurements of various feature combinations on 3 GPUs.

of performance for specific GPUs and architectures.

By fully expanding every possible combination we were able to analyze and maintain a much broader range of measurements and identify unexpected findings. In addition, we can securely rely on the accuracy of each kernel, since the expansion of individual feature is globally consistent and investigation on the output code can be done with ease.

In conclusion it can be said that employing a feature oriented programming methodology can proof quite useful when it comes to exploring and analyzing a vast amount of possibly suitable variants, especially when interesting future variations are to be expected.

Acknowledgments

The authors gratefully acknowledge the generous funding by the German Research Foundation (GRK 1773).

8. REFERENCES

[1] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development, July/August 2009. Refereed Column.

[2] S. Apel and C. Kästner. Virtual Separation of Concerns - A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[3] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *Proceedings of the 2Nd International Conference on Generative Programming and Component Engineering*, GPCE '03, pages 57–76, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, May 2000.

[5] J. Demers. Depth of field: A survey of techniques. In R. Fernando, editor, *GPU Gems*. Pearson Higher Education, 2004.

[6] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA, 2013. ACM.

[7] A. Fredriksson. Amplifying C. http://voodoo-slide.blogspot.de/2010/01/amplifying-c.html, 2010.

[8] K. O. W. Group. *The OpenCL Specification*, March 2016.

[9] T. Harada, J. McKee, and J. C. Yang. Forward+: Bringing deferred lighting to the next level. In *Eurographics 2012 - Short Papers Proceedings, Cagliari, Italy, May 13-18, 2012*, pages 5–8, 2012.

[10] S. E. Keene. *Object-oriented programming in COMMON LISP - a programmer's guide to CLOS.* Addison-Wesley, 1989.

[11] J. Kessenich, D. Baldwin, and R. Randi. *The OpenGL Shading Language*, January 2014.

[12] J. Krivanek, J. Zara, and K. Bouatouch. Fast depth of field rendering with surface splatting. In *Computer Graphics International, 2003. Proceedings*, pages 196–201. IEEE, 2003.

[13] M. McCool, S. Du, T. Tiberiu, P. Bryan, and C. K.

Moule. Shader algebra. *ACM Transactions on Graphics*, pages 787–795, 2004.

[14] R. Membarth, P. Slusallek, M. Köster, R. Leißa, and S. Hack. High-performance domain-specific languages for gpu computing. GPU Technology Conference (GTC), March 2014.

[15] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, September 2015.

[16] M. Potmesil and I. Chakravarty. A lens and aperture camera model for synthetic image generation. In *Proceedings SIGGRAPH 1981*, pages 297–305. ACM, 1981.

[17] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, Lecture Notes in Computer Science, pages 419–443. Springer-Verlag, June 1997.

[18] G. Riguer, N. Tatarchuk, and J. R. Isidoro. Real-time depth of field simulation. In W. Engel, editor, *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0.* Wordware, Plano, Texas, 2003.

[19] K. Selgrad, L. Franke, and M. Stamminger. Tiled Depth of Field Splatting. In J. Jorge and M. Lin, editors, *Eurographics 2016 – Posters.* The Eurographics Association, 2016.

[20] K. Selgrad, A. Lier, J. Dörntlein, O. Reiche, and M. Stamminger. A High-Performance Image Processing DSL for Heterogeneous Architectures. In *Proceedings of ELS 2016 9rd European Lisp Symposium*, pages to–appear, New York, NY, USA, 2016. ACM.

[21] K. Selgrad, A. Lier, F. Köferl, M. Stamminger, and D. Lohmann. Lightweight, generative variant exploration for high-performance graphics applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 141–150, New York, NY, USA, 2015. ACM.

[22] K. Selgrad, A. Lier, M. Wittmann, D. Lohmann, and M. Stamminger. Defmacro for C: Lightweight, ad hoc code generation. In *Proceedings of ELS 2014 7rd European Lisp Symposium*, pages 80–87, 2014.

[23] P. Slusallek and I. Georgiev. Rtfact: Generic concepts for flexible and high performance ray tracing. In R. J. Trew, editor, *Proceedings of the IEEE / EG Symposium on Interactive Ray Tracing 2008*, pages 115–122, RT08 Reception Warehouse Grill 4499 Admiralty Way Marina del Rey, CA 90292, 2008. IEEE Computer Society, Eurographics Association, IEEE.

[24] W. Taha. A gentle introduction to multi-stage programming. In *Domain-specific Program Generation, LNCS*, pages 30–50. Springer-Verlag, 2004.

[25] G. Thomas. Compute-Base GPU Particle Systems, 2014. GDC'14.

[26] T. Veldhuizen. Template metaprograms. *C++ Report*, May 1995.

