FRIEDRICH-ALEXANDER-UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT ▪ DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Petalisp — Design and Implementation

Marco Heisig

Master's Thesis

Petalisp — Design and Implementation

Marco Heisig

Master's Thesis

Aufgabensteller:	Dr.-Ing. habil. Harald Köstler
Betreuer 1:	M. Sc. Sebastian Kuckuk
Betreuer 2:	Dr. Nicolas Neuß
Bearbeitungszeitraum:	14.07.2016 – 22.12.2016

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Masterarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 22.12.2016 .....

Abstract

This thesis presents a novel approach to High-Performance Computing by embedding a small, functional programming language with high data parallelism into the existing general purpose language Common Lisp.

Contents

List of Figures

1 Introduction

Parallel computers become evermore ubiquitous. In 1997, ASCI Red was the world's first terraflop computer, which means it could execute 10^{12} double-precision floating-point operations per second. It had 7264 CPU cores. In June 2008, the Los Alamos National Laboratory presented Roadrunner, the world's first petaflop system, with 122,400 CPU cores. At the time of writing, in 2016, the fastest computer is the Sunway TaihuLight, which reaches 93 petaflops and has a staggering 10,649,600 CPU cores. Yet the predominant programming languages on these systems are Fortran and C. When these languages were developed, the fastest computer imaginable was something like the the Cray-1 with 160 megaflops on a single CPU core. It is not surprising that these languages have poor support for parallel programming.

A wide range of tools have been developed to mitigate the shortcomings of C and Fortran with respect to parallel programming, e.g. OpenMP, threading building blocks, Cilk, Coarray Fortran and partitioned global address space models. All these tools have in common that the compiler of such code has only rudimentary knowledge of the runtime behavior. Consequently many optimizations have to be carried out by hand. Changes of algorithms, data structures or the underlying hardware require that some of this manual tuning has to be reapplied. Such manual optimization is extremely time consuming and error prone. Under these circumstances it seems promising to create a new, vastly simplified programming language that guarantees that its programs can be fully optimized by a machine. Petalisp is such a language.

1.1 High Performance Computing

High Performance Computing (HPC) is the process of designing, constructing and using supercomputers to reason about scientific problems. Several categories of algorithms play a dominant role in HPC and were memorably named "The seven dwarfs of HPC"[1] by Phillip Colella in a presentation in 2004:

1. **dense linear algebra** — operations on arbitrary matrices and vectors. The memory access patterns are predictable, most time is spent in computation. Examples are the Choleski decomposition, Gaussian elimination or dense matrix multiplication.

2. **sparse linear algebra** — matrices and vectors that contain mostly zeroes. Data is stored in a compressed format and communication plays an important role. Typical algorithms in this category are Conjugate Gradient methods, Krylov subspace methods, or sparse matrix vector multiplications.

3. **spectral methods** — conversions between the frequency domain and the temporal/spatial domain, usually via fast Fourier transformations.

4. **n-body methods** — simulations of many discrete particles with pairwise interactions. Primary concerns are domain decomposition and nearest neighbor search, e.g. via the Barnes-Hut algorithm.

5. **structured grids** — methods on highly regular domains, that can essentially be stored as contiguous arrays. The simple structure of these problems allows for high optimization.

6. **unstructured grids** — data is stored as an undirected graph, which allows to model complicated domains. Such problems arise frequently in Finite-Element or Finite-Volume methods.

[1] As it happens with categorizations, there are meanwhile at least 13 dwarfs, ruining the analogy with the famous german fairy tale.

7. **map reduce methods** — embarrassingly parallel problems, i.e. many completely independent tasks. Those problems arise for example in Monte Carlo methods.

Any tool for HPC has to face the question how it can be applied to these seven problems. Petalisp aims to be a high level language that relieves the programmer from dealing with implementation details. As a consequence, Petalisp must be able to fully comprehend the behavior of a given program, conduct a performance analysis and optimize accordingly. Unfortunately such an automated analysis is not generally possible. All problems that require domain specific insight and human ingenuity can not be optimized by a machine. Such problems occur for example when dealing with the shape of an unstructured grid or when deriving an upper bound on the particle density in an n-body program. From the given "seven dwarfs", only dense linear algebra, structured grids and map reduce methods have enough regularity that a fully automated analysis and optimization seems feasible.

1.2 Common Lisp

Lisp is a family of programming languages and Common Lisp [1] is one of the two most prominent dialects of Lisp, the other one being Scheme [2]. Petalisp is written in Common Lisp. More precisely, Petalisp is a single Common Lisp package that extends the language with functionality to denote parallel algorithms, much like OpenMP [3] extends the C and Fortran programming languages. This way, Petalisp avoids reinventing the wheel and has access to every feature and library of Common Lisp.

There is not enough room in this thesis to give a satisfactory introduction to Common Lisp. Fortunately there are numerous excellent books that teach the language, e.g. [4, 5, 6, 7]. The advanced topics of Lisp programming are covered in [8, 9, 10, 11, 12, 13]. All of these books are most warmly recommended and teach valuable programming concepts. In the authors opinion, it is worth learning Lisp even for the sole purpose of reading the vast and enlightening literature related to that language.

At its core, Lisp is a very simple language. Its syntax consists purely of function or macro calls in prefix notation. Instead of $f(x)$, a Lisp programmer would write $(f\ x)$ and instead of $2 + 3$, a Lisp programmer would write $(+\ 2\ 3)$. Functions are introduced by the macro `defun` and the last expression in the body of a Lisp function is automatically returned. A side by side comparison of the syntax of a small C program and the roughly equivalent Lisp program is given in figure 1.

```c
int factorial(int n) {
    if(n == 0) return 1; else
    return n * factorial(n-1);
}
```

```lisp
(defun factorial (n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

Figure 1: Comparing the syntax of C and Lisp

Some readers may question the decision to write a modern HPC application in Common Lisp. According to typical programming language popularity rankings [14, 15, 16], there are dozens of programming languages that are more popular. This reasoning that popularity implies quality could be refuted by pointing at various political elections or popular pieces of music. Instead the next sections give actual examples why Common Lisp is not just a possible choice for new software projects, but also a very desirable one.

1.2.1 Integers and Rational Numbers

Computations involving integers are undoubtedly the backbone of most computer programs. The way numbers are treated has a deep impact on the behavior of the whole program. Yet many popular languages trade sanity for speed and replace operations on integers with operations on bounded integers, usually in a way that they can be represented using only 32 or 64 bits of memory. The two reasons given for this decision are the better performance and the fact that usually numbers are not bigger than 2^{32}. Both of these assumptions were reasonable in 1970, given the computers and compiler technology at that time, but should be reconsidered nowadays.

In general, operations on integers of arbitrary size are slower than those on fixed size integers. The former have to check for overflows and potentially allocate memory, while the latter can be executed in a single hardware operation. But what eliminates this performance penalty entirely is the use of type systems and type annotations. A modern compiler can infer that an array index is always a bounded integer, as arrays are allocated in the computer's memory which is itself bounded. Iteration variables in simple loops will also never exceed the bound in the loop test, and so on. Even in cases where it is not obvious to the compiler that an integer is bounded, a programmer can add an explicit annotation. Once a compiler dealing with arbitrary precision integers has such special knowledge, it emits the same code as a compiler for a language that supports only bounded integers. Modern programs that do not employ arbitrary size integers by default sacrifice correctness without cause.

The other argument for supporting only bounded integers is that numbers are usually not that big. This assumption is plain wrong and frequently disproved in prominent places, such as when in June 4, 1996 an Ariane 5 rocket left its trajectory and exploded because its horizontal velocity exceeded 16 bit [17]. Urging the programmer to ensure that integers do not overflow is not a viable option. In 2006 Joshua Bloch discovered an integer overflow bug that plagued the standard libraries of almost every programming language that relied on fixed size integers, even implementations that were proven correct and checked by hundreds of skilled programmers [18]. In 2012, a study of integer overflows in C and C++ programs found integer overflows that result in undefined behavior in 16% of the 1172 most popular Debian packages just during the invocation of their test suites, and more such errors even in popular compiler benchmarks [19]. The conclusion is that fixed size integers do not belong in the hand of humans — irrespective of their skill — and that a program that works with fixed size integers is most certainly broken! In order to illustrate this further, the reader is encouraged to write, in a language of his choice, a simple program to compute the factorial of a number and invoke it with numbers bigger than 20.

Common Lisp provides not only integers of arbitrary size, but also rational numbers and arithmetic functions that follow established mathematical practice. In Common Lisp, two divided by three will evaluate to two-thirds. As a consequence, many mathematical theorems can be readily transferred to Common Lisp code, such as the property that the equation $a\,x = b,\ a, b \in \mathbb{N},\ a \neq 0$ will have a solution given by $x = b/a,\ x \in \mathbb{Q}$. This intuitive treatment of numbers has facilitated the development of Petalisp considerably.

1.2.2 Metaprogramming

The Lisp family of languages is famous for its metaprogramming capabilities. This section explains why it is so easy to write Lisp programs that manipulate other Lisp programs. The fact that metaprogramming is so easy in Lisp can not be traced back to a single mechanism

or language feature. It is the interplay between farsighted language design[2] and decades of innovation, that make Lisp metaprograms so pleasant to write and to use.

Perhaps the most notorious difference between Lisp programs and essentially all other programming languages is the abundance of parentheses and the prefix notation. Contrary to widespread belief, this syntax was not introduced to vex programmers. It makes the language *homoiconic*, i.e. source code is at the same time a data structure. By using only prefix notation, Lisp source code corresponds closely to the syntax tree of the language. Figure 2 is an example of a data structure that that could be either treated as a tree of linked lists of symbols and numbers, or as a piece of code that yields true when evaluated in an environment where `a` and `b` are integers.

```
1    (= (* (+ a b)
2          (- a b))
3       (- (* a a)
4          (* b b)))
```

Figure 2: A mathematical identity, or three levels of nested linked lists

A homoiconic syntax alone is not sufficient to write complicated metaprograms. It is the evaluation model of Lisp that makes the language so interesting. According to computer science lore, there are compiled languages and scripting languages. Compiled languages are faster, because they are translated to optimized byte code or machine code. Scripting languages can be used interactively and sacrifice performance for productivity. Lisp fits in both categories. A Lisp program can load (and possibly compile) new code at any time and redefine its own functionality. In particular, a Lisp program can extend the way that code is read and evaluated. This allows to introduce special syntax or even domain specific languages. A typical Lisp program starts by defining some special syntax for the given problem domain and then solves the actual problem in this specially tailored dialect. Such special syntax may range from simple abbreviations to full blown code generators. If used correctly, such programs are much shorter and more maintainable that it would be possible in any general purpose programming language.

The canonical way to define special syntax in Lisp is via *macros*. A macro can be used as the first element of an expression, much like a function call. It differs from functions in that it is already invoked before the code is passed to the compiler or interpreter and that its arguments are not evaluated. The expression is then replaced by the result of the macro invocation. Recalling that Lisp code is essentially a syntax tree, a macro can be considered a function from syntax to syntax. Of course it is permissible that the input syntax of a macro is not valid Lisp syntax. Furthermore, the result of a macro invocation may again be an expression containing a macro. Consequently, macros are just functions from arbitrary data structures to arbitrary data structures, with the constraint that the final result ought to be valid Lisp code.

It should be clear at this point that there is no way to apply a static type system to a language with such flexibility. Traditional type systems prove that an expression has a certain type. In Lisp, one would have to prove that arbitrary layers of macros produce only expressions that evaluate to a certain type. This is not possible.[3] Nevertheless Common Lisp is still a type safe language. The type of all objects is known at runtime and an error is signaled whenever

[2]Examples of farsighted language design: Lisp introduced garbage collection in 1960 [20]. It is also the birthplace of conditional expressions, recursive functions and multi-character variable names.

[3]MetaML is a language that restricts the power of macros so that static type checking can be done [21].

a function is invoked on objects outside of its domain. Additionally, Common Lisp performs static type checking where applicable to improve performance and emit helpful warnings for the programmer. A dynamic type system has also its merits. It is possible to inspect data structures at any time, even while debugging or during macro expansion. A programmer can query the type of any object, the names of the currently defined functions and the list of predecessors of a class. The power of a program to reason about itself is a boost for programming productivity and a basis for many advanced metaprogramming techniques.

Syntax extensions occur in many programming languages. Loop statements are usually just syntactic sugar for code blocks separated by a conditional goto statement. All newer functionality of Java is implemented via syntax extensions, because it is not desirable to modify existing Java Virtual Machines. Even the C++ language can be considered a big monolithic syntax extension. Its original compiler, Cfront translated C++ source code to C code [22] and even today, C and C++ compilers are so similar that they share most of their code base. The features introduced by C++ are a typical example of what can be achieved with syntax macros. Unfortunately, C lacks the extensibility of Lisp and so C and C++ have diverged. Common Lisp is easily extensible. Features like the introduction of an object system or a special infix notation for mathematical expressions can be achieved with portable and encapsulated libraries. A consequence is that Common Lisp is under constant evolution and adapts quickly to new developments in computer science. Figure 3 shows how syntactic evolution leads to code that is both easier to understand and easier to write. For the curious reader, figure 4 gives a possible macro expansion of the loop macro in figure 3 (b), i.e. the code that is actually passed to the Lisp compiler after the loop macro is expanded.

