Towards Practical High-Quality Software

Benjamin Chetioui

Thesis for the degree of Philosophiae Doctor (PhD)
at the University of Bergen

Part I

Overview

Scientific environment

I was employed by the University of Bergen for the entire duration of my PhD research fellowship, and carried out my research at the university's Department of Informatics.

I was enrolled in the Research School of Computer and Information Security (COINS).

The research presented in this dissertation has benefited from the Experiment Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

Acknowledgements

I have received far too much support from far too many people to be able to give everyone the credit they're due. Nevertheless, I shall attempt to do so.

My gratitude goes first and foremost to my supervisors, Jaakko Järvi and Magne Haveraaen. Both of them were endlessly supportive, and extremely generous with their time throughout the entire course of their supervision. I learned a lot from our three-way collaboration—in terms of the subject matter itself, in terms of good academic writing, and simply in terms of what good research looks like.

I want to thank my co-authors, Ole Abusdal, Mikhail Barash, Mathilde Boltenhagen, Alessandro Budroni, Ermes Franch, Laurine Huber, Marius Kleppe Larnøy, Sandra Macià, and Lenore Mullin. Though not all the papers we worked on together are part of this dissertation, they were certainly all a part of the journey recounted here. Particular thanks go to Lenore Mullin and Mikhail Barash, who offered me not only their co-authorship, but also their mentorship.

The quality of my papers was systematically increased thanks to the contribution of my peer reviewers—both before and after submission! I want to extend my gratitude to Andrea Tenti, Christer Steinfinsbø, Emmanuel Arrighi, Håkon Robbestad Gylterud, James H. Davenport, Jeremy Gibbons, Martha Norberg Hovd, Norman Rink, Sofie Hopland, Wrya Kadir, and all my anonymous reviewers.

I am glad I chose to pursue my research fellowship at the University of Bergen, where I was surrounded by tons of amazing colleagues, whether at the University itself, at Simula UiB, or at HVL. Though I won't name everyone here, I need to call out my officemates Jonathan Prieto-Cubides and Tam Thanh Truong. Sharing an office with them made work fun (almost) every day.

I was also lucky to be able to take a break from my research fellowship to spend time working with the JAX team at Google. This experience significantly influenced my approach to research in the second half of my fellowship. I thank the team for helping me grow as a researcher—and in particular George Necula for his daily mentorship.

Among the friends I haven't already named, I'd like to also call out Maxime Augier, Aurélien Billot, Julien Gamba, and Roxana Pop—thank you for having been constant sources of support over the past 7 years. This dissertation has also been shaped by you, in some ways.

Merci maman et papa pour votre soutien sans faille, et pour avoir tout fait pour me donner les opportunités d'arriver jusqu'ici.

Abstract in English

With society becoming more and more reliant on software, the importance of software quality is becoming ever more clear. Poor quality software is estimated to have cost $2.41 trillion to US society in 2022. Poor quality software is software that rates poorly along dimensions such as *security* (data cannot be compromised), *reliability* (the program performs as intended), or *performance* (the program performs its intended task efficiently).

If poor quality software is so staggeringly expensive, why do developers not produce high quality software to begin with?

Experience has shown that developing high quality software tends to be prohibitively expensive. Lack of proper tooling and the difficulty of the construction of correctness proofs seem to be the main factors behind the high costs.

This thesis studies generic programming and the Magnolia programming language for constructing high quality software—and as a vehicle for driving down the cost of this activity. Magnolia proposes to bridge the gap between API specifications and their actual implementation by allowing to express both of them in a common language. The means to express mechanized formal specifications and their relation to implementations are core to producing proofs of correctness. They also enable high software performance by allowing the expression and automated application of semantics-preserving rewrites that can optimize programs. Magnolia is built with genericity as its core design goal. Magnolia's code can be parametrized freely, and it thus enables the construction of highly parametrized libraries whose algebraic properties are precisely specified. This encourages designing generic, reusable APIs. Code reuse is a tried-and-true method to directly increase software quality and reduce development costs. Code reuse allows proof reuse, which amortizes the cost of constructing proofs over time.