APPENDIX

```
// This is DOF accumulation with the following features:
// - block-load-with-syncthreads
// - peel-odd-blocks
// - node-rgb16-c32-xy16
// - copy-current-entry-to-register
// - synced-blockwise-early-out
// - directly-load-from-global-memory
// - load-alpha
// - front-to-back
// - float-dist

__global__ void kernel_213(int width, int height, int2 tileWH, int2
      tilesizes, dataSpl **lists, float max_coc, uint
      *atomic_counters, ushort4 **hdr_output)
{
        int2 gid = make_int2((blockIdx.x * blockDim.x) + threadIdx.x,
              (blockIdx.y * blockDim.y) + threadIdx.y);
        if ((gid.x >= width) || (gid.y >= height))
                return;
        float2 gidF = make_float2(float(gid.x), float(gid.y));
        int tile_index = (gid.x / tileWH.x) + (tilesizes.x * (gid.y /
              tileWH.y));
        int num_per_tile = (tileWH.x * tileWH.y) +
              ((int)atomic_counters[tile_index]);
        float4 color = make_float4(0, 0, 0, 1);
        int tid = (threadIdx.y * 16) + threadIdx.x;
        __shared__ dataSpl local_data[256];
        float alpha_dest = 1.0f;
        const int elems_per_iter = 256;
        int iterations = num_per_tile / 256;
        int last_iter_width = num_per_tile % 256;
        for(int i = 0; i < iterations; ++i){
                local_data[tid] = lists[tile_index][(i * 256) + tid];
                __syncthreads();
                if (__any(alpha_dest > 0.01f)) {
                        for(int j = 0; j < elems_per_iter; j += 1){
                                dataSpl elem = local_data[j];
                                float dist_x = __half2float(elem.x())
                                      - gidF.x;
                                float dist_y = __half2float(elem.y())
                                      - gidF.y;
                                float coc_sq = elem.sq_coc();
                                if (((dist_x * dist_x) + (dist_y *
                                      dist_y)) <= coc_sq) {
                                        float alpha =
                                              __half2float(elem.w());
                                        color.x += (alpha_dest *
                                              alpha *
                                              __half2float(elem.r()));
                                        color.y += (alpha_dest *
                                              alpha *
                                              __half2float(elem.g()));
                                        color.z += (alpha_dest *
                                              alpha *
                                              __half2float(elem.b()));
                                        alpha_dest = (1.0f - alpha) *
                                              alpha_dest;
                                }
                        }
                }
                else
                        break;
                __syncthreads();
        }
        local_data[tid] = lists[tile_index][(iterations * 256) + tid];
        __syncthreads();
        if (alpha_dest > 0.01f)
                for(int j = 0; j < last_iter_width; j += 1){
                        dataSpl elem = local_data[j];
                        float dist_x = __half2float(elem.x()) -
                              gidF.x;
                        float dist_y = __half2float(elem.y()) -
                              gidF.y;
                        float coc_sq = elem.sq_coc();
                        if (((dist_x * dist_x) + (dist_y * dist_y))
                              <= coc_sq) {
                                float alpha = __half2float(elem.w());
                                color.x += (alpha_dest * alpha *
                                      __half2float(elem.r()));
                                color.y += (alpha_dest * alpha *
                                      __half2float(elem.g()));
                                color.z += (alpha_dest * alpha *
                                      __half2float(elem.b()));
                                alpha_dest = (1.0f - alpha) *
                                      alpha_dest;
                        }
                }
        __syncthreads();
        hdr_output[gid.x][gid.y] = make_half4(color.x / (1.0f -
              alpha_dest), color.y / (1.0f - alpha_dest), color.z /
              (1.0f - alpha_dest), 1.0f);
}
```

Figure 10: C-Mera generated kernel with the best performance on Nvidia GTX 980Ti.

Demonstrations

Accessing local variables during debugging

Michael Raskin
CS Dept., Aarhus University
raskin@mccme.ru*

Nikita Mamardashvili
Moscow Center for Continuous
Mathematical Education

ABSTRACT

Any reasonably large program has to use local variables. It is quite common in the Lisp language family to also allow functions that exist only in a local scope. Scoping rules often allow compilers to optimize away parts of the local environment; doing that is good for performance, but sometimes inconvenient for debugging.

We present a debugging library for Common Lisp that ensures access to the local variables during debugging. To prevent the optimisations from removing access to these variables, we use code-walking macros to store references to the local variables (and functions) inside global variables.

Keywords

Software and its engineering, Software testing and debugging, lexical environment, lexical closures

1. INTRODUCTION

We hope that every program of non-negligible size uses some local variables. Unfortunately, during debugging these variables may be inaccessible because of optimisation. For example, when debugging the following code:

```
(labels ((f (z) (+ z 1))) (let ((x 2))
    (cerror "Continue" "Error invoked") (f x)))
```

in SBCL[5] 1.3.4 (the freshest available at the time of writing) neither x nor f were accessible from the debugger with any combination of safety and debugging declarations.

Lacking access to local variables makes debugging runtime errors significantly less convenient. Also, using a continuable error to get a REPL inside the context of a function is significantly less useful as a debugging and exploration tool if the local variables become inaccessible.

CLISP[6] seems to do the right thing from the debugging point of view, but, unfortunately, many libraries (for example, CLSQL) do not fully support CLISP.

Since searching hasn't revealed a solution to this problem, we implemented a brute-force solution, which became the local-variable-debug-wrapper library[1], presented in the current paper.

1.1 Feature set

The features of the presented library include:
- Access to local variables and functions from the debugger, including the lexical contexts lower on the stack
- Altering local variables during debugging
- A reader trick that allows wrapping the contents of the entire file by adding one line in the beginning

1.2 Example use

The following code illustrates an example file which has wrapping enabled:

```
(use-package :local-variable-debug-wrapper)
; Wrapping to the end of file
(wrap-rest-of-input)
; Inspecting local variables in a function
(defun test-w-1 ()
  (let ((x 1)) (let ((x 3) (y 2)) (pry) (+ 2 3))))
; Debugging a failure
(defun test-w-2 ()
  (let ((x 1)) (let ((x 3) (y 2)) (error "Oops"))))
```

2. TECHNIQUES USED

Currently, we use hu.dwim.walker[2] to annotate the input code (annotations are represented using CLOS objects). The forms whose lexical environment differs from that of their parent form get wrapped in a special call.

At the top level the special call is defined as a local macro by macrolet. It uses the &environment parameter to access lexical environment of each form and the hu.dwim.walker wrapper over implementation-specific lexical environment objects to obtain the names of local bindings.

For example, we get the following expansion:

```
(with-local-wrapper (let ((x 1)) x))
-->
(macrolet
  ((push-lexenv-to-saved-inner (&rest args)
```

*The author acknowledges support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61361136003) for the Sino-Danish Center for the Theory of Interactive Computation and from the Center for Research in Foundations of Electronic Markets (CFEM), supported by the Danish Strategic Research Council.

9th ELS May 9–10, 2016, Kraków, Poland

```
      `(push-lexenv-to-saved ,@args)))
  (progn
    (let ((x (push-lexenv-to-saved-inner 1)))
      (push-lexenv-to-saved-inner x))))
```

We build an alist of local functions and their corresponding names. For variables, we want to allow the user to modify local variables and resume execution. This functionality requires capturing a reference, and we use anonymous functions and lexical closures for that (apparently, there is no safe alternative).