```
1 int counter = 0;
2 for(int i = 3; i <= 50; i = i + 2) {
3     if(prime(i)) ++counter;
4 }
```

(a) A for-loop in the C programming language

```
1 (loop for i from 3 to 50 by 2
2       counting (prime i))
3
4
```

(b) The LOOP Macro of Common Lisp

Figure 3: Equivalent loop constructs in C and Common Lisp

```
1  (block nil
2    (let ((i 3))
3      (declare (type (and real number) i))
4      (let ((#:loop-sum-720 0))
5        (declare (type fixnum #:loop-sum-720))
6        (tagbody
7         next-loop
8          (when (> i '500) (go end-loop))
9          (when (prime i) (setq #:loop-sum-720 (1+ #:loop-sum-720)))
10         (setq i (+ i 2))
11         (go next-loop)
12        end-loop
13         (return-from nil #:loop-sum-720)))))
```

Figure 4: A possible macroexpansion for the code in figure 3(b)

Programming Lisp macros is a craft that has to be learned. There are rules and guidelines of how to write macros with unsurprising behavior and there are Lisp editors and programming environments that help debugging complicated macros. A fine book concerning macro programming is [10].

1.2.3 The Common Lisp Object System

I invented the term object oriented, and I can tell you
that C++ wasn't what I had in mind.
– Alan Kay (2003)

Object Oriented Programming (OOP) is a widely known programming paradigm. It was developed by Alan Kay by studying the organization of biological systems, which are incredibly complicated, yet resilient and adaptable. Kay wanted to design software systems that have the same sturdiness. He found out that the key ingredients of such a system are "messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things" [23]. Late-binding means that all functions and variables are (at least conceptually) looked up each time they are used and not earlier. This conflicts with the distinction between compile time and runtime that many programming languages employ, since it may be necessary to dynamically modify code whenever functions are called or new objects are defined. Drawing the analogy to biological systems, a human body requires insulin to live, so from a software perspective one could say that a human must have a method that returns insulin. At the time the human was "designed" (by nature, god or whatever), this method was provided by the pancreas. Unfortunately some humans develop diabetes, where the pancreas fails to provide enough insulin. The good news is that modern medicine has found other ways to provide insulin, for example via injections or insulin pumps. One could say that the method that provides insulin has been redefined at runtime. In other words late-binding literally saves lives.

The Common Lisp Object System (CLOS) is probably the most sophisticated OOP system at present. It evolved from several other Lisp object systems to satisfy the need for a truly portable and adaptive solution for all Lisp users. The design and a possible implementation are described in [13]. The most prominent high level features of CLOS are:

classes A class describes the structure and behavior of its instances. A class has a place in the directed acyclic graph of other classes and a set of slots that can hold values. Class definitions can also automatically define accessor functions and default constructors.

instances An instance of a certain class has at least all the slots and properties of that class. Instances are created using several generic functions to permit customization of every aspect from allocation to initialization.

generic functions A generic function is a function whose behavior depends on the classes of its arguments. For example, a generic function *paint* could be implemented with three arguments: The object to paint, the tool and the image that should be painted. Afterwards, different behavior can be defined for all possible classes of arguments, including error messages for nonsensical combinations and highly specialized routines for cases like drawing triangles on the screen of a computer.

dynamic redefinition It is possible at any point in time to add or redefine classes or generic functions. All existing methods and classes are immediately updated. Typically this

functionality is implemented by caching expensive computations like the class precedence list or specific generic functions. A redefinition will the simply invalidate the affected caches.

metaclasses Each class has a metaclass. The metaclass describes the behavior of the class in the same way that a class describes the behavior of its instances. Metaclasses allow different types of object orientation to coexist. For example it is possible to define a metaclass "java-class" whose classes have the semantics of the Java programming language. This may be useful to migrate existing code from other languages or to perform research on object oriented design.

introspection All meta information about classes, metaclasses and instances is available at runtime. It is possible to query the class of an object and the slots, superclasses and properties of a class.

method combinations Generic functions can do more than just describe the behavior for a certain set of argument classes. It is also possible to define behavior that is executed *before*, *after* and *around* a particular generic function by using method combinations. Advanced users can even specify their own method combinations, for example to state that the result of all suitable methods should be accumulated in some way.

The most exciting part about CLOS is that all these features can be implemented in an efficient manner, so that there is no point in choosing a less powerful OOP system just for the sake of performance. The next section sheds some more light on the issue of performance of Common Lisp programs.

1.2.4 Performance

The term *performance* is widely used in computer science, yet remarkably vague. A program is said to have better performance than another one, when it can solve the same problem in less time or with less resources. The crux lies in the term "same problem". It is near impossible to write two different programs that do exactly the same, especially when different languages and compilers are involved. A typical difference is the behavior when an error occurs, e.g. whether the program simply terminates, or whether it enters an interactive debugger and describes the error in a helpful way. Sometimes it is not even possible to reproduce similar performance results when using the very same source code on the same hardware (due to differences in the used libraries, operating system parameters, room temperature, etc). That being said, the author attempts to give a reasonable insight into the performance of Common Lisp.

In order to avoid the aforementioned problem that it is hard to solve exactly the problem in two different programming languages, it makes sense to review two separate performance aspects of Common Lisp. The first aspect is the performance of Common Lisp when mimicking the behavior of a C or Fortran program and solving a typical C or Fortran task. This is a metric that many HPC programmers are interested in and it allows a fair comparison to many existing benchmarks. The second aspect is the performance of Common Lisp when solving complicated software problems that would not be feasible without all its excellent tools and mechanisms. There is no way to compare the performance of such features as they are unique to the language. But it is nevertheless meaningful to determine their runtime cost because it allows a programmer to estimate whether they are appropriate for a certain task. All the performance results for the Common Lisp language were obtained using the benchmark suite `the-cost-of-nothing` [24], running the Common Lisp implementation SBCL, version 1.3.4. The C benchmarks have been conducted using the C compiler GCC 4.9.2. The benchmarks system for both cases has an i7-5500U CPU at 2.4GHz.

On to the low level benchmarks! Starting with some very low level comparisons, it turns out that function calls in C and Lisp have about the same cost of 3 nanoseconds ($\approx$ 10 CPU cycles). Memory allocation benchmarks, i.e. calling `malloc` in C or calling `cons` or `make-array` in Lisp, reveal that it takes 22 nanoseconds to allocate a 16byte data structure in C, but only 4 nanoseconds to allocate a Lisp data structure of the same size. For bigger allocations of 100kB the picture is reversed, with a cost of 1 microsecond in C and 20 microseconds in Lisp[4]. The allocation measurements reveal an important difference between the two languages. The garbage collected approach of Lisp seems to incur a high penalty for big allocations, but also permits that small objects are allocated extremely fast, essentially by doing an atomic pointer increment operation (since the garbage collector can move objects later to compact memory). Another difference between Lisp and C manifests when considering number crunching operations. The benchmark in question is to compute repeatedly $x = C \cdot (a + b)$, for two arrays a and b and a constant C, such that all data fits into the L1 cache of the CPU. It turns out that the C implementation is roughly four times as fast. Investigating the assembler code shows that the Lisp code does not utilize the AVX vector instructions of the target CPU.

After this short comparison of C and Lisp follow some pure Lisp benchmarks. Garbage collected languages like Lisp have a bad reputation in fields that require high performance, because programmers fear that their programs would be plagued by long hiccups where their program would stop to perform garbage collection. Naive garbage collectors do have this problem, but modern generational garbage collectors can be reasonably fast. On the benchmark system, a single run of the garbage collector takes about 3 milliseconds.

Common Lisp has the most sophisticated object system of all languages, where it is possible to redefine classes at runtime, introduce new types of object orientation via metaclasses and much more. All this functionality is provided by *generic functions* which can have different behavior depending on the types of their arguments and can contribute to or change the behavior of all more specific calls than the given signature. Benchmarks determine that it takes about 19 nanoseconds to allocate a generic class and 4 nanoseconds to access a class slot with a generic accessor function. A full call to a generic function has an (amortized) cost of about 7 nanoseconds. The explanation for this remarkable performance is that specific implementations of generic functions are extensively cached and reused. Figure 5 summarizes all aforementioned results.

operation	**Lisp**	**C**
function call	3ns	3ns
16B allocation	4ns	22ns
100kB allocation	20µs	1µs
number crunching	2.7 Gflops	8.6 Gflops
garbage collection	3ms	-
dynamic class creation	19ns	-
dynamic method dispatch	7ns	-
class slot access	4ns	-

Figure 5: Common Lisp benchmarks and comparisons

[4]It should be noted that it is also possible to call `malloc` from Lisp, in cases where this seems prohibitively expensive.

1.2.5 Maturity

The previous sections demonstrated that Common Lisp has several valuable strengths, ranging from mathematical treatment of numbers to metaprogramming, a truly object oriented design and high performance. But there are other programming languages who rival Common Lisp in some of these qualities, for example Julia [25] and Clojure [26]. Newer programming languages have always a certain appeal under the assumption that mankind gets smarter over time. Nevertheless there is a good reason why programmers should prefer Common Lisp: Maturity.

Maturity manifests itself in several ways. It means that the language has access to well designed and tested libraries and that there is an existing community of experienced programmers. It also means that there is a stable language standard and implementations for all typical hardware platforms and operating systems. And especially it means that code written today can still be used 20 (or 50?) years in the future. This is where Common Lisp truly excels. The ANSI Common Lisp standard was published in 1994 and was based on previous experience with the languages MacLisp, Zetalisp, Scheme and Interlisp. Since this point there was no need to change the language standard at all — every new concept in computer science could be implemented as a portable Common Lisp library. Extreme examples of such libraries are [27] for programming with multiple threads and [28] for efficient pattern matching. Another sign of maturity is that there are many different Common Lisp implementations, both commercial [29, 30] and non-commercial [31, 32, 33, 34, 35].

This section concludes the motivation for the Common Lisp language. Of course the language has many more features that are not covered here and also idiosyncrasies that can only be understood in historical context. After all, the ANSI Common Lisp standard is more than 1000 pages long. But now it is time to progress to the development of Petalisp.

2 The Design of Petalisp

For twenty years programming languages have been steadily progressing toward their present condition of obesity; as a result, the study and invention of programming languages has lost much of its excitement.
– John Backus (1977)

The ambition of Petalisp is to introduce some new ideas for parallel computing. Many existing programming languages have influenced this work: The concept of working on whole data structures instead of scalar values is taken from APL [36]. The SISAL programming language [37] demonstrates that destructive modifications of array elements are not necessary to to write fast code and do actually hamper a compilers ability to parallelize a program. Functional programming and the importance of parallel apply-to-all and reduction operations have been explored by Connection Machine Lisp.

2.1 Petalisp Core Operations

In theory, a modern HPC compiler matches human expert programmers when it comes to applying optimizations and parallelization techniques. In practice, seemingly unrelated changes in the source code or the target hardware can change the quality of the generated code dramatically. Writing abstract source code that is reliably compiled to fast machine code is more an art than a science. There are two principal reasons why compilers do not optimize reliably. The first one is that compilers typically perform ahead-of-time compilation, where the runtime behavior of the program is not known. The second problem is that it is impossible to consider all interactions of language constructs in a programming language whose standard is several hundred pages long.

Petalisp solves the problem of compile time uncertainty by just-in-time evaluation and compilation. In order to address the problem of language complexity, Petalisp uses only five core operations with well defined semantics: *application*, *reduction*, *fusion*, *reference* and *repetition*. Their definitions are given in the next sections. All user visible functionality is expressed by combinations of these core operations, which were chosen such that they explicitly express data parallelism and allow the notation of typical HPC algorithms.

Statements for conditional execution (e.g. if-statements) or control flow (e.g. loops or goto-statements) have been voluntarily omitted in Petalisp. Their absence can be compensated since Petalisp is embedded in the Common Lisp programming language and Common Lisp itself has such statements. However, omitting control flow statements in the core language improves the potential for optimization dramatically. The execution of Petalisp programs is completely predictable and permits reliable a priori performance analysis. This analysis is mandatory to guide later domain decomposition, scheduling and optimization techniques.

All operators have been formulated independently of the data structures they apply to. For the concerns of the core operations, the following generic definition is sufficient:

Definition 1 (data structure)**.** A Petalisp data structure a is a mathematical function from a set of keys Ω (e.g. tuples of integers) to a set of values.

A data structure has a *dimension* of d if its domain can be represented as the cartesian product of d sets, that is $\Omega = \Omega_1 \times \ldots \times \Omega_d$.

There are numerous data structures that fulfill this definition. Arrays can be considered a mapping from a set of integers to a set of values, a dictionary might map from a set of strings to another set of strings. Since the definitions of the core operations are decoupled from the data structures they operate on, the language is easily extensible. Even different versions of Petalisp that are optimized for particular data structures are imaginable. Considerations in later sections show that it may be desirable to support only a single data structure, at least for the extent of this thesis.

2.1.1 Distributed Application

The distributed application of functions is the primary source of parallelism in any Petalisp program. It is essentially equivalent to the *map* construct of many functional languages. It applies an n-ary function element-wise to n given data structures. All applications in Petalisp are distributed to the data structures they operate on, so for the sake of brevity, the distributed application of a function will simply be called application from now on.

Definition 2 (application)**.** Let f be an arbitrary Common Lisp function that accepts n arguments, and let $a_1, \ldots, a_n$ be Petalisp data structures from a set of keys Ω to some values. Then the application of f to $a_1, \ldots, a_n$ is a Petalisp data structure that maps each element k of Ω to $f(a_1(k), \ldots, a_n(k))$.

As an example, the application of the Lisp function `log` to a data structure containing floating point numbers yields a data structure containing the natural logarithm of those numbers. The application of the function `+` to two data structures of integers yields a data structure, where each value is the sum of the corresponding value in the first data structure and the corresponding value in the second data structure. There are no guarantees when or in which order the given function is applied to the individual arguments, so the application of a function with side-effects may result in surprising behavior.

2.1.2 Reduction

Another source of parallelism is introduced by reduction operations. Informally speaking, a reduction combines the elements of a data structure pairwise, until a single value remains. The reduction operation is also called *fold* in some functional languages.

Definition 3 (reduction)**.** Let f be an arbitrary Common Lisp function that accepts two arguments, and let a be Petalisp data structure of dimension n, i.e. a mapping from each element of the cartesian product of the sets $\Omega_1, \ldots, \Omega_n$ to some values. Then the reduction of a by f is a Petalisp data structure of dimension $n-1$ that maps each element k of $\Omega_1 \times \ldots \times \Omega_{n-1}$ to the value of some arbitrary expression tree (see figure 6) with f as nodes and the values of a at $k \times \Omega_n$ as leaves.

The definition, particularly the notion of an expression tree becomes clear when examined for a particular example. Figure 6 shows several different ways how the one-dimensional vector of integers $1, 2, 3, 4$ can be reduced by a function f. The result of the reduction depends on the specified Common Lisp function f. Reducing with `+` yields the sum of the given numbers, reducing with `max` yields the biggest number.

The current definition of a reduction in Petalisp permits the reduction in any order. This decision was made to simplify the parallelization of Petalisp programs. A consequence is that

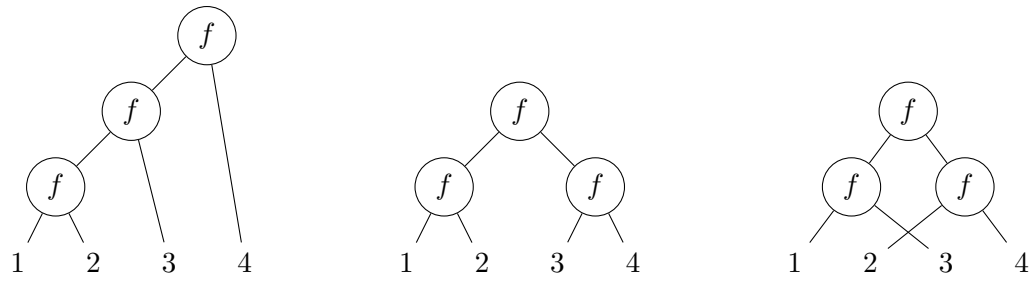


Figure 6: Different expression trees for a particular reduction

reduction operations are only deterministic for functions that are associative and commutative. The author is aware that some numerical applications require a more fine-grained control of the order of reductions. Later versions of Petalisp may address this issue.

2.1.3 Fusion

The fusion of two or more data structures merges all key-value pairs of the given input data structures. Depending on the data structures it operates on, it can be used for a wide range of effects from computing the set union to stacking arrays.

Definition 4 (fusion)**.** Let $a_1, \ldots, a_n$ be n Petalisp data structures, where each data structure a_k maps from a set of keys Ω_k to a set of values. Furthermore, let the sets $\Omega_1, \ldots, \Omega_n$ be pairwise disjoint. Then the fusion of $a_1, \ldots, a_n$ is a Petalisp data structure that maps each element x of $\bigcup_{k=1}^{n} \Omega_k$ to the value of x of the unique data structure a_k whose domain contains x.

There are several ways how a fusion can be erroneous. One error is when a reduction is applied to a data structure of dimension zero. Another error occurs when a key is found in more than one input data structure. In both cases there is no sensible way to resolve the problem, so an error is signaled. There is one more erroneous situation, that is not immediately obvious from the above definition: When there exists no data structure whose domain is the union of the domains of the input data structures. E.g. in a system whose only data structure are contiguous arrays with arbitrary start and end index, the fusion of arrays with the domains $\{1, 2\}$, $\{3, 4\}$ and $\{5, 6\}$ is an array with the domain $\{1, 2, 3, 4, 5, 6\}$, while the fusion of arrays with the domains $\{1, 2\}$ and $\{4, 5\}$ is erroneous, because the result can not be represented as a contiguous array.

2.1.4 Reference

While the application operates on the set of values of a data structure, the reference operation operates on the set of keys. It can transform and shrink a given data structure. The name was chosen because it is essentially a vectorized and extended variant of the array reference known from C-style programming languages.

A possible definition of reference would be to map an arbitrary function over the set of keys, while retaining the old values. The drawback of this approach is that it is impossible to perform automated analysis on arbitrary functions. As a consequence, Petalisp allows only reference operations with a small set of functions. A function that is permissible for reference operations is henceforth called *transformation*. The set of transformations is further discussed in section 2.4.

Definition 5 (reference)**.** Let a and x be Petalisp data structures and t be an invertible Petalisp transformation. Let Ω_x be the domain of x. Then the reference of a by x and t is a Petalisp data structure that maps each k from $\{k \mid \exists i \in \Omega_x : k = t(i)\}$ to $a(t^{-1}(k))$.

Trivial references arise in the case where t is the identity mapping. In this case the reference operation simplifies to a selection of all those values of a that have indices that are also valid for x. References with identity transformations and a suitable space x can be used to select e.g. the first five elements of a vector or the lower right corner of a matrix. More elaborate references are obtained by transformations that are not the identity mapping. A permuting transformation can be used to transpose whole matrices and vectors, and a translating transformation can be used to prepare the index spaces of several arrays for a subsequent fusion operation, e.g. to stack several arrays next to each other.

2.1.5 Repetition

The repetition of a data structures a to another data structure b is a data structure with the same keys as b that is filled with values from a. It can be used to generate huge data structures with repetitive content, like an array whose values are all zero.

Definition 6 (repetition)**.** Let a and b be Petalisp data structures with the respective domains $\Omega_a = \Omega_{a,1} \times \ldots \times \Omega_{a,d_a}$ and $\Omega_b = \Omega_{b,1} \times \ldots \times \Omega_{b,d_b}$, where $d_a \leq d_b$ and where each set $\Omega_{a,d}$, $1 \leq d \leq d_a$ is either $\Omega_{b,d}$ or contains only a single element $\omega_{a,d}$. Let the projection $p_d(k)$ be defined by

$$p_d(k) = \begin{cases} \omega_{a,d} & \text{if} \quad \Omega_{a,d} = \{\omega_{a,d}\} \\ k & \text{otherwise} \end{cases}$$

Then the repetition of a to b is a data structure that maps each $k = (k_1, \ldots, k_{d_b}) \in \Omega_b$ to $a(p_1(k_1), \ldots, p_{d_a}(k_{d_a}))$.

The trivial use case is the repetition of a scalar constant (i.e. with dimension zero) to fill a d-dimensional array. But the definition permits also more elaborate cases, i.e. to fill a $m \times n$ matrix with a $m \times 1$ column vector.

Of all core operations, the repetition is the least orthogonal one, since it can be emulated with the fusion and reduction operation. The rationale for including this operation is that in order to achieve good parallel scaling, the complexity of elementary operations must be independent of the size of the data structures they operate on. This would be violated e.g. by creating a $n \times n$ array of zeroes by n^2 invocations of the fusion operation (and even by a smart, hierarchical approach that requires only $2 \log_2(n)$ operations).

2.2 Strided Arrays

It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.
– Alan Perlis (1982)

A fundamental design decision of Petalisp are the kinds of data structures that are supported. Conceptually Petalisp is a collection oriented language, so one is tempted to allow a

wide range of collections like sets, vectors and hash tables. However, each new data structure increases the complexity of Petalisp considerably and would likely impair later optimization. To illustrate this, consider just the functions that convert from one data structure to another. Given Petalisp would support n different data structures, there would already be $n(n-1)$ conversion functions. Each of these conversion functions would have to be written to make use of parallel hardware, which is already a daunting task. Even more troublesome would be the necessary amount of logic in the later optimization stages in order to exploit all special properties of all interactions of all data structures. With these considerations in mind, Petalisp will, for now, only support a single data structure. It is assumed that a Petalisp programmer will value reliable optimization more than a plethora of features.

After the decision to keep the number of data structures at a minimum of one, there is still the intricate task of choosing this data structure. Analysis of other programming languages for HPC suggests that multidimensional arrays are most useful, as they can represent everything from matrices and tensors to structured grids. After careful consideration, Petalisp uses a slightly generalized variant of multidimensional arrays, where the valid set of indices in each dimension is denoted by three integers: A lower bound, a step size and an upper bound. Such a triple of integers will be called *range* (Definition 7), the data structure itself will be called *strided array* (Definition 8).

Definition 7 (range)**.** A range is a set of integers defined by

$$\text{range}(x_L, s, x_U) := \{\, x \in \mathbb{N} \mid x_L \leq x \leq x_U \ \wedge\ (\exists k \in \mathbb{N})\,[x = x_L + ks] \,\} \quad x_L, s, x_U \in \mathbb{Z}.$$

The integers x_L, s and x_U are called the *start*, *step* and *end* of the range, respectively.

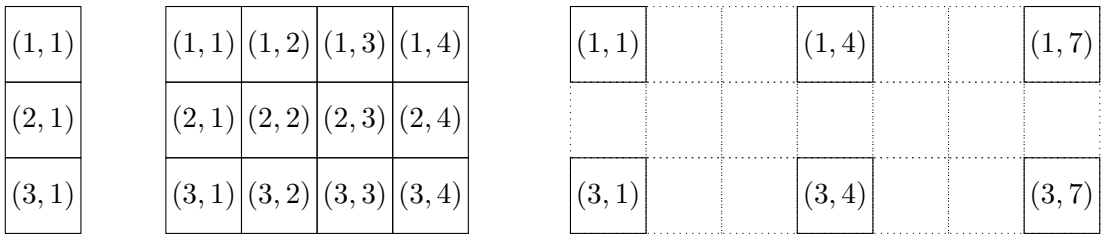
Ranges have been inspired by the array slicing operator, which is featured in almost every programming language that specializes on numerical computation. A range can be used to represent the iteration space of a single for-loop in C style languages or to describe a slice of data in an existing array. The fact that a range is essentially a linear mapping with box constraints makes it pleasant to manipulate them.

Definition 8 (strided array)**.** A strided array in n dimensions is a function from elements of the cartesian product of n ranges to a set of values.

Strided arrays have a number of advantages over traditional arrays. They completely avoid the debate of whether array indices should start from one or from zero and make it very convenient to stack or split existing arrays. Furthermore, a structured grid can be decomposed into multiple interleaving strided arrays to implement coloring algorithms like the Red-Black Gauss-Seidel method. Some examples of strided arrays are given if figure 7.

2.3 Strided Array Index Spaces

Petalisp encourages maximal data parallelism. Instead of statements to access a single value of a strided array, there are operations to select and possibly transform whole subspaces of a strided array. Denoting index spaces and computing intersections, differences and unions of index spaces are therefore crucial operations. Technically the index space of a strided array is nothing but its domain, which is a set of tuples of integers. So in order to operate on the index space of a strided array, Petalisp must have some way to represent these sets. However, introducing a new data structure would break with the decision to support only strided arrays.



(a) 3×1 array **(b)** 3×4 array **(c)** strided array with step sizes of 2 and 3

Figure 7: Several two-dimensional strided arrays

The solution is that index spaces are themselves represented as strided arrays, that map from a set of indices to those indices (Definition 9), e.g. the domain of a 2×2 array is the set $\{(1,1),\ (1,2),\ (2,1),\ (2,2)\}$, but the index space in the sense of Petalisp is another 2×2 array, where each key equals its value.

Definition 9 (strided array index space)**.** A strided array index space in n dimension is the identity function whose domain is restricted to the cartesian product of n ranges.

One can think of strided array index spaces as data structures that carry no information but their domain. They are so frequently used that they deserve their own notation. Throughout this thesis and also in the Petalisp source code, strided array index spaces are denoted by σ, followed by triples of integers for the start, step and end parameter of the range of this particular dimension, as shown in figure 8.