We contribute a formal specification of the Magnolia programming language, and conduct a rigorous study of how Magnolia's approach to language design fares in the landscape of generic languages. Magnolia also emphasizes that genericity and reusability of code are not antithetical to performance; its programming model is particularly well-suited to high-performance computing. We apply generic programming with Magnolia to the domain of array programming through a triplet of case studies culminating in the specification of formally verified normalization rules and their guided application as program optimizations on array programs for several hardware platforms.

Abstract in Norwegian

Høy programvarekvalitet er viktig. Dette blir tydeligere ettersom samfunnet blir stadig mer digitalisert. For eksempel anslås dårlig programvarekvalitet å ha kostet det amerikanske samfunnet $2.41 billioner i 2022. Dårlig programvare svikter på sikkerhet (data blir kompromittert), pålitelighet (programvaren fungerer ikke hensiktsmessig) og/eller ytelse (programvaren er ineffektiv).

Hvis dårlig programvare koster samfunnet så mye, hvorfor lages ikke høykvalitetsprogramvare i utgangspunktet?

Utvikling av høykvalitetsprogramvare har lenge vært uhensiktsmessig dyrt. Mangel på riktig verktøy og utfordringer med å utvikle bevis, er viktige faktorer bak de høye kostnadene.

Denne avhandlingen studerer generisk programmering og programmeringsspråket Magnolia som virkemiddel for å utvikle programvare av høy kvalitet—og for å få ned kostnadene for dette. Magnolia bygger bro over det semantiske gapet mellom API-spesifikasjoner og -implementering ved å bruke samme språk for begge. Dette løser et kjernepunkt for programbevis: en presis relasjon mellom formelle spesifikasjoner og implementasjoner. Magnolias algebraiske spesifikasjoner kan også brukes til automatiserte semantikkbevarende omskrivninger for å øke programvarens ytelse. Magnolia sin språkstruktur er grunnleggende generisk slik at kode kan parametriseres fritt. Til sammen åpner dette for høyt parametriserte kodebiblioteker med presist spesifiserte algebraiske egenskaper, som oppmuntrer til design av generisk, gjenbrukbar programvare. Gjenbruk av kode er en utprøvd metode for å direkte øke programvarekvaliteten og redusere utviklingskostnadene. Gjenbruk av kode gir også gjenbruk av korrekthetsbevis, som fordeler kostnadene ved å utvikle bevisene på mange applikasjoner.

Avhandlingen bidrar med en formell spesifikasjon av programmeringsspråket Magnolia, og gjennomfører en nøyaktig utforskning av hvordan Magnolias tilnærming til språkdesign ser ut i landskapet av generiske programmeringsspråk. Arbeidet understreker også at generisitet og gjenbruk av kode ikke er i motsetning til høy ytelse; Magnolias programmeringsmodell er tvertimot spesielt godt egnet til dette. Avhandlingen tar for seg tre case-studier i Magnolia for problemstillinger fra tungregning. Tungregning krever høy ytelse på store datamengder representert i array-strukturer. Avhandlingen kulminerer i formelt verifiserte algebraiske regler og deres anvendelse for optimalisering av array-programmer mot ulike maskinvareplattformer.

List of publications

1. Benjamin Chetioui. magnoliac: A Magnolia compiler, Dec. 2020. doi:10.5281/zenodo.6572953
2. Benjamin Chetioui, Jaakko Järvi, and Magne Haveraaen. Revisiting language support for generic programming: When genericity is a core design goal. *The Art, Science, and Engineering of Programming*, 7(2), oct 2022. doi:10.22152/programming-journal.org/2023/7/4
3. Benjamin Chetioui, Lenore Mullin, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Sandra Macià. Finite difference methods fengshui: Alignment through a Mathematics of Arrays. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, page 2–13, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367172. doi:10.1145/3315454.3329954
4. Benjamin Chetioui, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Lenore Mullin. Padding in the Mathematics of Arrays. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2021, page 15–26, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384667. doi:10.1145/3460944.3464311
5. Benjamin Chetioui, Marius Larnøy, Jaakko Järvi, Magne Haveraaen, and Lenore Mullin. P^3 problem and Magnolia language: Specializing array computations for emerging architectures. *Frontiers in Computer Science*, 4, 2022. doi:10.3389/fcomp.2022.931312

Contents

Chapter 1

Introduction

Poor quality software is expensive for society. A report by the Consortium for Information & Software Quality (CISQ) prices the whole IT labor base in the US for 2022 at roughly $1.51 trillion [114]. The same report evaluates that poor quality software costs at least $2.41 trillion for the US society for the same year—i.e., about 1.6 times as much as the whole IT labor base itself. The report points to *operational failures* and *legacy systems* as the two largest contributors to this overall figure, with an aggregate cost in the ballpark of $2.3 trillion.