This is performed by push-lexenv-to-saved. It is a macro using an &environment parameter. For examples, one of its calls expands as follows:

```
(push-lexenv-to-saved-inner x)
-->
(let ((*saved-lexenvs*
       (cons (list :variables (list (cons 'x
        (lambda (&optional (value nil value-given))
            (if value-given (setf x value) x)))))
            *saved-lexenvs*)))    x)
```

To make inspecting a saved lexical environment easier we provide the pry macro that creates the dynamic variables with the same names as the lexical local variables in the environment under consideration. The dynamic extent of the created variables is limited to the pry call, so they affect the debugging session but not the semantics of the program after continuation.

This function is named after the Pry REPL[3] for Ruby, seeing as how not only is it aimed towards the same use case, but the original inquiry that motivated the development was finding a Common Lisp equivalent for that very library.

We also provide lower-level functions for accessing the saved environments, and some other convenience helpers.

To make wrapping all the forms in a file easier, we provide (wrap-rest-of-input) functionality: wrap-rest-of-input clones the readtable and makes (a macro-character. The corresponding reader function immediately reverts to the previous readtable, calls unread-char on the opening parenthesis, and reads a form; afterwards the form is put inside the with-local-wrapper.

3. EVALUATION

To test a bad case, we used a very inefficient Fibonacci number calculation:

```
(defparameter *pry-on-bottom* nil)
(defun fib-uw (n)
  (if (<= n 1)
    (progn (when *pry-on-bottom* (pry)) 1)
    (+ (fib-uw (- n 1)) (fib-uw (- n 2)))))
```

We ran the same code wrapped and unwrapped, recording time and memory. The tests were run on a 4-core i7-4770R.

On SBCL, each function call consed 16 bytes when unwrapped and 128 bytes when wrapped. For large parameter values the wrapped version was approximately 4 times slower than the unwrapped one.

CCL ran the unwrapped test slightly faster than SBCL, but the wrapped version was slower than on SBCL. For large parameter values the slowdown was slightly below 9 times.

CLISP does provide full access to local variables on its own in most cases, but this implementation is not currently supported by hu.dwim.walker. The unwrapped version runs 80 times slower than on CCL .

An example run for $n = 40$ gave the following results (values in parenthesis are relative to the unwrapped code on the same implementation):

	time, s	slower	bytes consed
		than CCL	per call
CCL	3.18		16
SBCL	3.50	1.10×	16
CLISP	259.90	81.68×	0
CCL w/w	28.29 (8.89×)	8.89×	160 (+144)
SBCL w/w	13.95 (3.98×)	4.38×	128 (+112)

3.1 Known limitations

To modify the bindings used outside the pry session one has to use the low-level local-variable macro.

The library currently doesn't provide access to local macros.

The limitations of wrapping a piece of code in progn apply (note that wrap-rest-of-input wraps each form separately).

Macro definitions can't be wrapped. This is due to limitations of hu.dwim.walker. If all the macro definitions are top-level defmacro forms wrap-rest-of-input will do the right thing.

Portability is limited by the hu.dwim.walker package. Currently the library is known to be usable on SBCL and CCL and broken on ECL and CLISP.

3.2 Conclusion

Our testing shows that the wrapper provides reliable access to local variables.

We think that this library can make debugging easier in many cases. Impossibility to enforce local variable availability seems surprising (and confusing) to newcomers; we hope that our work can make Lisp slightly more accessible for programmers coming from other languages.

We plan to fix some of the limitations in future. We will be grateful for pointing out corner cases that we have missed.

4. ACKNOWLEDGEMENTS

We thank an anonymous reviewer who provided a lot of valuable advice for improving the present article.

5. REFERENCES

[1] local-variable-debug-wrapper homepage. Retrieved on 22 April 2016. https://gitlab.common-lisp.net/mraskin/\\local-variable-debug-wrapper

[2] hu.dwim.walker package. Retrieved on 15 April 2016. http://dwim.hu/darcsweb/darcsweb.cgi?\\r=LIVE\%20hu.dwim.walker;a=summary

[3] PRY REPL for Ruby. Retrieved on 15 April 2016. http://pryrepl.org/

[4] ANSI Common Lisp Specification, ANSI/X3.226-1994. American National Standards Institute, 1994.

[5] Steel Bank Common Lisp homepage. Retrieved on 15 April 2016. http://www.sbcl.org/

[6] GNU CLISP homepage. Retrieved on 15 April 2016. http://www.clisp.org/

Building Common Lisp programs using Bazel

or Correct, Fast, Deterministic Builds for Lisp

James Y. Knight
Google
jyknight@google.com

François-René Rideau
Google
tunes@google.com

Andrzej Walczak
Google
czak@google.com

ABSTRACT

We will demonstrate how to build Common Lisp programs using Bazel, Google's hermetic and reproducible build system. Unlike the state of the art so far for building Lisp programs, Bazel ensures that incremental builds are always both fast and correct. With Bazel, one can statically link C libraries into the SBCL runtime, making the executable file self-contained.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques

Keywords

Bazel, Build System, Common Lisp, Determinism, Hermeticity, Reproducibility

1. INTRODUCTION

Common Lisp, a general-purpose programming language, is used at Google for the well-known low-fare search engine QPX (de Marcken 2001). Google now builds its Lisp code incrementally, using Bazel, its recently open-sourced scalable build system.

Bazel is designed to build software in a reproducible and hermetic way. Hermeticity means all build dependencies, including build tools such as compilers, are kept under source control. Bazel can thus assess what was or wasn't modified, and either rebuild or reuse cached build artifacts. Reproducibility means building the same target multiple times from the same source code produces the same output. Reproducibility facilitates testing and debugging production code. Bazel further enforces determinism by executing each build action in a container wherein only declared inputs may be read, and any non-declared output is discarded. Bazel can thus parallelize build actions, either locally or to remote workers. Bazel supports writing software in a mix of languages notably including C++, Java, Python, Go and Javascript. Compilers and other build tools must be tuned to remove sources of non-determinism such as timestamps, PRNG seeds, etc.

While mainly written in Java, Bazel is extensible using *Skylark*, a subset of Python with strict limits on side-effects. We used Skylark to add support for building software written in Common Lisp.

9th ELS May 9–10, Kraków, Poland

2. PREVIOUS WORK

The state of the art so far for building large Common Lisp applications is ASDF (Rideau 2014). A descendent of the original Lisp DEFSYSTEM from the 1970s, ASDF builds all the code in the current Lisp image; incremental builds may therefore be affected by all kinds of potential side-effects in the current image; and to guarantee a deterministic build one has to build from scratch. ASDF also lacks good support for multi-language software. An attempt to build Lisp code deterministically, XCVB (Brody 2009), failed for social and technical reasons, though it had a working prototype.