```
(σ)                          ;; the zero-dimensional space
(σ (0 1 8) (0 1 8))          ;; the index space of a 9 × 9 array
(σ (10 2 98))                ;; all even two-digit numbers
(σ (1 2 3) (1 2 3) (1 2 3)) ;; the corners of a 3 × 3 × 3 cube
```

Figure 8: Strided array index space examples

2.4 Possible Transformations in Petalisp

A final ingredient for this HPC language is still missing. Strided arrays have been defined and it is possible to apply functions to them, reduce them, select parts of them and fuse them. But there is no possibility to modify the index space of an existing strided array. These modifications are introduced by specifying the permitted transformations in the reference operator from definition 5. It is important to keep the space of permitted transformations small enough to allow optimization, yet big enough to solve many reasonable HPC problems. Study of real world applications has determined, that the following five elementary transformations are particularly useful:

1. **translate** Change all indices by a constant, e.g. change an array starting with index zero to an array with starting index one.

2. **scale** Multiply all indices with a constant, e.g. change a strided array with domain $(\sigma\ (0\ 3\ 9))$ to an array with domain $(\sigma\ (0\ 1\ 3))$, by multiplying it with $\frac{1}{3}$

3. **permute** Modify the order of indices, e.g. transpose a matrix by changing the first and the second index.

4. **shrink** Drop one or more indices. This is only permitted when the array has only one element in the corresponding dimension, e.g. it is allowed to change a $3 \times 1 \times 5$ array to a 3×5 array, but not to a 3×1 array. This restriction makes the transformation unambiguous.

5. **enlarge** Add one or more indices, e.g. change a 2×9 array to a $1 \times 2 \times 9$ array or change the strided array (σ (4 2 8)) to the strided array (σ (7 1 7) (4 2 8)).

These transformations have the beautiful property that they are isomorphisms, i.e. they have an inverse and that the composition of any number of them is essentially an affine mapping (with some restrictions on the domain, due to the shrink operation). To celebrate this mathematical clarity, these transformations get their own special notation that will be used throughout this thesis and the Petalisp source code. A transformation is introduced with the letter τ, followed by a list of inputs and several outputs. Conceptually, a transformation is applied by matching each element of the domain against the list of inputs, binding variables accordingly and then evaluating the outputs with those variables bound. Several examples of valid Petalisp transformations are given in figure 9.