The quality of software can be measured along many axes. Common desired software traits include *security* (data cannot be compromised), *reliability* (the program performs as intended), and *performance* (the program performs its intended task efficiently). Poor quality software is software that rates poorly along one or more of these axes. Ever-increasing cybercrime losses incurred due to existing software vulnerabilities constitute a large part of the operational failures in the CISQ's report. The prevalence of software vulnerabilities points to a lack of security or reliability. Similarly, part of the cost associated with legacy systems stems from poor software performance. If poor quality software is so expensive, why do developers not produce high quality software to begin with?

In 2014, Klein et al. published a well-known case study in producing real-life high quality software through the development of the seL4 microkernel [111]. The microkernel is fully verified for functional correctness, comes with security proofs, and is highly performant—between a factor of two and a factor of ten faster than other microkernels [157]. The detailed cost analysis provided in the paper indicates that developing the microkernel took 2.2 person years, while writing a proof of correctness for that implementation took 20.5 person years—a staggering 932% increase in development costs [111]. Producing proofs of correctness is the only way to guarantee that software does precisely what it is meant to do[1].

In addition to these initial formal verification costs, maintaining the correctness of the proofs throughout the evolution of the seL4 software likewise proved to be expensive and difficult. Elphinstone and Heiser—two of the authors of seL4—go as far as saying that *"the formal verification of seL4 creates a powerful disincentive to changing the kernel"* [52]. Maintaining formally verified software is widely considered to be prohibitively expensive [48; 72; 150].

In summary, the seL4 authors produced highly performant software with strong correctness

[1] Up to what can be modeled and the trusted computing base.

guarantees; it was roughly 10 times more expensive to develop than comparable software without correctness proofs, and it is expensive to maintain. While the figures discussed here are anecdotal and come with many asterisks, the point remains: producing high quality software seems to be *prohibitively expensive*.

We formulate the—hopefully uncontroversial—hypothesis that given the means to produce and maintain high quality software cheaply, developers would do so. The follow-up question comes naturally: how can we reduce the costs associated with producing and maintaining high quality software?

Mechanized specifications are the gateway to correctness proofs. As Dullien puts it, it is only possible to conclude that software is flawed if one understands how a correct version of the software would behave [51]—i.e., if one understands how the implementation of the software deviates from its intended specification. By the same definition, software that conforms to its specification can be deemed correct. Mechanized specifications allow the production of machine-checked proofs of correctness [118], and also play an important role in enabling high software performance by allowing the expression and automated application of semantics-preserving rewrites that can optimize programs [119].

Despite these benefits, software is rarely written with associated formal specifications. Even when these formal specifications exist, they are rarely mechanized—e.g., typeclass laws in Haskell are typically stated only as documentation [10].

The seL4 report evaluates the cost of developing an abstract and executable specification for the microkernel to be a mere 7 person months. Given the benefits outlined above, it seems unlikely that this moderate additional cost is the main reason behind the lack of formal specifications in mainstream software.

Another more likely explanation for the low incidence of formal specifications in software could be the lack of accessible and good tooling for harnessing their potential benefits. "Mainstream" programming languages provide little to no support for expressing sophisticated mechanized specifications or relating implementations to specifications. And indeed, 9 out of the 20.5 person years ($\approx$ 40%) invested in developing the proofs of correctness for seL4 were spent improving frameworks, tools, and libraries for proof automation and theorem proving.

Assume the seL4 team had had a programming language that had provided the means to express mechanized specifications, a convenient way to relate them to implementations, and all the necessary library support so that tooling would not have been a concern. They would still have spent 11.5 person years developing their correctness proofs—a significantly lower but still prohibitively high cost.

One way to approach making the development of proofs cheaper is to enable reuse; code reuse is a tried-and-true method to directly increase software quality and reduce development costs [23; 56; 138]. While the high costs associated with developing ad-hoc proofs will still have to be paid initially, reuse across projects will allow amortizing this cost—whether across several products for a given company or across independent projects [140].