Meanwhile, QPX was built using an ad-hoc script loading hundreds of source files before compiling them and reloading the resulting FASLs. The multi-stage build was necessary because of circular dependencies between the files; the dependencies formed a big "hairball" which any replacement build solution had to handle.

3. BUILDING LISP CODE WITH BAZEL

The input to Bazel is a set of `BUILD` files with Python-like syntax, that declaratively specify software *targets* using *rules*.

Our first Lisp support function, `lisp_library`, declares an intermediate target which can be referenced from other Lisp rules. With SBCL (Steel Bank Common Lisp), `lisp_library` creates a FASt Load (FASL) archive by concatenating the FASL files produced by compiling each of its Lisp sources.

Bazel users specify *attributes* when defining rules. For Lisp rules, these include the Lisp sources `srcs`, Lisp libraries `deps`, C sources `csrcs` and C libraries `cdeps`, auxiliary `data` available at runtime to all executable targets depending on the library, and auxiliary `compile_data` available at compile-time.

Lisp rules have additional build options. The `order` attribute notably specifies a build strategy; the default `"serial"` order loads each source file in sequence before compiling the next file. The `"parallel"` order compiles all files in parallel without loading other ones. The `"multipass"` order first loads all source files, then compiles each one separately in parallel, which is useful to handle a "hairball" aggregate.

```
load("@lisp__bazel//:bazel/rules.bzl",
     "lisp_library")
lisp_library(
    name = "alexandria",
    srcs = ["package.lisp",
        # ...
        "io.lisp"],
    visibility = ["//visibility:public"])
```

The above example is from the `BUILD` file for the "alexandria" general utility library. Bazel first loads `lisp_library` from its conventional *build label* under the `lisp__bazel` "external repository". The `visibility` attribute indicates which Bazel packages are allowed to reference the rule's target — in this case, all packages.

The following command builds `alexandria.fasl` and makes it available at a well defined path:

```
    bazel build :alexandria
```

Our second function, `lisp_binary`, creates an executable including both Lisp runtime and Lisp core image. If Lisp or C sources are specified, they will be compiled into corresponding Lisp and C components before being statically linked into the final binary. Our third function, `lisp_test`, is a variation on the `lisp_binary` rule meant to be invoked with the `bazel test` command.

```
  load("@lisp__bazel//:bazel/rules.bzl",
       "lisp_binary")
  lisp_binary(
    name = "myapp",
    srcs = ["myapp.lisp"],
    main = "myapp:main",
    deps = [
      "@lisp__alexandria//:alexandria"])
```

This `BUILD` file contains a `lisp_binary` target which references the "alexandria" `BUILD` target seen before. When running the binary, the Lisp function `myapp:main` will be called with no arguments at startup. The program may be compiled and executed using:

```
    bazel run :myapp
```

A `lisp_binary` can directly or transitively depend on C or C++ libraries. Static linking of the libraries makes it more reliable to deploy such a binary on multiple hosts in the cloud, without the opportunity to get library dependencies wrong; in particular, it helps with minimizing discrepancies between test and production environments. C and C++ dependencies can be specified via the `cdeps` rule attribute, which can refer to any `cc_library` built with Bazel. The `csrcs` and `copts` rule attributes allow to directly specify C source files for which an internal target will be generated.

Thanks to these build rules, the duration of the incremental QPX build went from about 15 minutes to about 90 seconds, with qualitative effects on developer experience. However, this is for a large project, using a computing cloud for compilation. The open source version of Bazel currently lacks the ability to distribute builds, but it can already take advantage of multiple cores on a single machine. The typical Lisp user will therefore not experience as large a speedup when using the Bazel lisp rules.

4. INSIDE THE LISP RULES

Lisp support was implemented using Bazel's *Skylark* extension language. The functions `lisp_library`, `lisp_binary` and `lisp_test` are Skylark *macros* calling internal implementation *rules*. A Skylark *macro* is basically a Python function that is executed by Bazel at the time the `BUILD` file is loaded and invokes the actual Skylark rules as side-effects. A Skylark *rule* consists of an implementation function and a list of attribute specifications that notably define type-checked inputs and outputs for the rule's target. The Lisp support macros establish two separate graphs for each of the Lisp and C parts of the build, that are connected at the final binary targets. Thus, the `lisp_library` macro calls the `_lisp_library` rule to create Lisp related actions, and also calls the `make_cdeps_library` macro to create the C related targets using Skylark's `native.cc_library`.

The rules compile C sources and Lisp sources in parallel to each other, and the resulting compilation outputs are combined together in the last step. This improves the build parallelism and reduces the latency. In order to facilitate linking, all C symbols referred to at Lisp-compilation time are dumped into a linker script file. The final linking step uses that `.lds` file to include from C libraries only the referenced objects, and to statically detect any missing or misspelled C symbol. The `lisp_binary` and the `lisp_test` macros then create the executable by combining the runtime of SBCL linked with additional C libraries and a core image dumped after loading all FASLs.

The `_lisp_library` rule implementation compiles each file in a new Lisp process (possibly on remote worker machines) that will first `load` all the Lisp *source* files from all the transitive dependencies as well as relevant files from the current rule (depending on the `order` attribute). This is faster than loading FASLs, thanks to SBCL's `fasteval` interpreter (Katzman 2015), written specifically to speed up building with Bazel; this also enables all compile actions to run in parallel, whereas loading FASLs would introduce long chains of dependencies between actions and cause high latency. On the downside, some source files must be fixed to add missing `:execute` situations in `eval-when` forms, and optionally to explicitly `compile` computation-intensive functions used at compile-time.

The `_lisp_library` implementation also returns a *provider* structure containing transitive information about: all Lisp sources loaded; all reader features declared; runtime and compile-time data for each library; FASLs from each `lisp_library` target, used when linking the final binary target; deferred warnings from each compilation — mostly for undefined functions — checked after all FASLs have been loaded into the final target.