```
(τ ())                ;; the transformation from (σ) to (σ)
(τ (i) (+ i 2))       ;; shift all indices of a 1D array by 2
(τ (m n) m (* n 1/2)) ;; scale the second dimension by 1/2
(τ (m n) n m)         ;; invert a matrix
(τ (5 a) a)           ;; drop the first index if it is 5
(τ (a) a 9)           ;; introduce a second dimension with index 9

;; an example featuring all elementary transformations together
(τ (foo 3 m)
   (/ (+ (* 90 (+ 2 m)) 15) 2)
   (- foo (+ 1 1))
   (* 2 12))

;; the same transformation as before, but simplified
(τ (a 3 c)
   (+ (* 45 c) 195/2)
   (- a 2)
   24)
```

Figure 9: Transformation examples

2.5 Pushing Amdahl's Law to the Limit

Amdahl's Law [38] is a classical model to estimate the speedup S of running a task composed of a serial fraction s and a parallel fraction $p = 1 - s$ on N processors.

$$S(N) = \frac{1}{s + \frac{1-s}{N}} \qquad \text{(Amdahl's Law)} \tag{1}$$

A consequence of Amdahl's Law is that a program with a fixed serial fraction will not achieve infinite speedup by increasing the number of processors. This leads to the derived metric of the parallel efficiency ε_p, the percentage of processors that is efficiently used during the execution of a task. The parallel efficiency is interesting for all operators and users of compute centers, as it determines whether it is worth running a program on a parallel computer or not.

$$\varepsilon_p(N) = \frac{S(N)}{N} = \frac{1}{s(N-1)+1} \qquad \text{(parallel efficiency)} \tag{2}$$

If one demands a parallel efficiency of at least 50% from modern HPC programs, a consequence of the above formula is that the serial fraction of the program must be less than $\frac{1}{N-1}$ of the total execution time. At a first glance this seems dire, because in order to scale to more than ten processors, at most 10% of the work may be serial. But since in reality, the serial fraction of work can get very close to zero, HPC applications are rarely limited by Amdahl's Law. In many numerical simulations, the size of the problem and the potential for parallelism grows with the number of processors.

Petalisp turns the tables with respect to Amdahl's Law. It is assumed that Petalisp programs consist of a small, sequential, high-level program that invokes completely parallel subroutines to do the bulk of the work. The question is how much processor time can be spent in the high-level part of Petalisp whilst still attaining reasonable parallel efficiency. In order to live up to its name, Petalisp should run reasonable on systems with about 10^5 processor cores. At $\varepsilon_p \geq 0.5$ this permits a serial fraction of 10^{-5}. Given a typical processor clocks at around $2.5 \cdot 10^9$Hz, the serial fraction of Petalisp has about $2.5 \cdot 10^4$ processor cycles per second to execute. Petalisp uses these cycles to delay domain partitioning, compilation and scheduling decisions to the runtime. The predicted $2.5 \cdot 10^4$ processor cycles per second are obviously not sufficient to generate and compile full scientific programs. Scaling to true petascale systems requires that most of the code generation overlaps with actual computation and that already compiled operations are cached and reused where possible.

3 Implementation

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.
– Edsger W. Dijkstra (1984)

Many classical software projects tend to grow rapidly in size. Of course it makes sense that a project grows proportional to its functionality, but often the growth is also caused by redundancy, lack of abstractions and the general attitude to never rewrite code that "already works". In the author's opinion, programs should be more viewed as mathematical proofs, where brevity and elegance are the highest virtues and where new, simplifying insights should always be incorporated. Petalisp has been written (and constantly rewritten) in this spirit. It has currently less than 2000 lines of easily comprehensible source code that is publicly available as free software [39].

Since Petalisp is actively being discussed and improved, the information in this thesis is likely to become outdated[5]. As a consequence, there is little use in giving a line by line explanation of the whole source code. Instead, this chapter strives to give a high-level overview of the Petalisp project and an explanation of some crucial algorithms.

3.1 High-Level Overview

The Petalisp implementation consists of the following high-level components:

data type definitions The definitions of transformations, data structures, index spaces and ranges have been abstract. The implementation maps these abstract objects to Common Lisp classes and structures and provides sensible operations to manipulate them. As an example, ranges are implemented as Lisp structs of three integers (start, step and end), together with a normalizing constructor.

index space operations All Petalisp operations work on whole spaces of values. To reason about these operations, the implementation provides methods to compute the intersection, difference and union of the corresponding index spaces.

transformations Transformations are a subset of the invertible functions on multidimensional indices. The implementation can analyze and convert arbitrary functions to transformation objects and has methods to compose and invert such transformations and to transform given index spaces.

type inference Petalisp is a dynamically typed language. However, type inference can determine the type of an expression in many cases such that specialized functions and memory representations can be used. For example it is known that the sum of two double-precision floating-point numbers is again a double-precision floating-point number and that the product of a double-precision floating-point number and a single-precision complex number is a double-precision complex number.

code generation All Petalisp operations operate on spaces of arbitrary dimensions and types, and there is a vast space of possible transformations on given data. To provide fast implementations for all these cases, Petalisp generates and compiles code when

[5]For reference, the git revision number of the Petalisp implementation at the time of writing is `0f1d50a46e0d62160ea08eaef65946c9fd10ee4d`.

needed. E.g. permuting a ten-dimensional array of integers triggers the generation and compilation of an expression with ten nested loops. All compiled expressions are cached and reused, so that all subsequent calls to a given operator have an overhead of a single hash table lookup.

graph optimization Programs written with Petalisp may result in arbitrarily convoluted data flow graphs. Often these graphs can be simplified to require less nodes. For example, a sequence of references can be replaced by a single reference, where the individual transformations are composed. The Petalisp implementation applies many possible optimizations already in the individual node constructors, so that the actual construction of a new node can be avoided in many cases.

graphviz output Petalisp data flow problems can be converted to the Graphviz dot file format and visualized – a invaluable tool for debugging and testing.

parsing of the σ and τ notation The σ notation for strided array index spaces and the τ notation for transformations are not standard Common Lisp syntax. Both notations are implemented as Lisp macros to fit seamlessly into the language.

an API The bulk of the Petalisp implementation operates on specialized, orthogonal functions that are not adequate for everyday usage. Petalisp provides an API with convenient "do what I mean" operations that expand into possibly many invocations of the five fundamental operators. It is recommended that all Petalisp programs are written using the convenient API functions.

test suite The Petalisp implementation strives to achieve production quality. The individual parts were written to be "obviously correct" where possible. Nevertheless, there is also an extensive test suite that performs many checks, ranging from tests on individual functions to full real-world applications.

error handling Apart from correctly executing correct programs, the implementation also detects and reports all erroneous situations. In fact, many algorithms in Petalisp are only present for the purpose of error checking. For example emitting the data flow graph node for a fusion operation is simple, but checking that some given inputs can be fused successfully is a tough problem.

The aforementioned components are sufficient to run every imaginable Petalisp program, but the potential for optimization and parallelization is not yet exploited. In the current state, all programs are executed somewhat naively and without utilizing parallel hardware. Nevertheless, Petalisp has reached an important milestone, because it is possible to develop and run programs in it. Real world algorithms can be implemented in Petalisp and generate important feedback for the design decisions. It is likely that practical experience may lead to further refinements and improvements in the implementation. The next sections provide a more in-depth description of selected algorithms and techniques in the Petalisp implementation.

3.2 Intersection of Strided Array Index Spaces

As clarified during the design chapter of Petalisp, especially by definition 8, strided arrays are a more general variant of classical arrays. Each strided array has an index space, i.e. a description of the set of permissible indices to this data structure. This section describes the algorithm that is used to determine the intersection of two strided array index spaces. A single

index space can be viewed as a the intersection of two sets of vectors, where $\mathbf{x}_L$ are the lower bounds, $\mathbf{x}_U$ are the upper bounds and $\mathbf{s}$ are the step sizes of the ranges of this index space:

$$\{\mathbf{x} \in \mathbb{Z}^d \mid \mathbf{x}_L \leq \mathbf{x} \leq \mathbf{x}_U\} \tag{3}$$

$$\{\mathbf{x} \in \mathbb{Z}^d \mid \exists \mathbf{k} \in \mathbb{Z}^d,\ \mathbf{x} = \mathbf{x}_L + \mathbf{k}\mathbf{s}\} \tag{4}$$

Equation 3 is a simple box constraint. The intersection of two such boxes is either empty or another box constraint. The tricky part is the intersection of two spaces as defined by equation 4. The constraints in each dimension are independent, so it is sufficient to find an algorithm that computes the intersection of two one-dimensional spaces and apply it to each dimension. The one-dimensional intersection problem corresponding to equation 4 is

$$\{x \in \mathbb{Z} \mid \exists k_1, k_2 \in \mathbb{Z},\ x_{L,1} + k_1 s_1 = x = x_{L,2} + k_2 s_2\}. \tag{5}$$

A way to compute this set is by finding a particular element x and adding all multiples of the least common multiple of s_1 and s_2 to them. This works because the definition of the least common multiple ensures that it is always possible to find suitable integers k_1 and k_2. So it remains to find a particular solution to the problem in equation 5. If such a particular solution exists, there must be k_1 and k_2 such that

$$k_1 s_1 - k_2 s_2 = x_{L,2} - x_{L,1}. \tag{6}$$

All ranges in Petalisp are normalized such that s_1 and s_2 are always positive. So the crucial part in computing the index space intersection is to find two integers x and y such that

$$ax + by = c \qquad a,\, b \in \mathbb{N},\ c \in \mathbb{Z}. \tag{7}$$

Under the assumption that c is divisible by the greatest common denominator of a and b (which is a necessary condition for solubility anyways), equation 7 can be rewritten as

$$a \underbrace{\frac{s\,c}{\gcd(a,b)}}_{x} + b \underbrace{\frac{t\,c}{\gcd(a,b)}}_{y} = c \tag{8}$$

and finally by multiplying both sides by $\gcd(a,b)/c$, one obtains

$$as + bt = \gcd(a,b). \tag{9}$$

The extended euclidean algorithm is an efficient way to compute s, t and $\gcd(a,b)$ in equation 9. It is treated extensively in [40] under the name Algorithm X, where it is presented in the following way: Given non-negative integers u and v, determine the vector (u_1, u_2, u_3) such that $uu_1 + vu_2 = u_3 = \gcd(u,v)$.

X1. [Initialize.] Set $(u_1, u_2, u_3) \leftarrow (1, 0, u)$, $(v_1, v_2, v_3) \leftarrow (0, 1, v)$

X2. [Is $v_3 = 0$?] If $v_3 = 0$, the algorithm terminates.

X3. [Divide, subtract] Set $q \leftarrow \lfloor u_3/v_3 \rfloor$, and then set

$$(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - (v_1, v_2, v_3)q$$
$$(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3),\ (v_1, v_2, v_3) \leftarrow (t_1, t_2, t_3)$$

Return to step **X2**.

It is also hinted in [40] that the implementation can be slightly simplified by omitting u_2, v_2 and t_2, since u_2 can be regained via the relation $uu_1 + vu_2 = u_3 = \gcd(u, v)$. Now it remains to turn this algorithm into efficient Lisp code. Algorithm X has been presented in an iterative way, but can be viewed as the fixed point of a recursive function. Figure 10 is an implementation of Algorithm X that uses the `labels` construct to define a local recursive function and invokes it with the right initial values. The `values` statement in line 5 allows this function to return multiple values, in this case u_1 and $\gcd(u, v)$. Since the recursive call in line 7 is the last action in this function, any reasonable compiler will avoid a full function call and emit the same, if not better code than any iterative implementation.

```
1 (defun extended-euclid (u v)
2   (labels
3       ((euclid (u1 u3 v1 v3)
4          (if (zerop v3)
5              (values u1 u3)
6              (let ((q (floor u3 v3)))
7                (euclid v1 v3 (- u1 (* q v1)) (- u3 (* q v3)))))))
8     (euclid 1 u 0 v)))
```

Figure 10: An implementation of the extended Euclidean Algorithm

The code as presented in figure 10 computes the desired result for all positive integers u and v, including numbers that are far bigger than one machine word. This is desirable because it is not possible to impose a sane upper bound on the inputs. On the other hand it means that all operations in this implementation need to be prepared to deal with integers of arbitrary size, which occurs some performance penalty. This seems wasteful, because typically Petalisp will deal with step sizes that are small. Furthermore, the intersection function is invoked very frequently, so the speed of this algorithm is relevant. Fortunately there is a way to make the extended euclidean algorithm both fast and correct by noting that the variables u_1, u_3, v_1 and v_3 are bounded by the least common multiple of u and v. Using the product of u and v as a cheap upper bound for $\mathrm{lcm}(u, v)$ allows to choose between two versions of the algorithm: One for small integers that fit into machine words ("fixnums" in Lisp parlance) and one for integers of arbitrary size, as depicted in Figure 11. The `declare` statements in lines 2 and 12 are used to pass special knowledge to the compiler, such as the type of the variables. The optimize statement in line 14 declares that the following lines of code should be compiled to particularly fast code and should not expect any runtime errors.

At a first glance, the code in figure 11 seems redundant. Actually it outlines an important technique of how functions can be made both fast and correct. There can never be a (reasonably fast) compiler with the insights necessary to derive an upper bound on the variables in the recursive function, particularly for the variables $u1$ and $v1$, since they are monotonically increasing in magnitude. Manually splitting the domain into parts that can be computed efficiently and using a slower, generic default version otherwise gives both speed and correctness. It is disconcerting that programming languages like C and Fortran support only the fast versions of algorithms on integers and resort to undefined behavior otherwise.