Generic programming is a discipline that enables code reuse, by cleanly structuring and organizing abstract data types and algorithms around the key algebraic properties they (or their dependencies) must fulfill. Generic programming seeks to distill algorithms to their essence, expressing them in a form that makes as few assumptions as possible about the interface and behaviour of the data types

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.
- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized to the concrete case, the result is just as efficient as the original algorithm.
- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.
- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.

Figure 1.1: Definition of generic programming from Jazayeri, Musser, and Loos [102].

involved. In following this methodology, a programmer naturally builds parametrized and composable modular building blocks—reusable code. Through providing good language support for generic programming on the one hand and for expressive specifications on the other hand, we can realize the cost-reduction opportunities outlined above and pave the way towards correct, highly efficient software.

Interpretations of generic programming vary depending on what kind of parametrization a programming language supports [62]. We follow here the definition given by Musser and Stepanov in their seminal work in 1988 [137]. Figure 1.1 reproduces their structured definition of generic programming, taken from Jazayeri, Musser, and Loos [102]. This definition of generic programming, unlike many others, points to the preservation of runtime performance as an explicit goal for any lifted abstraction.

1.1 Line of Research

This work is centered around studying Magnolia [14; 35], a programming language that aims to enable *generic programming* as its core design goal. Magnolia is designed with a module system based on Goguen and Burstall's theory of institutions [67]: it allows defining both specifications with syntactic and semantic requirements on implementations, and corresponding implementations in the same core language. Implementations and specifications can be related to one another by declaring modeling (or satisfaction) relations [38]. This makes Magnolia in principle well-suited as a vehicle towards less expensive verified, highly performant software. As outlined above, formal specifications also pave the way for improving software performance.

By offering parametrization involving both syntactic and semantic requirements, Magnolia falls into the category of languages offering *genericity by property*—as defined in Gibbons' taxonomy of generic programming [62]. Despite its seeming attractiveness, fully-fledged support for genericity

by property—i.e. support for property-based specifications—has not managed to find its way into mainstream programming languages so far.

Genericity by property is closely related to algebraic specifications. Algebraic specifications are at the core of Stepanov's work on generic programming [49; 104; 137; 165]. It is thus no surprise that the mainstream programming language that came closest to concretizing genericity by property was C++, through *concepts*, as envisioned in C++0x. Concepts ended up materializing in C++20 as a scaled back version of C++0x's: with support for syntactic (but not semantic) requirements, and no (early) type checking of generic definitions.

As a result of this lack of support, much of the literature studying generic programming in practice does not discuss property-based specifications—and the feature is not considered key to enabling generic programming [61; 163]. This is somewhat surprising, since genericity by property seems to be, by design, the most natural way of supporting generic programming. Clearly, mainstream programming languages must be passing on opportunities for reuse by not offering this axis of parametrization.

The work presented in this dissertation seeks to explore more deeply the interplay between property-based specifications and more "classic" support for generic programming. Through rigorous experiments, we confirm that Magnolia's design choices are conducive to effective generic programming—despite the language not offering many of the features previously thought to be key for enabling generic programming [38]. This points to these features being more means to an end rather than true requirements. Maybe unsurprisingly, the success of these experiments does not hinge at all on Magnolia's property-based specifications. It is instead better explained by Magnolia's powerful parametrized module system, which is sufficient to express syntactic requirements as in C++20 concepts. We seek then to understand what additional opportunities for reuse property-based specifications offer. For instance, Hamre already managed to exploit them for automated code verification [83].

The remaining of the exploration highlights the importance of property-based specifications for generic programming, by bringing in focus additional kinds of reuses that they enable in the context of software performance. We employ domain engineering techniques [84] to investigate the domain of array programming for stencil computations, formally grounded in the Mathematics of Arrays (MoA) formalism [127]. Property-based specifications serve first to help discover the right application programming interface (API) to express and optimize stencil computations [28; 36; 37]. Later, we also reuse these specifications to build term rewriting systems able to optimize stencil computations for several hardware platforms, and to automatically derive default implementations for semantically constrained operations [39]. As we design and extend programming languages to offer additional opportunities for code reuse, we will make building high quality software more practical. In this endeavour, Magnolia's module system and property-based specifications can be sources of inspiration.