5. REQUIREMENTS

The Lisp support for Bazel so far only works on SBCL, on the x86-64 architecture, on Linux and MacOS X. It shouldn't be hard to make it work on a platform supported by both SBCL and Bazel. However, porting to another Lisp implementation will be non-trivial, notably with respect to linking C libraries statically or to achieving latency as low as with the `fasteval` interpreter.

Bazel itself is an application written in Java. It takes seconds to start for the first time; then it becomes a server that can start an incremental build instantly but consumes gigabytes of memory.

6. CONCLUSION AND FUTURE WORK

We have demonstrated simultaneously how Common Lisp applications can be built in a fast and robust way, and how Bazel can be extended to reasonably support a new language unforeseen by its authors. Bazel may not be a lightweight solution for writing small programs in Lisp; but it is a proven solution for building large industrial software projects tended by several groups of developers using multiple programming languages, including Lisp.

In the future, we may want to add Lisp-side support for interactively controlling Bazel: we would like to be able to build code, and load the result code into the current image, without reloading unmodified FASLs and object files.

Our code is at: `http://github.com/qitab/bazelisp`

Bibliography

François-René Rideau and Spencer Brody. XCVB: an eXtensible Component Verifier and Builder for Common Lisp. 2009.

Carl de Marcken. Carl de Marcken: Inside Orbitz. 2001. http://www.paulgraham.com/carl.html

Douglas Katzman. SBCL's Fasteval interpreter. 2015.

François-René Rideau. ASDF3, or Why Lisp is Now an Acceptable Scripting Language (extended version). 2014.

Fast Interactive Functional Computer Vision with Racket

A Demonstration using Racket and the Kinect Sensor

Benjamin Seppke
University of Hamburg
Dept. Informatics
Vogt-Kölln-Str. 30
22527 Hamburg, Germany
seppke@informatik.uni-hamburg.de

Leonie Dreschler-Fischer
University of Hamburg
Dept. Informatics
Vogt-Kölln-Str. 30
22527 Hamburg, Germany
dreschler@informatik.uni-hamburg.de

ABSTRACT

Functional programming languages, like Lisp or Racket are known to be general purpose languages with a steep learning curve and wide range of applications. They can be used interactively to solve problems and have inspired other comparably new languages with respect to functional extensions (e.g. Python or Swift). In this work, we will demonstrate the use of the Racket programming language with respect to fast interactive computer vision. Based on the VIGRACKET module, which combines the best of the compiled and interactive worlds with respect to common tasks in computer vision, Racket and the VIGRA C++ library (see [4]), we will present a (near-) realtime computer vision demo. For this demo we have selected the Microsoft Kinect Sensor as the continuous the image source. We present the connection to the sensor by means of image transfer to Racket and a case study: a natural pointer interface for human computer interaction.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming—*Allegro Common Lisp, SBCL, Racket*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces, software libraries*; I.4.8 [**Image Processing and computer vision**]: Scene Analysis

1. INTRODUCTION

In [4] we have presented how well functional programming and computer vision approaches may be combined by means of the VIGRACKET library. Like other state-of-the-art interactive development environments, functional language interpreters natively offer an interactive development cycle, generic modeling and even powerful garbage collectors. So, instead of using a new language with functional extensions, like Swift, Python or others, why not simply use well-known functional languages like Lisp or Racket?

The main aim of this paper is to demonstrate the use of the extended VIGRACKET module with respect to the Kinect sensor as a data source. Since the VIGRACKET module has been developed to introduce computer vision to Racket, neither to support interactive realtime processing nor to support fast functional development interfaces or fast visualizations, some optimizations were additionally needed. For sake of clarity of this demo, these optimization steps can be found elsewhere (see [3]). Herein, we present how to connect with the Kinect sensor to Racket, acquire images and to use the acquired images by means of an interactive natural pointing device interface.

2. PRELIMINARIES

To run the demonstration, some preliminaries need to be fulfilled. Besides Racket, the VIGRACKET module needs to be downloaded and installed from the author's GitHub account: `https://github.com/bseppke`. Depending on the target's operating system, this module may require additional dependencies, which are described in [4].

Since the VIGRACKET module is not an acquisition library, it does not support real-time acquisition devices. For this demonstration, we have selected the Microsoft Kinect sensor as the acquisition device, and the OpenKinect libfreenect library for accessing the sensor. Unfortunately, this library does provide interaction layers for Python and Java, but none for Racket (see [5]). Based on a low level USB-interface (via the libusb), the libfreenect library allows access to all the necessary data. In order to avoid concurrent asynchronous calls and callbacks, we use the synchronous grabbing access API. Finally, it is necessary to download and install the "rackinect" Module from the same GitHub account as mentioned above.

3. CONNECTION TO THE KINECT

Since the image formats of the raw data, which are used by the libfreenect library, are not compatible with the image representation of the VIGRACKET module, we need to introduce another small Racket/C-wrapper, to which we refer to as "rackinect". On the Racket side of this wrapper we define grabbing functions for the acquisition of the raw data. The function `(grabdepth)` grabs the depth data (in mm) and stores it inside a newly allocated one channel VIGRACKET image. The function `(grabvideo)` grabs the current RGB image and stores it by means of a new three channel VIGRACKET image. Finally the `(grabdepth+video)` grabs both data and stores it by means of new a four chan-

9th ELS May 9–10, 2016, Kraków, Poland

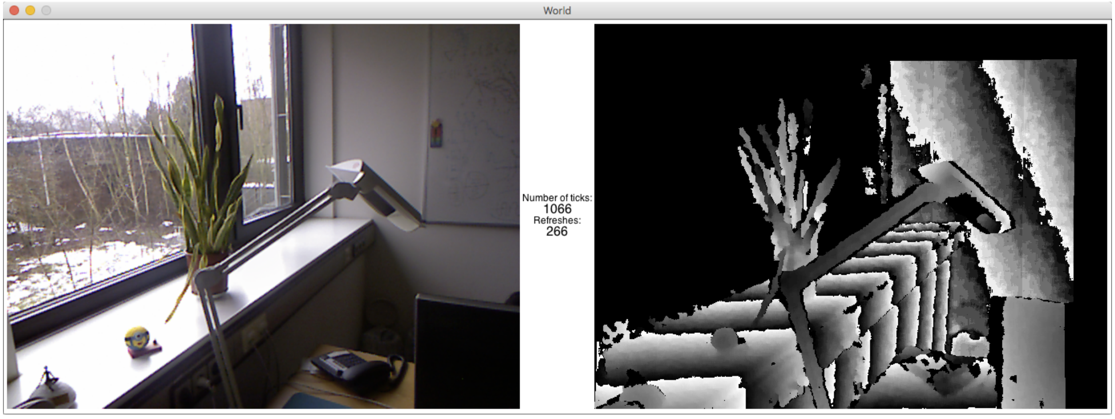


Figure 1: Image taken out of the sequence we get by calling (animate live-view-combined). Left: RGB image, center: current tick and refresh counter, right: depth image. Since the depth values are normalized to millimeters, there is one overflow each 25.5 cm. Black parts of the depth image denote unknown depth.