```
(defun extended-euclid (u v)
  (declare (type unsigned-byte u v))
  (labels
      ((bignum-euclid (u1 u3 v1 v3)
         (if (zerop v3)
             (values u1 u3)
             (let ((q (floor u3 v3)))
               (bignum-euclid
                v1 v3
                (- u1 (* q v1)) (- u3 (* q v3))))))
       (fixnum-euclid (u1 u3 v1 v3)
         (declare (type fixnum u1 v1)
                  (type (and unsigned-byte fixnum) u3 v3)
                  (optimize (speed 3) (safety 0)))
         (if (zerop v3)
             (values u1 u3)
             (let ((q (floor u3 v3)))
               (fixnum-euclid
                v1 v3
                (- u1 (the fixnum (* q v1)))
                (- u3 (the fixnum (* q v3))))))))
    (if (<= (* u v) most-positive-fixnum)
        (fixnum-euclid 1 u 0 v)
        (bignum-euclid 1 u 0 v))))
```

Figure 11: A fast implementation of the extended Euclidean Algorithm

Finally no optimization is complete without a before/after comparison, so some benchmarks were conducted for the code in figures 10 and 11. The benchmark system has an Intel i7-5500U CPU running at 2.4GHz and uses the Common Lisp implementation SBCL, version 1.3.4. The input arguments were chosen such that the algorithm makes eight recursive calls. The simple implementation in figure 10 takes about 240 nanoseconds per invocation, the faster implementation in figure 11 takes only 136 nanoseconds. This is satisfying because 136 nanoseconds is roughly the time it takes for the benchmark system to execute eight consecutive integer divisions.

Equipped with an algorithm to find a particular solution x to the problem in equation 5, it is possible to compute the smallest solution bigger than the start of the current range x_L and biggest solution less than the end x_U by

$$\text{smallest} = x + \operatorname{lcm}(s_1, s_2) \lceil (x_L - x) / \operatorname{lcm}(s_1, s_2) \rceil \tag{10}$$

$$\text{biggest} = x + \operatorname{lcm}(s_1, s_2) \lfloor (x_U - x) / \operatorname{lcm}(s_1, s_2) \rfloor, \tag{11}$$

where $\operatorname{lcm}(s_1, s_2)$ can be cheaply computed from the $\gcd(s_1, s_2)$ that is returned from the extended Euclidean Algorithm via

$$\operatorname{lcm}(a, b) = \frac{a\,b}{\gcd(a, b)}. \tag{12}$$

Putting all these components together results in a fast method to compute the intersection of two ranges as depicted in figure 12. The intersection of two index spaces is then simply the result of applying this method on each dimension separately.

```
(defmethod intersection ((range-1 range) (range-2 range))
  (let ((lb (max (range-start range-1) (range-start range-2)))
        (ub (min (range-end   range-1) (range-end   range-2)))
        (a (range-step range-1))
        (b (range-step range-2))
        (c (- (range-start range-2) (range-start range-1))))
    (multiple-value-bind (s gcd)
        (extended-euclid a b)
      (when (integerp (/ c gcd))
        (let ((x (+ (* s (/ c gcd) a)
                    (range-start range-1)))
              (lcm (/ (* a b) gcd)))
          (let ((smallest (+ x (* lcm (ceiling (- lb x) lcm))))
                (biggest  (+ x (* lcm (floor (- ub x) lcm)))))
            (when (<= lb smallest biggest ub)
              (range smallest lcm biggest)))))))))
```

Figure 12: Computing the intersection of two ranges

3.3 Handling Transformations

Transformations have been introduced in section 2.4 with the letter `τ`. Petalisp users will want to specify their own transformations, so intuitive and robust input and readable output of transformations are vital. Several methods must also be implemented to incorporate any transformation neatly into the rest of the Petalisp system. Petalisp requires that there are two functions `compose` and `invert` that compute the functional composition of two transformations and the inverse of a transformation, respectively. The bridge between transformations and data structures is formed by the function `transform`, which reshapes a given index space with a given transformation. The Common Lisp object system allows dispatch based on types. This opens the prospect of classifying certain special cases of transformations and dispatch to faster and more specialized variants of `transform`, `invert` and `compose`. At the moment this technique is only used for identity transformations. Later versions of Petalisp may also provide specialized versions for pure translations, or transformations that do not permute their arguments.

3.3.1 Parsing

The input of transformations means converting source code expressions with the special `tau` notation from section 2.4, like `(τ (m n) ((+ m 1) (- n 1)))`, to an object of type transformation. The presented technique uses a Lisp macro to offload the task of parsing and evaluating the expression to the Common Lisp implementation. The code uses the fact that in Lisp, unnamed functions of any number of variables can be introduced with `lambda` and that functions can return any number of values using the `values` statement. The symbol `tau` is now implemented as a Lisp macro that converts its body to a regular function call to `classify-transformation`, as shown in figure 13.

```
;; two-dimensional translation
(τ (m n) ((+ m 1) (- n 1)))
```

```
(classify-transformation
 (lambda (m n)
   (declare (ignorable m n))
   (values (+ m 1) (- n 1)))
 #(nil nil) 2)
```

```
;; example with input constraints
(τ (1 2 i) (4 i 6))
```

```
(classify-transformation
 (lambda (#:g708 #:g709 i)
   (declare
     (ignorable #:g708 #:g709 i))
   (values 4 i 6))
 #(1 2 nil) 3)
```

Figure 13: Expansions of the τ Macro

The three arguments to `classify-transformation` are the function that shall be classified, a vector of input constraints and the number of output arguments. The `ignorable` declarations are purely cosmetic and avoid potential compiler warnings of unused variables. The macro that performs this conversion is given in figure 14. In line 13 it demonstrates the use of the backquote notation to specify a data structure where some parts are constant and other parts (those prefixed with a comma) need to be evaluated.

```
(defmacro τ (input-forms output-forms)
  (loop for form in input-forms
        collect (when (integerp form) form) into input-constraints
        collect (if (symbolp form) form (gensym)) into symbols
        finally
           (return
             `(classify-transformation
               (lambda ,symbols
                 (declare (ignorable ,@symbols))
                 (values ,@output-forms))
               ,(apply #'vector input-constraints)
               ,(length output-forms)))))
```

Figure 14: Implementation of the τ Macro

It remains to classify a given transformation. It is not generally possible to classify a function with only a fixed number of evaluations. Fortunately Petalisp limits itself to affine linear functions and permutations. Determining the affine coefficients of a function with n arguments requires only two function evaluations: one to determine the translation and another one to determine the slope. A third evaluation allows Petalisp also to detect most cases of nonlinear transformations and signal an appropriate error message. Afterwards, it takes another n evaluations to reliably determine the permutation of the input arguments. Cases where an input variable affects more than one output are detected in this step and reported as errors. The current implementation has only 56 lines of source code and is not interesting enough for inclusion in this thesis. The value returned from `classify-transformation` is

an object of type transformation that consists internally of three arrays: The vector of input constraints, whose length is the number of inputs, a two-dimensional array of affine coefficients and a permutation vector that describes which input corresponds to which output.

3.3.2 Output

Printing Common Lisp objects aesthetically is as simple as providing a new specialization for the generic function `print-object` for the particular type, in this case transformations. The permissible transformations contain affine-linear mappings, permutations, input constraints and newly introduced dimensions with only a single index. This gives rise to a number of special cases that need to be addressed during printing. A list of such special cases appears in figure 15.

```
1  (τ (a b) b a)        ; a permutation
2  (τ (5 a b) a b)      ; from three to two dimensions
3  (τ (a) 0 a)          ; from one to two dimensions
4  (τ (a) (+ a 2))      ; translation only
5  (τ (a) (- a 2))      ; translation with a negative sign
6  (τ (a) (* a 2))      ; scaling only
7  (τ (a) (+ (* a 9) 2) ; translation and scaling
8
9  ;; more variables than letters in the alphabet
10 (τ (V1 V2 ... V39 V40) V40 V39 ... V2 V1)
```

Figure 15: Different special cases when printing transformations

The current implementation of `print-object` for objects of type transformation is shown in figure 16. It starts by generating a list of variables, one for each input argument. These variables are either the first letters of the alphabet[6], or consecutively numbered and starting with the letter V. The function `intern` in line 6 is used to convert a given string to a Lisp symbol. The variables and coefficients of the transformation are used to construct the list of input expressions and the list of output expressions, while handling all the special cases shown in figure 15. Finally the `format` function in line 27 is used to print the generated expressions to the given output stream. The second argument to format is a string that describes the desired output in a special syntax. The `~a` directive renders the next given object in an aesthetically pleasing way. The `~{` and `~}` directives instruct `format` to process the next given list element-wise.

3.3.3 Identity Transformations

Occasionally, a user defined transformation, or the composition of two transformations will result in an identity transformation. This case is frequent, especially since the Petalisp reference operator uses an identity transformation unless otherwise specified. In order to reliably detect whether an affine transformation is an identity transformations one can exploit that every Lisp instance is initialized by the generic function `initialize-instance`. It is possible to amend its definition so that after an affine transformation is initialized, it is tested for being an identity transformation and if so, its class is changed to this more specific type. Afterwards,

[6]The dot after the letter t in line 2 of figure 16 seems odd at first. It is necessary to avoid a name clash with frequently used the Lisp symbol for true, which is t.

```
(defmethod print-object ((object affine-transformation) stream)
  (let* ((abc '(a b c d e f g h i j k l m n o p q r s t. u v w x y z))
         (variables
           (if (<= (input-dimension object) (length abc))
               (subseq abc 0 (input-dimension object))
               (loop for inpos below (input-dimension object)
                     collect (intern (format nil "V~d" inpos)))))
         (input-forms
           (loop for input-constraint across (input-constraints object)
                 and variable in variables
                 collect (or input-constraint variable)))
         (output-forms
           (loop for outpos below (output-dimension object)
                 with coefficients = (affine-coefficients object)
                 collect
                 (let* ((p (aref (permutation object) outpos))
                        (a (aref coefficients outpos 0))
                        (b (aref coefficients outpos 1))
                        (var (and p (nth p variables)))
                        (mul-form (cond ((zerop a) 0)
                                        ((= a 1) var)
                                        (t `(* ,a ,var)))))
                   (cond ((zerop b) mul-form)
                         ((eql mul-form 0) b)
                         ((plusp b) `(+ ,mul-form ,b))
                         ((minusp b) `(- ,mul-form ,(abs b)))))))))
    (format stream "(τ ~a~ ~a~)" input-forms output-forms)))
```

Figure 16: The output method for transformations

the definition of the functions `compose`, `invert` and `transform` can be specialized for the case of identity transformations as shown in figure 17.

This optimization for a common case shows that generic function dispatch can actually improve both the modularity of a system and the performance. The overhead of choosing the appropriate function at runtime is more than compensated by such custom-tailored code. The generic functions in figure 17 do nothing more than return their first or second arguments and avoid not only a good deal of computation, but also the allocation of memory for a new object of type transformation.

3.4 Code Generation

The ease of code generation was one of the principal motivations for choosing Common Lisp as an implementation language for this project. In fact, it can be as simple as invoking the function `compile` on a suitable expression. Expressions are represented as linked lists, therefore all tools that manipulate linked lists can also be used on Lisp source code. For illustration, all steps are applied to a model problem of providing a function that increments a given integer x by another integer N. The classical implementation of this function would be

```
(defmethod invert ((tr identity-transformation)) tr)

(defmethod compose ((g transformation) (f identity-transformation)) g)

(defmethod compose ((g identity-transformation) (f transformation)) f)

(defmethod transform ((object data-structure)
                      (f identity-transformation))
  object)
```

Figure 17: Efficient operations on identity transformations

```
(lambda (x) (+ x N))
```

This approach works fine as long as N is known at the time the code is written. When this is not the case, it seems that the function must receive `N` as a second argument. Lisp offers a more elegant way to avoid this second function argument, by utilizing closures. Closures are a by-product of true lexical scoping of variables and allows a function to preserve local state. The unary increment function can be obtained at runtime by writing

```
(defun adder (N)
  (lambda (x) (+ x N)))
```

and calling the function with a suitable increment. Closures can be compiled to remarkably fast code, so the given model problem can be considered solved. But for the purpose of illustrating code generation in Common Lisp, it is assumed that general addition of integers is an expensive operation and that it is advantageous to have only constant increments. In this case, such a special purpose code could be achieved by building the suitable expression from symbols and linked lists and compiling it at runtime:

```
(defun adder (N)
  (compile nil
    (list 'lambda (list 'x) (list '+ 'x N))))
```

The Lisp compiler will receive the given N as a constant and may emit specialized hardware instructions, e.g. when N is one. The only objection to this solution could now be that it is cumbersome to explicitly construct lists of symbols. It would be more pleasant to declare that the lambda expression is mostly constant and only the increment variable should be substituted. Lisp offers a special syntax for this frequent case: Mostly constant expressions are introduced with the backquote symbol `, and the parts that should be evaluated in the surrounding environment are marked with a comma. Exploiting this notation simplifies the code generator further:

```
(defun adder (N)
  (compile nil
    `(lambda (x) (+ x ,N))))
```

The actual code generator in Petalisp works very similar to the above examples. The only extension is that it defines special syntax to create named code generators. These code generators cache all compiled results, so that repeated invocation with the same arguments costs exactly one hash table lookup. Furthermore, Petalisp detects when a code generator is redefined at runtime and purges all affected functions from the compilation cache. A named code generator is introduced with `defkernel`. Applied to the model problem, it results in the following two-line solution:

```
(defkernel adder (N)
  `(lambda (x) (+ x ,N)))
```

Automatic code generation has made it possible that Petalisp supports arbitrary dimensional operators, e.g. a single, eight-dimensional transformation of index spaces results in some code with eight specialized nested loops. The array element types are hard coded, which permits Petalisp to utilize special representations like bit-vectors where possible. Furthermore, since the high-level part of Petalisp asserts that all operations are well formed, there can be no array out of bounds accesses and the Lisp compiler can be instructed to omit special bounds checking operations. As a result, nothing prevents the generated Lisp code from being slower than any C or Fortran code[7].

3.5 The User Interface

The core operations of Petalisp have been chosen to maximize orthogonality and to avoid complicated special cases. Applications require that all their arguments have the same shape and references can only be declared with both a target space and a transformation. Using these operations directly in a program is overly verbose. Consequently, Petalisp defines a user interface of convenient functions and macros that abbreviate commonly used patterns. These wrappers have the same expressiveness as the core operations, but allow for a more terse notation. At the time of writing, the whole user interface consists of six operations that are covered hereafter.

While the core operations are an integral part of the language, the beauty and design of the user interface are a moving target. New useful abstractions may crop up and existing ones may turn out impractical and may get removed. To accommodate for such changes, the user interface of Petalisp is loaded and defined just after the rest of the language, so that there can be no dependencies from any core component to it. This way, the whole appearance of the language can be customized within minutes.

3.5.1 α and β

The native constructors for application and reduction have the signature

[7] Admittedly, C and Fortran compilers still have an advantage when it comes to numerical codes. However, this is not the result of language differences, but of a different focus of the compiler developers of both communities.

(`application OPERATOR OBJECT ...`)

and

(`reduction OPERATOR OBJECT`).

The user interface of Petalisp provides more general alternative constructors, which are internally converted to one or more primitive constructors. Applications are abbreviated by α and reductions are abbreviated by β. These names have been inspired by the similar operators in CM-Lisp [41]. Both functions have the same signature as their primitive counterparts, but accept not only Petalisp data structures, but also Lisp arrays and scalars, which are suitably converted. Furthermore, the α operator performs broadcasting on its arguments, i.e. when possible, all arguments are extended with the repetition operation to a suitable common space. For example applying any operator to a 3×1 array and a 1×9 array has the effect that each input is extended to a 3×9 array and that the operator is applied to the extended inputs. The single character functions α and β admit already some nice examples, as seen in figure 18. The `#'` in front of the first arguments in figure 18 instructs Common Lisp to access the function with of the given name, instead of the variable. This is necessary because Common Lisp has separate namespaces for functions and variables.

```
1  ;; the sum of a vector of numbers
2  (β #'+ #(1 2 3 4 5 6 7 8 9))
3
4  ;; adding two numbers
5  (α #'+ 2 2)
6
7  ;; the dot product of vectors u and v
8  (β #'+ (α #'* u v))
9
10 ;; the row sum norm of matrix A
11 (β #'max (β #'+ (α #'abs A)))
```

Figure 18: Examples of the `α` and `β` operator

3.5.2 The → Operator

The signature of the reference operator is

(`reference SPACE TRANSFORMATION`),

which is rarely appropriate in user code. Often one desires to write only a transformation or only a reference. Sometimes it is easiest to use an alternating sequence of transformations, e.g. to transform a strided array to a contiguous array, select a subset of it and transform it back. The Petalisp user interface introduces the arrow operator to apply any sequence of modifications to a given data structure. Its syntax is

(→ OBJECT MODIFIER ...).

The modifications are processed from left to right and handled depending on their type. The result of the first modification is used as the argument to the second one and so on. The result of the last modification is returned. The valid types of modifications are

larger index-space When an index space, e.g. `(σ (0 9) (1 7))` is encountered that has a higher dimension than the current object, or when the index space of the current object is a subspace of this index-space, emit a repetition from the current space to this space.

smaller index-space When an index space is encountered that is smaller than the index space of the current object, emit a reference to this particular subset. For example `(→ A (σ (0 0)))` selects the element of `A` with the index zero.

transformation When a transformation is encountered, emit a reference that applies this transformation to the current object.

Any number of such modifiers can be placed in a sequence. Given the above description, a user might fear that a single call to → might result in many nested primitive references. This fear is unsubstantiated, because the primitive constructors of Petalisp will simplify any number of subsequent references and fusions automatically. Some examples of using the arrow operator are given in figure 19.

```
;; a 10x10 array of zeroes
(→ 0 (σ (0 9) (0 9)))

;; a 2x2 array of all ones
(→ #(1 1) (σ (0 1) (0 1)))

;; the first three elements of a vector
(→ #(1 2 3 4 5 6 7 8 9) (σ (0 2)))

;; the last three elements of a vector
(→ #(1 2 3 4 5 6 7 8 9) (τ (x) (- x 6)) (σ (0 2)))

;; again, the last three elements of a vector
(→ #(1 2 3 4 5 6 7 8 9) (σ (6 8)) (τ (x) (- x 6)))

;; the first three elements of a vector, in reverse order
(→ #(1 2 3 4 5 6 7 8 9) (τ (x) (- 2 x)) (σ (0 2)))
```

Figure 19: Examples of the → operator

3.5.3 `fuse` and `fuse*`

The fusion is an operator that takes any number of arguments and merges them to a single data structure. Its signature is

(`fusion OBJECT ...`).

The user interface provides the operator `fuse` that does nothing but convert all arguments to Petalisp objects and apply the fusion operation to the converted arguments. An error is signaled when the arguments do partially overlap, because this situation would otherwise be ambiguous. Another way to resolve this ambiguity is the `fuse*` function. It is similar, but whenever some argument indices coincide, it takes the corresponding value of the rightmost argument. Among other things, this permits to define arrays with selective modifications. Some illustrative examples are given in figure 20.

```
;; an array of six zeroes
(fuse (→ 0 (σ (3 5))) (→ 0 (σ (6 8))))

;; an array of three zeroes and three ones
(fuse (→ 0 (σ (3 5))) (→ 1 (σ (6 8))))

;; an array of two zeroes, two eights and two ones
(fuse* (→ 0 (σ (3 5))) (→ 1 (σ (6 8))) (→ 8 (σ (5 6))))
```

Figure 20: Examples of the `fuse` and `fuse*` operator

3.5.4 The σ* Macro

Occasionally, a programmer will want to specify an index space relative to an existing one. For translations, scaling and permutations, this effect can be achieved with transformations. But a case that is not handled is e.g. to select the interior points of a certain strided array. To facilitate this, Petalisp provides a macro that creates a context in which the variables start, step and end are bound to the respective values of a reference space. The usage of this macro is shown in in figure 21.

```
;; select the interior of GRID
(σ* GRID
  ((+ start step) step (- end step))
  ((+ start step) step (- end step)))
```

Figure 21: An example of using the σ* macro

It is not possible to obtain the behavior of this utility by ordinary function calls, because the start, step and end variables would be unbound. Instead the expression must be syntactically modified to introduce these variables around each expression. Such a scenario is the typical use case for Lisp macros. This particular macro works by placing each expression after the input object in a context where two hidden variables for the space and the respective dimension are bound and by defining start, step and end to be locally equivalent to accessing the corresponding value. The macro definition of σ* is given in figure 22. When applied to the example in figure 21, it will expand roughly to the code in figure 23. In case the reader has not had previous experience with Lisp, this macro definition may not be obvious. What

should be noted regardless, is that a completely new syntax for data structures of arbitrary dimensions has been introduced with only ten lines of portable code. The particular details of how to access the parameters of an index space are completely hidden by the macro. Furthermore, since macros are expanded before code is compiled or evaluated, such an abstraction introduces no runtime cost.

```
(defmacro σ* (space &rest dimensions)
  (with-gensyms (dim)
    (once-only (space)
      `(symbol-macrolet
           ((,(intern "START") (range-start (aref (ranges ,space) ,dim)))
            (,(intern "STEP") (range-step (aref (ranges ,space) ,dim)))
            (,(intern "END") (range-end (aref (ranges ,space) ,dim))))
         (make-index-space
          ,@(loop for form in dimensions and d from 0
                  collect `(let ((,dim ,d)) (range ,@form))))))))
```

Figure 22: Imlementation of the σ* macro

```
(let ((#:space917 GRID))
  (make-index-space
   (let ((#:dim916 0))
     (range
      (+ (range-start (aref (ranges #:space917) #:dim916))
         (range-step (aref (ranges #:space917) #:dim916)))
      (range-step (aref (ranges #:space917) #:dim916))
      (- (range-end (aref (ranges #:space917) #:dim916))
         (range-step (aref (ranges #:space917) #:dim916)))))
   (let ((#:dim916 1))
     (range
      (+ (range-start (aref (ranges #:space917) #:dim916))
         (range-step (aref (ranges #:space917) #:dim916)))
      (range-step (aref (ranges #:space917) #:dim916))
      (- (range-end (aref (ranges #:space917) #:dim916))
         (range-step (aref (ranges #:space917) #:dim916)))))))
```

Figure 23: The macroexpansion of the σ* macro in figure 21

4 Petalisp Examples

As a by-product of this thesis, there is now a fully functional Petalisp implementation. It is publicly available at [39]. Readers are encouraged to experiment with it and feedback is most welcome. The software repository contains a growing number of example programs. Three selected examples are hereinafter discussed.

4.1 Matrix Multiplication

Matrices are rectangular arrays of numbers. The matrix multiplication is a function that maps an $m \times n$ matrix A and an $n \times k$ matrix B of the shapes

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1k} \\ b_{21} & k_{22} & \cdots & k_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nk} \end{pmatrix} \tag{13}$$

to a $m \times k$ matrix C, whose elements are defined by

$$c_{i,j} = \sum_{p=1}^{n} A_{ip} B_{pj}. \tag{14}$$

From the Petalisp perspective, the sum in equation 14 is a reduction with the function $+$. The elements of the matrix C are all reductions, so the whole $m \times k$ matrix C can be viewed as the reduction of a three-dimensional data structure of dimension $m \times k \times n$, whose elements are the product of numbers from A and B. The desired $m \times k \times n$ data structure can be obtained by stacking k instances of A and m instances of B, transforming both with a suitable permutation, and combining them with the multiplication function. Combining all these steps results in the matrix multiplication program in figure 24.

```
1 (defun matrix-multiplication (A B)
2   (β #'+
3      (α #'*
4         (→ A (τ (m n) (m 1 n)))
5         (→ B (τ (n k) (1 k n))))))
```

Figure 24: A Petalisp program to compute the matrix multiplication

It is instructive to examine the individual steps it takes to execute the program in figure 24. The standard evaluation rules for Common Lisp apply, so the expressions are evaluated inside-out and from left to right. When the matrix multiplication routine is invoked on two matrices, the implementation proceeds as follows:

1. The `τ` expressions in lines 4 and 5 are evaluated. After macro expansion, each transformation is converted to an unnamed Lisp function, which is, together with the number of arguments and input constraints, passed to the function `classify-transformation`. During classification, the unnamed function is called multiple times to determine whether it is an affine linear mapping. If so, the coefficients of this mapping are identified and used to construct a transformation object.

2. The → functions in lines 4 and 5 receive their arguments. If their first arguments are Lisp arrays, they are converted to Petalisp strided arrays, otherwise they are left unchanged. Afterwards, suitable references to these strided arrays are created depending on the remaining arguments. In line 4, this yields a reference node from a $m \times 1 \times n$ space to the given $m \times n$ array A. In line 5, this yields a similar reference node to a $1 \times k \times n$ space.

3. The α function in line 3 receives three arguments: The Lisp function for multiplication and two three-dimensional array references. It determines that its arguments are spaces of different size, but can be broadcast to a single space of size $m \times k \times n$. Consequently it wraps both references by repetition nodes to this common space. The repetition nodes have now the same shape and can be used to construct an application node with the multiplication operator. The application node is then returned

4. The β function in line 2 introduces a reduction. It receives the Lisp function for addition and the previously computed application node. By definition, the reduction applies to the last dimension, so the β function returns a reduction node that maps from the given $m \times k \times n$ array to a $m \times k$ array.

5. The result of the last form is automatically returned from `matrix-multiplication`. It is a data flow graph, whose root is a reduction node, followed by an application node, followed by two repetition nodes, that each wrap reference nodes to constant input arrays. The graph for the multiplication of a 3×2 matrix and a 2×5 matrix is shown in figure 25.

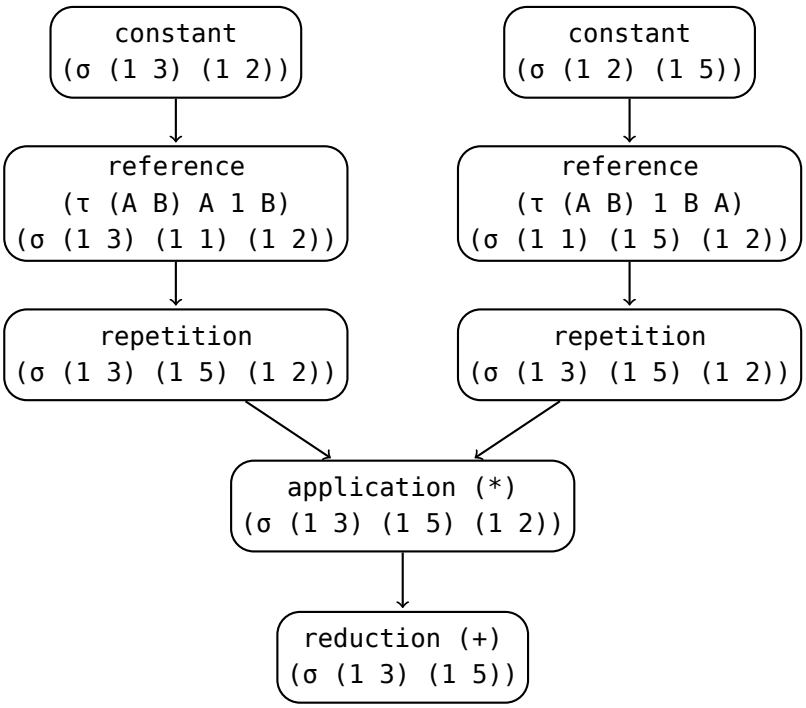


Figure 25: The data flow graph of a matrix multiplication

It is important to realize that the matrix multiplication method in figure 24 does not actually perform any computation of the target matrix C. Its return value is a data flow graph that tells Petalisp how to compute C, and that this computation is well formed, i.e. all

operations have matching data types and index spaces. To actually compute C, the resulting data flow graph must be passed to the Petalisp function `compute`. The interplay between graph creation and evaluation can be viewed as an extreme form of lazy-evaluation, as practiced by some functional programming languages.

4.2 The Jacobi Method

The Jacobi method is an algorithm to solve diagonally dominant systems of linear equations. Such systems arise frequently in the discretization of partial differential equations (PDEs). A simple and instructive example of such a PDE is the Laplace equation with Dirichlet boundary conditions

$$-\Delta u = 0 \qquad \text{on } \Omega \tag{15}$$

$$u = C \qquad \text{on } \partial\Omega. \tag{16}$$

Equation 15 states that the divergence of the negative gradient of a given quantity u shall be zero at each point in the domain Ω. Equation 16 provides the Dirichlet boundary conditions for the problem, by assigning u fixed values on each point of the domain boundary $\partial\Omega$. In practice, the Laplace equation occurs e.g. when solving heat diffusion problems. In this case, u would be the temperature of the medium. Heat flows from hot regions to colder ones, so the heat flux can be modeled by the negative gradient of u. The conservation of energy implies that the divergence of the heat flux is zero and leads exactly to the Laplace equation.

For most domains and boundary values, there is no way to derive an analytic solution to the Laplace equation. Instead the solution must be approximated by numerical methods. A classical approach is to replace equation 15 by a linear system of equations that approximate the Laplace operator at a finite set of grid points in the interior of Ω. If a two-dimensional domain is discretized by a rectangular grid with distance h, a linear approximation of the Laplace equation at each point is given by

$$-\Delta u(x,y) \approx \frac{-4u(x,y) + u(x+h,y) + u(x-h,y) + u(x,y+h) + u(x,y-h)}{h^2} = 0. \tag{17}$$

The validity of this formula can be confirmed by Taylor expansion and solving for the second derivatives. The Jacobi method approximates the solution of such a system by starting with a random initial guess u_0. Based on this initial guess, an improved guess u_{k+1} is obtained by solving the linear system point-wise, using the values of u_k for each neighbor:

$$u_{k+1}(x,y) = \frac{u_k(x+h,y) + u_k(x-h,y) + u_k(x,y+h) + u_k(x,y-h)}{4} \tag{18}$$

This update formula is applied in parallel to the interior point of the discretized domain Ω. The Dirichlet boundary values are never modified. The Petalisp implementation of the Jacobi method for this particular problem is shown in figure 26. It starts in line 2 by defining the interior of the given domain. The interior can then be used in the lines 10–13 to select exactly those points of u that are in the interior *after* the specified transformation. The `fuse*` function then describes the next value of u_{k+1} by overriding the interior of u_k according to the update formula from equation 18.

```
(defun jacobi-2d (u iterations)
  (let ((interior
         (σ* u ((+ start 1) 1 (- end 1))
               ((+ start 1) 1 (- end 1)))))
    (loop repeat iterations do
      (setf u (fuse*
               u
               (α #'* 0.25
                  (α #'+
                     (→ u (τ (x y) (1+ x) y) interior)
                     (→ u (τ (x y) (1- x) y) interior)
                     (→ u (τ (x y) x (1+ y)) interior)
                     (→ u (τ (x y) x (1- y)) interior))))))
    u))
```

Figure 26: The Jacobi Method to solve the equation $-\Delta u = 0$

4.3 The Multigrid Method

The multigrid method is one of the most sophisticated solvers for differential equations. It is widely applicable and extremely efficient, often achieving optimal asymptotic complexity. An excellent introduction to this method is given in [42]. The key idea is that classical iterative methods like the Jacobi or Gauss-Seidel method have a poor convergence for large grids. This is not surprising. Looking at an implementation of the Jacobi method in figure 26, reveals that each grid point depends only on the values of its neighbors. As a result, it takes at least n Jacobi iterations until a point on the one side of the grid "notices" a change on the boundary of the other side, and much longer until a reasonable solution is obtained. The multigrid method improves on these classical methods by working with a hierarchy of smaller and smaller grids, applying a Jacobi or Gauss-Seidel solver on each level, and transporting the improved solution between the grids. A multigrid algorithm consists of four independent parts that are examined in the next sections. The example problem is again the Laplace equation with Dirichlet boundary conditions, as specified in equation 15 and 16.

4.3.1 Computing the Residual

The residual of a linear system $Ax = b$ is defined by

$$r = b - Ax. \tag{19}$$

If x is the exact solution of the linear system, the residual is zero. For a randomly chosen $\tilde{x}$, it allows to quantify the error $e = \tilde{x} - x$. The multigrid algorithm uses the residual to correct an existing solution, by first approximating the error e in

$$Ae = r = b - A\tilde{x} \tag{20}$$

and then correcting $\tilde{x}$ via

$$\tilde{x}_{\text{new}} = \tilde{x} - e \tag{21}$$

Using the same discretization for the Laplace equation as in equation 17 leads to a point-wise residual of

$$r(x,y) = b - \frac{-4u(x,y) + u(x+h,y) + u(x-h,y) + u(x,y+h) + u(x,y-h)}{h^2}. \tag{22}$$

The grid spacing h in equation 22 is usually passed as a parameter to the method. For simplicity, it is assumed that the grid is equidistant and has a height and width of one, so that the spacing can be derived from the number of grid points N as

$$h = \frac{1}{1 - \sqrt{N}} \tag{23}$$

The Petalisp code for the computation of the residual is shown in figure 27. It starts by defining the interior of the given grid and computing the grid spacing h. In line 4, the `fuse*` function declares that the residual is zero by default (here for the boundaries of the grid) and the value of the computation in lines 6–13 otherwise, which is just the the Petalisp equivalent of equation 22.

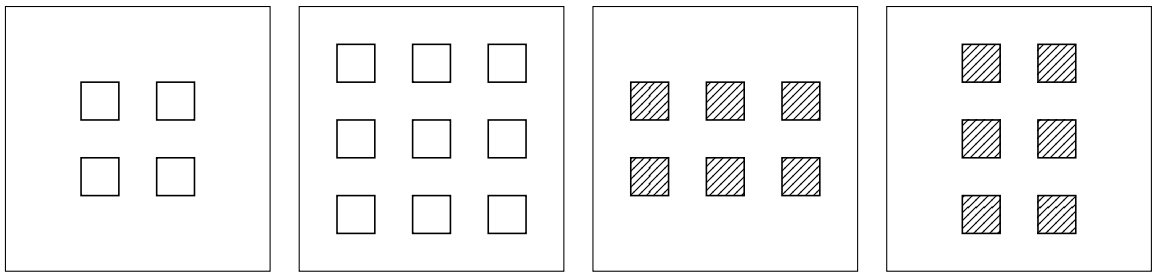
```
1  (defun residual (u b)
2    (let ((interior (σ* u ((1+ start) 1 (1- end))
3                          ((1+ start) 1 (1- end))))
4          (h (/ (1- (sqrt (size u))))))
5      (fuse* (→ 0.0 (index-space u))
6             (α #'- (→ b interior)
7                (α #'* (/ (* h h))
8                   (α #'+
9                      (→ (α #'* -4.0 u) interior)
10                     (→ u (τ (i j) (1+ i) j) interior)
11                     (→ u (τ (i j) (1- i) j) interior)
12                     (→ u (τ (i j) i (1+ j)) interior)
13                     (→ u (τ (i j) i (1- j)) interior)))))))
```

Figure 27: Computing the residual $r = b - Ax$ for the discretized Laplace equation

4.3.2 The Red-Black Gauss Seidel Method

The Jacobi method has two drawbacks. The first one is that it may have bad convergence for some initial values. All nodes are updated simultaneously, which means that the improvement at one point may get completely nullified by the changes at the neighboring points. The second disadvantage of the Jacobi method arises during the practical implementation. Every grid point in iteration $k + 1$ depends on points in the previous iteration k, so a computer needs to store two instances of the whole grid. The Red-Black Gauss Seidel method fixes both deficiencies. It partitions the grid into alternating red and black nodes, just like a chessboard. Then it uses the update scheme from equation 18 to update all red points in parallel and afterwards all black points.

The implementation is shown in figure 29. It is the first program that uses an array stride other than one. There is no direct way to represent all nodes of the same color as a Petalisp



(a) first set of red points **(b)** second set of red points **(c)** first set of black points **(d)** second set of black points

Figure 28: Domain partitioning for the Red-Black Gauss Seidel method

```
1  (defun rbgs (u b iterations)
2    (let ((r1 (σ* u ((+ start 2) 2 (1- end)) ((+ start 2) 2 (1- end))))
3          (r2 (σ* u ((+ start 1) 2 (1- end)) ((+ start 1) 2 (1- end))))
4          (b1 (σ* u ((+ start 2) 2 (1- end)) ((+ start 1) 2 (1- end))))
5          (b2 (σ* u ((+ start 1) 2 (1- end)) ((+ start 2) 2 (1- end))))
6          (h (/ (1- (sqrt (size u))))))
7      (labels ((update (u what)
8                 (α #'* 0.25
9                    (α #'+
10                      (→ u (τ (i j) (1+ i) j) what)
11                      (→ u (τ (i j) (1- i) j) what)
12                      (→ u (τ (i j) i (1+ j)) what)
13                      (→ u (τ (i j) i (1- j)) what)
14                      (→ (α #'* (* h h) b) what)))))
15       (loop repeat iterations do
16         (setf u (fuse* u (update u r1) (update u r2)))
17         (setf u (fuse* u (update u b1) (update u b2))))
18       u)))
```

Figure 29: The Red-Black Gauss Seidel method

data structure, but it is possible to split each color into two strided arrays as shown in 28. The points on the boundary are not colored, because they are known to be exact.

The code in figure 29 is greatly abbreviated by introducing the utility function `update` in line 7 to apply the numerical stencil to a given set of points. It is sufficient to specify the desired target points (here with the variable `what`) and the coordinates of the inputs are automatically inferred from the transformations in lines 10–13. This is a great improvement compared to implementations in classical programming languages, where both input and output indices must be specified manually. In all other respects, the code is similar to the implementation of the Jacobi method in figure 26.

4.3.3 The Prolongation Operator

The multigrid algorithms works on grids of different size. More precisely it uses grids with a side length of $2^l + 1$, where l is the current level of computation. In other words the grid grows exponentially with l, up to the size of the original grid that discretizes the domain Ω.

Linear interpolation allows to move the data of a grid with level l to a finer grid with level $l+1$. The program in figure 30 distinguishes four types of nodes to handle the interpolation efficiently. The first set `u*` are points that take exactly the values of corresponding nodes in the smaller grid. It is defined in line two by multiplying the coordinates of each input point by two. The second and third set of nodes are those points who lie directly between two points from `u*`. Their coordinates are defined in line 3 and 5. The last set are those points whose four diagonal neighbors are from `u*`. The result of the prolongation is simply the fusion of the values of all four spaces. The values of `u*` are already known. The other three strided arrays can be computed as the weighted average of their neighbors, which happens in line 10, 14 and 18.