nel VIGRACKET image, where the first channel contains the depth data (in mm) and the last most three channels contain the RGB data.

As a first demonstration, we present the use of both functions for a near real-time depth and video display GUI. Here we use the animate functionality of Racket's `2htdp/universe` module (see [1]). It takes a function with one argument and a Racket bitmap as a result type and calls the given function 28 times per second using an increasing argument `t` and updates the images every fourth frame. An example output is shown in Fig. 1.

```
(define dp (grabdepth+video))
(define d  (image->bitmap (list (car dp))))
(define p  (image->bitmap (cdr dp)))

(define (live-view-combined t [update_each 4])
  (when (= (modulo t update_each) 0)
     (begin
        (set! dp (grabdepth+video))
        (image->bitmap (list (car dp)) d)
        (image->bitmap (cdr dp)        p)))
  (beside p (status t update_each) d))

(animate liveview-combined)
```

4. NATURAL POINTER INTERFACE

The computer mouse has probably been the most common pointing device for human-computer interaction for decades. Due to the electronic mobile revolution with tablets and smartphones, we are now able to use our fingers directly as pointing devices, e.g. by means of (multi-) touch displays. Although this is closer to the “natural pointing” metaphor, it is still artificial and might not be well applicable for general virtual environments. The main limitation is the two-dimensional interface, since virtual worlds are usually not as flat as the displays' touch surfaces.

The Microsoft Kinect sensor with its depth image stream provides another alternative for a more natural pointer interface by tracking your fingers movements (cf. [2]). To simplify the finger detection, we make the following assumptions for this case study:

1. Only a certain range of the depth data is allowed to contain the image of the finger.
2. A finger pointer is defined as the top- and left-most point, which is found within this range.

Under these assumptions, it is quite clear, that we will detect one pointer position if any object is inside the depth interval of interest. Since it is the top- and left-most position, it is not necessary, that the finger is put in front of other objects, but above them. The first necessary step is to threshold the raw depth image accordingly to the range of the depth interval.

```
(define (closerThan depth_img [t 800])
  (image-map!
     (lambda (val) (if (< 0 val t) 255.0 0.0))
     depth_img))
```

In this function we are able to use the in-place method `image-map!` to save allocation costs, because the original depth values are no longer needed in the further processing. To find the top-left part of the thresholded depth image, we may apply the function above while the result is true. The first false coordinate denotes the pointer position. Experiments have shown, that this approach works stable at about 5 frames per second (see [3]).

5. REFERENCES

[1] M. Felleisen. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.

[2] P. Premaratne. *Human Computer Interaction Using Hand Gestures*. Cognitive Science and Technology. Springer Singapore, 2014.

[3] B. Seppke. Near-realtime computer vision with racket and the kinect sensor. Technical report, University of Hamburg, Dept. Informatics, 2016.

[4] B. Seppke and L. Dreschler-Fischer. Efficient applicative programming environments for computer vision applications: Integration and use of the vigra library in racket. In *Proceedings of the 8th European Lisp Symposium*, 2015.

[5] The OpenKinect team. The OpenKinect project. Retrieved February 19, 2016 from: https://openkinect.org.

An Inferred System Description Facility

James Anderson
Datagraph GmbH
Berlin

ABSTRACT

A ”continuous integration” program development process involves interpreting dependency relations

- between program components respective the given development step, and
- between test and program components.

This essay describes how to capture these relations among the entities of a Lisp system without a reified system description, but instead in a language which allows one to build, test and deliver such systems by inferring the system management operations from that information which is implicit in the source code, apparent in the state of the source artefacts, or to be captured introspectively in the running system.

Categories and Subject Descriptors

[**Information Systems Applications**]: Query Languages, Information systems, Resource Description Framework

1. INTRODUCTION

Any system which automates a program development process will base its actions on a model for the relations among the program's components. The precedent for Lisp development has been to construct the model from a system definition document and to manage and interpret the model with functions implemented in the host language.[2][6]

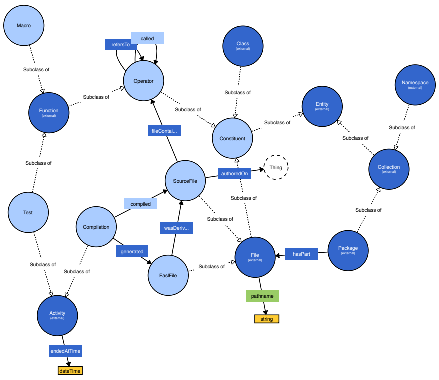


Figure 1: The system description ontology

2. AN IMPLICIT SYSTEM DESCRIPTION

It is possible to achieve the same ends through an alternative approach which relies on RDF[4][3] to capture the description as a graph model and expresses logic to derive build and test plans from the model in the RDF query language, SPARQL[1]. The description in such a model is in terms of an ontology derived from the W3C provenance ontology[5] and extended to refer to entity classes and properties in accord with the terms used for source code management. Within this ontology, one might describe a very simple system, such as

```
example
|-- file1.lisp
|-- file2.lisp
|-- file3.lisp
|-- package.lisp
\-- test.lisp
```

with content, such as

```
(in-package :example)
(defun function3 (arg)
  (+ (function3-part1 arg) (function3-part2 arg)))
(defun function3-part1 (arg)
  arg)
(defun function3-part2 (arg)
  (function2 arg))
```

would yield a graph model which would include,

```
@prefix ssd: <http://dydra.com/schema/ssd#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
ssd:Function#example:function3-part2
  rdf:type ssd:Function ;
  ssd#referenced
   ssd:System#example/Function#example:function2 .
ssd:Pathname#/source/example/file3.lisp
 a ssd:SourceFile ;
 ssd:defines
  ssd:Function#example:function3-part2 .
```

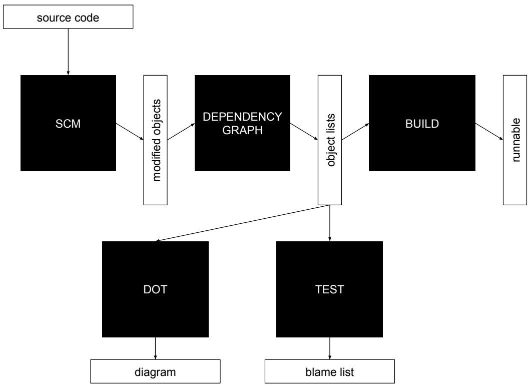
9th ELS May 9–10, 2016, Kraków, Poland

```
ssd:Pathname#/source/example/file2.lisp
 a ssd:SourceFile ;
 ssd:defines
  ssd:Function#example:function2 .
```

The representation offers in itself no great advantage over the various system definition formats which have evolved over the past decades. The benefit derives from the means which can be applied to such models to infer and generate plans for such operations as

- dependency-directed builds
- modification-directed builds
- dependency-directed testing
- system visualization
- version-specific regression testing.

It permits management facilities, in which the components are decoupled through the mediation of a store, the core management logic is articulated as a clear collection of queries, and additional presentation and analysis facilities can be easily applied to the stored system model.



For example, it is possible to introduce rules to infer indirect dependency relations to render them apparent for path navigation:

```
prefix ssd: <http://dydra.com/schema/ssd#>
construct {
 ?file1 ssd:requiredBy ?file2
} where {
  ?file1 ssd:defines ?function1 .
  ?file2 ssd:referenced ?function1 .
}
```

3. DEPENDENCY-DIRECTED BUILDING

Given the RDF model to describe the relations among operators and between operators and source code entities in terms of the illustrated ontology, it is possible to formulate a query which infers the load process from the elementary relations in the following manner.

```
prefix ssd: <http://dydra.com/schema/ssd#>
prefix prov <http://www.w3.org/ns/prov#>
prefix pav <http://purl.org/pav/>
select ?loadPathname
where {
  ?source ssd:requiredBy* ?dependentSource .
  ?source pav:authoredOn ?timeModified .
  filter ( ?timeModified > ?SYSTEM_LOAD_TIME )
  ?dependentSource ssd:pathname ?sourcePathname .
  ?dependentSource pav:authoredOn ?sourceTime .
  optional {
    [] prov:used ?dependentSource ;
       prov:generated ?dependentBinary ;
       prov:endedAtTime  ?binaryTime .
    ?dependentBinary ssd:pathname ?binaryPathname .
    }
  }
  bind (if( bound(?binaryTime),
            if ((?binaryTime > ?sourceTime),
                 ?binaryPathname, ?sourcePathname ),
            ?sourcePathname )
        as ?loadPathname )
```

The result is a list of those files which depend on any file which has been modified subsequent to the last known load, in an order which corresponds to their degree of dependency on modified source files. In a similar manner, based on a graph which captures `code:refersTo` relations from an initial test run of an instrumented system, it is possible to infer the test complement sufficient to cover a particular set of source code changes.

```
prefix ssd: <http://dydra.com/schema/ssd#>
prefix prov <http://www.w3.org/ns/prov#>
prefix pav <http://purl.org/pav/>
select ?test
where {
  ?test code:refersTo+ ?function .
  ?source code:fileContainsDefOf ?function .
  ?source pav#authoredOn ?timeModified .
  filter ( ?timeModified > ?SYSTEM_TEST_TIME )
  }
```

4. REFERENCES

[1] W. W. W. Consortium et al. Sparql 1.1 overview. 2013.

[2] M. Kantrowitz. Portable utilities for common lisp. user guide & implementation. 1991.

[3] G. Klyne and J. J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2006.

[4] O. Lassila and R. R. Swick. Resource description framework (rdf) model and syntax specification. 1999.

[5] T. Lebo, S. Sahoo, D. McGuinness, K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, and J. Zhao. Prov-o: The prov ontology. *W3C Recommendation*, 30, 2013.

[6] F.-R. Rideau and R. P. Goldman. Asdf 2: Evolving an api to improve social interactions.

Distributed High Performance Computing in Common Lisp

Marco Heisig
Chair for Applied Mathematics 3
FAU Erlangen-Nürnberg
Cauerstraße 11
91058 Erlangen
marco.heisig@fau.de

Dr. Nicolas Neuss
Chair for Applied Mathematics 3
FAU Erlangen-Nürnberg
Cauerstraße 11
91058 Erlangen
neuss@math.fau.de

ABSTRACT

High Performance Computing (HPC) is the Formula 1 of computer programming and deals with the solution of problems that have a practically inexhaustible demand for compute resources, e.g. accurate physics simulations. We present the library CL-MPI and how it can be applied to distribute scientific computing workloads over several Lisp images and computers.

In particular, we describe our work towards the parallelization of FEMLISP and the tools and techniques developed in the process. FEMLISP is a Common Lisp framework for solving partial differential equations using the finite element method.

Keywords

MPI, distributed computing, parallelization

1. INTRODUCTION

The Message Passing Interface[6] (MPI) is the de facto standard for distributed programming on all modern compute clusters and supercomputers. It features a large number of communication patterns with virtually no overhead. Our work on bringing MPI functionality to Common Lisp resulted in vast improvements to the message passing library CL-MPI [1] and the development of several new approaches to distributed computing.