```
1  (defun prolongate (u)
2    (let* ((u* (→ u (τ (i j) (* i 2) (* j 2))))
3           (space-1 (σ* u* ((1+ start) step (- (1+ end) step))
4                           (start step end)))
5           (space-2 (σ* u* (start step end)
6                           ((1+ start) step (- (1+ end) step))))
7           (space-3 (σ* u* ((1+ start) step (- (1+ end) step))
8                           ((1+ start) step (- (1+ end) step)))))
9      (fuse u*
10           (α #'* 0.5
11              (α #'+
12                 (→ u* (τ (i j) (1+ i) j) space-1)
13                 (→ u* (τ (i j) (1- i) j) space-1)))
14           (α #'* 0.5
15              (α #'+
16                 (→ u* (τ (i j) i (1+ j)) space-2)
17                 (→ u* (τ (i j) i (1- j)) space-2)))
18           (α #'* 0.25
19              (α #'+
20                 (→ u* (τ (i j) (1+ i) (1+ j)) space-3)
21                 (→ u* (τ (i j) (1+ i) (1- j)) space-3)
22                 (→ u* (τ (i j) (1- i) (1+ j)) space-3)
23                 (→ u* (τ (i j) (1- i) (1- j)) space-3))))))
```

Figure 30: A bilinear prolongation operator

The effect of the prolongation operator can be seen easily when studying a simple example:

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} \xrightarrow{\text{prolongation}} \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.25 & 0.5 & 0.25 & 0.0 \\ 0.0 & 0.5 & 1.0 & 0.5 & 0.0 \\ 0.0 & 0.25 & 0.5 & 0.25 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

In this example, a 3×3 grid with a single nonzero value in the middle is prolongated to a 5×5 grid.

4.3.4 The Restriction Operator

The restriction is the inverse operation of the prolongation. A grid with side length $(l+1)^2+1$ is mapped to a grid with side length l^2+1. A naive approach would be to drop all but the grid nodes with even indices. This should make the programmer uncomfortable, because three quarters of the original information are irrevocably lost. Indeed, numerical analysis or experiments show that this naive restriction operator cripples the whole multigrid algorithm. Instead it is better to define each target node as the weighted average of all surrounding nodes. The weights of each node are chosen to complement the bilinear prolongation operator from the previous section. Figure 31 shows the implementation of this operator. All points with even coordinates, as well as the set of interior nodes are defined in line 2 and 3. The interior even points are defined as the weighted sum of their neighbors, while the boundary points are taken without modification, via the `fuse*` operator. In the end, the transformation in line 16 takes the resulting array with stride two and compacts it to an array with stride one.

```
1  (defun restrict (u)
2    (let ((selection (σ* u (start 2 end) (start 2 end)))
3          (inner (σ* u ((1+ start) 1 (1- end)) ((1+ start) 1 (1- end)))))
4      (→
5       (fuse* (→ u selection)
6        (α #'+
7           (α #'* 0.25   (→ u selection inner))
8           (α #'* 0.125  (→ u (τ (i j) (1+ i) j) selection inner))
9           (α #'* 0.125  (→ u (τ (i j) (1- i) j) selection inner))
10          (α #'* 0.125  (→ u (τ (i j) i (1+ j)) selection inner))
11          (α #'* 0.125  (→ u (τ (i j) i (1- j)) selection inner))
12          (α #'* 0.0625 (→ u (τ (i j) (1+ i) (1+ j)) selection inner))
13          (α #'* 0.0625 (→ u (τ (i j) (1- i) (1+ j)) selection inner))
14          (α #'* 0.0625 (→ u (τ (i j) (1+ i) (1- j)) selection inner))
15          (α #'* 0.0625 (→ u (τ (i j) (1- i) (1- j)) selection inner))))
16      (τ (i j) (/ i 2) (/ j 2)))))
```

Figure 31: A two-dimensional restriction operator

The code in figure 31 is the first example to utilize the → operator to its full potential. It selects all those points which, after the given translation, lie at the even interior points of the grid. All these references are optimized during the creation of the data flow graph, so there is no performance penalty for using such an indirect addressing.

4.3.5 The Multigrid V-Cycle

There are several ways to combine the interpolation, prolongation, smoothing and residual operations to a fully functioning multigrid algorithm. A short and popular variant is the so called *V-cycle*. The V-cycle starts on a level l by applying a fixed number of Red-Black Gauss-Seidel steps. The residual of the thereby improved solution is restricted to the level $l-1$ and used to compute a correction e by solving the system $Ae = r$ with another invocation of the V-cycle. The correction is afterwards added to the current value of x. This corrected solution is then again improved by some number of Red-Black Gauss-Seidel steps. The recursion of the V-cycle ends once a sufficiently low level is reached. Ideally, the small linear system on this level is solved exactly, i.e. by Gauss elimination, but for the sake of simplicity it is

approximated by some number of Red-Black Gauss Seidel iterations. The name "V-cycle" stems from the pattern in which the algorithm traverses the different grid levels. It is shown in figure 32.

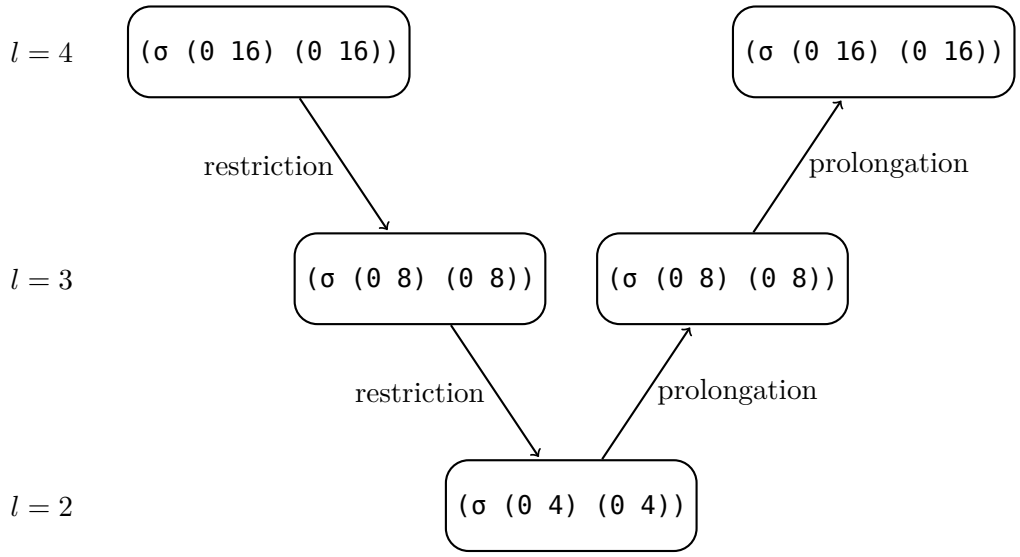


Figure 32: A simplified data flow graph of a multigrid V-cycle

The Petalisp implementation of the multigrid V-cycle so simple that there is no need to write a pseudocode variant of it. It is stated in figure 33. If the target grid has a size of at most 25 points, it is solved with a suitable number of Red-Black Gauss-Seidel steps and the result is returned. Otherwise, an approximate solution x is with derived with `v1` Gauss-Seidel steps (line 4), corrected by the approximated error on the coarser grid (lines 6–14) and finally improved by `v2` Gauss-Seidel steps (lines 5 and 14).

```
1  (defun v-cycle (u f v1 v2)
2    (if (<= (size u) 25)
3        (rbgs u f 5) ; solve "exactly"
4        (let ((x (rbgs u f v1)))
5          (rbgs
6           (α #'- x
7              (prolongate
8               (v-cycle
9                (→ 0.0 (σ* u (0 1 (/ end 2))
10                            (0 1 (/ end 2))))
11               (restrict
12                (residual x f))
13               v1 v2)))
14          f v2))))
```

Figure 33: The multigrid V-cycle

5 Conclusion

Petalisp has been designed as a simple language with only five primitive operations and high potential for optimization and parallelization. Compilers for general purpose programming languages struggle with compile-time uncertainty and complicated control flow, which prevents many optimizations. In Petalisp, these problems vanish. Its programs are massively parallel data-flow graphs and they are compiled on the fly during the actual execution.

A high-quality implementation of Petalisp has been written to study this programming model for real applications. Using this implementation, it has been demonstrated that parallel HPC algorithms can be written concisely as Petalisp programs. The given examples range from simple matrix multiplications to a complete multigrid algorithm. An unforeseen consequence of the focus on powerful parallel operations is that it is remarkably pleasant to write Petalisp programs. Algorithms that would require many nested loops and local variables in imperative languages can be expressed with only few lines of Petalisp code. Neither expressiveness, nor elegant notation have been a primary design goal of Petalisp. The original idea was to find a fundamental notation for parallel programming, which is well suited for compilers. It is a relief that the resulting programming language also turned out to be convenient for humans.

One final ingredient is still missing before the language can be considered ready for mainstream adoption: A parallelizing compiler back end. At the moment, all Petalisp programs are run in serial. The work on this subject has been postponed while the semantics of the language is still under development. More algorithms, e.g. Gaussian elimination, need to be written in Petalisp to determine whether the current set of features is really sufficient for all important problems. Once this work is done, it is possible to proceed to the next chapter of Petalisp development and investigate parallelization strategies. Any reasonable parallel scheduler requires detailed knowledge of the given system resources, so a big portion of this work will be devoted to performance introspection. The scheduler needs a way to detect the number of available compute nodes, CPU cores, main memory size, cache hierarchies and communication channels. Afterwards this information can be combined with performance models (e.g. Roofline [43] or ECM [44]) to make qualified scheduling and domain partitioning decisions. The ambitious long-term goal is that Petalisp generates code that outperforms even the handwritten code of expert HPC programmers.

In the end, using Petalisp is a trade-off. The programming model is completely different from existing approaches and requires some familiarity with the programming language Common Lisp. The benefits, however, are significant. Petalisp programs are short and maintainable and can be developed interactively, with almost instantaneous feedback. The whole multigrid algorithm in section 4.3 was written in less than a day, including many example problems to test the individual components for correctness. Another benefit is that the notation is completely hardware independent and hence portable. Nothing prevents Petalisp from utilizing GPUs or other specialized hardware, even if this means that the compiler back end has to generate code in different programming languages for each hardware device.

Petalisp is free software and available at www.github.com/marcoheisig/Petalisp. Readers are encouraged to experiment with the software and study the attached example programs. Feedback is most welcome.

References

[1] G. Steele, *Common LISP. The Language. Second Edition.* Digital Press, 1990.

[2] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr, D. H. Bartley, R. Halstead, *et al.*, "Revised5 report on the algorithmic language scheme," *Higher-order and symbolic computation*, vol. 11, no. 1, pp. 7–105, 1998.

[3] http://www.openmp.org/mp-documents/openmp-4.5.pdf. accessed 3 Sep. 2016.

[4] P. Seibel, *Practical Common Lisp.* Apress, 2005.

[5] P. Graham, *ANSI Common LISP.* Pearson, 1995.

[6] C. Barski, *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* No Starch Press, 2010.

[7] D. S. Touretzky, *Common LISP: A Gentle Introduction to Symbolic Computation (Dover Books on Engineering).* Dover Publications, 2013.

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

[9] P. Norvig, *Artificial Intelligence Programming.* Morgan Kaufmann, 1992.

[10] P. Graham, *On Lisp: Advanced Techniques for Common Lisp.* Prentice Hall, 1993.

[11] D. Hoyte, *Let Over Lambda.* Lulu.com, 2008.

[12] C. Queinnec, *Lisp in Small Pieces.* Cambridge University Press, 2003.

[13] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[14] www.tiobe.com/tiobe-index/. accessed 3 Sep. 2016.

[15] www.githut.info. accessed 3 Sep. 2016.

[16] https://pypl.github.io/PYPL.html. accessed 3 Sep. 2016.

[17] P. J.-L. Lions, "Ariane 5 — flight 501 failure report by the inquiry board." 1996.

[18] J. Bloch, "Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken." https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html, 2006. accessed 3 Sep. 2016.

[19] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, pp. 2:1–2:29, Dec. 2015.

[20] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[21] W. Taha and T. Sheard, "Metaml and multi-stage programming with explicit annotations," *Theoretical computer science*, vol. 248, no. 1, pp. 211–242, 2000.

[22] B. Stroustrup, "Evolving a language in and for the real world: C++ 1991-2006," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, (New York, NY, USA), pp. 4–1–4–59, ACM, 2007.

[23] "Dr. Alan Kay on the Meaning of Object-Oriented Programming." http://www.purl.org/stefan_ram/pub/doc_kay_oop_en. accessed 5 Sep. 2016.

[24] https://github.com/marcoheisig/the-cost-of-nothing. accessed 27 Nov. 2016.

[25] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *CoRR*, vol. abs/1411.1607, 2014.

[26] R. Hickey, "The clojure programming language," in *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, (New York, NY, USA), pp. 1:1–1:1, ACM, 2008.

[27] https://common-lisp.net/project/bordeaux-threads. accessed 12 Dec. 2016.

[28] https://github.com/m2ym/optima. accessed 27 Nov. 2016.

[29] http://franz.com/products/allegro-common-lisp/. accessed 27 Nov. 2016.

[30] http://www.lispworks.com/. accessed 27 Nov. 2016.

[31] http://ccl.clozure.com/. accessed 27 Nov. 2016.

[32] http://sbcl.org/. accessed 27 Nov. 2016.

[33] http://clisp.org/. accessed 27 Nov. 2016.

[34] https://www.gnu.org/software/gcl/. accessed 27 Nov. 2016.

[35] http://abcl.org/. accessed 27 Nov. 2016.

[36] K. E. Iverson, *A Programming Language*. New York, NY, USA: John Wiley & Sons, Inc., 1962.

[37] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee, *SISAL: streams and iteration in a single-assignment language. Language reference manual, Version 1. 1.* Jul 1983.

[38] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[39] https://github.com/marcoheisig/Petalisp. accessed 12 Dec. 2016.

[40] D. E. Knuth, *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

[41] G. L. Steele, Jr. and W. D. Hillis, "Connection machine lisp: Fine-grained parallel symbolic processing," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, (New York, NY, USA), pp. 279–297, ACM, 1986.

[42] W. Briggs, V. Henson, and S. McCormick, *A Multigrid Tutorial: Second Edition*. EngineeringPro collection, Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2000.

[43] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[44] H. Stengel, J. Treibig, G. Hager, and G. Wellein, "Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model," *CoRR*, vol. abs/1410.5010, 2014.