2. DISTRIBUTED COMPUTING WITH CL-MPI

The primary function of CL-MPI is to transmit subsequences of simple-arrays between collaborating processes. The dynamic type system and the condition system of Common Lisp allowed for an API that is much cleaner and less error-prone than the corresponding native interfaces to C or Fortran as shown in Figure 1. Care was taken that the convenience that is gained by automatically converting and checking all arguments has little impact on the computational cost. In our measurements the transfer time for a single message from one CPU core to another is below 500 nanoseconds (SBCL 1.2.4, OpenMPI 1.6, Intel i7-5500U @ 2.40GHz).

```
ierr = MPI_Recv(&buffer, count, MPI_INT, src, tag,
                MPI_COMM_WORLD, &status);

(mpi-recv buffer source) ; oh the clarity!
```

Figure 1: Comparing MPI bindings of C and Lisp

9th ELS May 9–10, 2016, Kraków, Poland

2.1 Getting Started

An MPI application is launched by a call to `mpiexec -np N EXECUTABLE`, which launches `N` identical processes on one or more interconnected machines. To register as an MPI application, each process has to call `(mpi-init)` initially and end with a call to `(mpi-finalize)`. The individual processes can use the commands `(mpi-comm-rank)` and `(mpi-comm-size)` to obtain their unique process rank and the total number of processes, respectively. Given those numbers, it is possible to partition given tasks and compute parts of them independently. The commands `(mpi-recv BUF RANK)` on the receiving side and `(mpi-send BUF RANK)` on the sending side can be used to communicate data where necessary.

The previous six commands are already sufficient for successful MPI programming. Figure 2 shows an implementation of the game "whisper down the lane" to illustrate the concepts. The `static-vectors` [2] package is used to allocate arrays with fixed memory locations.

It is recommended to use a utility like cl-launch or Roswell to create a standalone Lisp executable before launching it with `mpiexec`. A starting point to MPI programming is [4].

2.2 Advanced Usage

Now that MPI is accessible to Lisp, it is possible to explore new programming models for parallel computation in a rapid fashion. Most prominently we have implemented a distributed REPL, where commands are first broadcast to all processes and then evaluated in parallel. This approach is flexible, but erroneous communicating commands quickly lead to deadlocks of all processes. We conclude that a higher level of abstraction is necessary if one attempts to bring interactive computing to a distributed system.

```
(mpi-init)
(let* ((rank (mpi-comm-rank))
       (size (mpi-comm-size))
       (left-neighbor  (mod (- rank 1) size))
       (right-neighbor (mod (+ rank 1) size)))
  (with-static-vector (buffer 14
                       :element-type 'character)
    (cond ((= 0 rank)
           (mpi-send "secret message" right-neighbor)
           (mpi-recv buffer left-neighbor))
          (t
           (mpi-recv buffer left-neighbor)
           (mpi-send buffer right-neighbor)))))
(mpi-finalize))
```

Figure 2: MPI ring, a.k.a whisper down the lane

3. DYNAMIC DISTRIBUTED OBJECTS

As a next step towards productive distributed high performance computing, we introduce *dynamic distributed objects* (short: DDO). These are regular CLOS objects that can be shared across multiple processes.

3.1 Mechanics

Local modifications to a DDO are registered. A synchronisation command can be issued, after which all DDOs with local modifications broadcast their state to all other owners of that object. Generic functions are invoked to resolve any colliding modifications.

DDOs are not excluded from garbage collection. If an object is locally garbage collected, the process broadcasts that it is no longer an owner. This way garbage collection works even across process boundaries.

3.2 Implementation

Each DDO on each process has a locally unique id. The three place relation between the local id, the rank of a neighbor process and the id of the same object on this neighbor process is stored in red-black binary trees [5]. The mapping from local ids to the actual objects is done via a weak hash table to not interfere with regular garbage collection.

The synchronization of objects occurs in two steps. First, all processes exchange a list of modifications they want to broadcast. In the second step, the modifications of the individual objects are transferred with point-to-point communication.

4. APPLICATION TO FEMLISP

Finite Elements (FE) are a particularly successful method for solving partial differential equations, which, in turn model many important everyday problems. Since our world is three-dimensional or (including time) four-dimensional, discretizing the continuum usually leads to very large discrete problems, for which the solution on parallel architectures becomes a necessity.

Our library Femlisp [3] is a FE framework which is written completely in Common Lisp. Of course, this gives all the benefits one expects from CL, for example, compact and flexible code as well as interactivity. This flexibility is also the main reason, why Femlisp offers a comparatively large number of features (e.g. unstructured meshes in arbitrary space dimensions, arbitrary approximation order) with relatively little source code.

However, up to now, Femlisp has been a serial program, which, as indicated above, is a major impediment for most applications in scientific computing. Only recently, the situation has changed in such a way that a large part of Femlisp became thread-safe so that multi-threading can be used for some performance-critical sections.

Using the libraries CL-MPI and DDO, we have an additional level of parallelism available. The integration with Femlisp is a work in progress, but important parts (refinement, and discretization) are already parallelized. Only very little additional code was needed for achieving this, and we are very confident that this will be the case for the solver as well.

As a simple and already working example, we show timings for assembling matrix and right-hand side for a diffusion problem on a three-dimensional cube using a mesh with $4096 = 16^3$ cells and an approximation order of 5, which corresponds to more than 550,000 scalar unknowns.

processes × threads	1×1	1×6	2×6	4×6
time (sec)	188	34	20	10
speedup	-		1.7	3.4

We observe almost optimal speedup, which is to be expected because the communication overhead for discretization involves only unknowns lying on lower-dimensional interfaces, and, therefore, the number of communicated data is of lower order compared with the total amount of data.

5. CONCLUSIONS

A lot of effort is currently put into the development of new parallel computing paradigms, as the traditional methods are no longer feasible on machines with several petaFLOPS[1]. We believe that Common Lisp is an excellent vessel for such kinds of exploratory programming and will continue our research.

6. ACKNOWLEDGMENTS

We would like to thank the KONWIHR project (http://www.konwihr.uni-erlangen.de/about-konwihr.shtml) and the Chair for Applied Mathematics 3 under Prof. Eberhard Bänsch for supporting our work.

7. REFERENCES

[1] https://github.com/marcoheisig/cl-mpi.

[2] https://github.com/sionescu/static-vectors.

[3] http://www.femlisp.org/.

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface.* MIT Press, third edition, 2014.

[5] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:8–21, 1978.

[6] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0.* High Performance Computing Center Stuttgart (HLRS), 2012.

[1] Floating-Point Operations per Second