1.2 Outline of the Dissertation and Detailed Contributions

The dissertation consists of three parts—titled *"Overview"*, *"Scientific Results"*, and *"Reflections"*.

- **Part I** provides an overview of the work pursued in this dissertation and of Magnolia. It consists of two chapters:
  - **Introduction** is the ongoing chapter.

  - **Magnolia and magnoliac**—which follows this introduction—provides a never published before formal description of the syntax and static semantics of the Magnolia programming language as it is concretized in magnoliac. Detailed presentations of the various constructions accompany the formalization. The chapter also offers insights into magnoliac itself, and concludes with explanations for some notable implementation choices made over the course of its development.

- **Part II** contains the bulk of our scientific results, and reproduces the content of four (already published) research papers:

  - **Revisiting Language Support for Generic Programming: When Genericity Is a Core Design Goal (Paper 1)** applies a well-known framework for evaluating the generic programming capabilities of a language [61] to assess Magnolia's own generic facilities. Through a reimplementation of several algorithms from the Boost Graph Library [164], we confirm that the language lends itself well to effective generic programming. Along the way, we also discover that features previously thought to be key to effective generic programming are more of a means to an end, and introduce two new important features for generic programming: *variadics*, and *property-based specifications*. The paper introduces *magnoliac*, a new Magnolia compiler made available as an artifact of the paper[2] [35]. The version of the magnoliac compiler presented there is able to use either C++ and Python as a host language. The code for the experiments is likewise made available in the same repository[3].
  - **Finite Difference Methods Fengshui: Alignment through a Mathematics of Arrays (Paper 2)** proposes a framework for optimizing scientific array-based computations using the MoA formalism. Burrows et al. previously made use of Magnolia's property-based specifications as a domain engineering tool to discover the necessary API to express stencil computations [28]. This paper restates this API using MoA, and contributes a rewriting system sufficient to bring stencil computations to their Denotational Normal Form (DNF)—their abstract canonical representation. This rewriting system is shown to be canonical—i.e. strongly confluent and terminating—and is well-suited as the basis for automatic optimizations. We embed the MoA formalism in Magnolia using its property-based specification facilities and contribute generic code for the studied stencil computations.
  - **Padding in the Mathematics of Arrays (Paper 3)** extends the framework proposed in Paper 2 to perform hardware-specific optimizations for array computations. The paper contributes a full formalization of exterior padding in the MoA formalism, and derived rewriting rules suitable for introducing halos in both single and multi-process settings. The paper demonstrates the relevance of these rewriting rules for determining the right trade-off between inter-process communication and data/computation redundancy through a set of experiments. The experiments and an incomplete mechanization of the proofs from the paper are made available on GitHub[4] [5].
  - **P³ Problem and Magnolia Language: Specializing Array Computations for Emerging Architectures (Paper 4)** exploits the results of Paper 2 and Paper 3 to build an optimizer for stencil computations in Magnolia. The paper contributes two new

[2] https://github.com/magnolia-lang/magnolia-lang.
[3] https://github.com/magnolia-lang/magnolia-lang/tree/main/examples/bgl.
[4] https://github.com/mathematics-of-arrays/padding-in-the-mathematics-of-arrays.
[5] https://github.com/mathematics-of-arrays/moa-formalization.

module transformations in Magnolia that enable new ways of exploiting its property-based specification facilities: *rewrite*, and *implement*. The *rewrite* transformation allows to produce term rewriting systems from formal specifications expressed in Magnolia and to use them to automatically rewrite implementations. The *implement* transformation allows to derive function implementations from these formal specifications. We implement the core specifications developed in Paper 2 and Paper 3 in Magnolia, and complement the rewriting system with hardware-specific rewriting rules. We demonstrate through experiments that the resulting optimizer allows to run the same program efficiently across different types of hardware (CPU and GPU in the experiments). In order to allow for running programs on GPU, we also add support for using CUDA as a host language to magnoliac. The compiler extensions and experiments are available in a branch of the repository for magnoliac on GitHub[6] [7].

- **Part III** concludes the dissertation. It consists of a single chapter:
  - **Concluding Remarks** ties the bow on this dissertation with a reflection on the learnings gleaned throughout this work. The chapter outlines some possible future research and engineering directions around Magnolia and property-based specifications.

The papers mentioned above are included in the following as they were published, save for stylistic improvements to fit the format of this dissertation. The appendices for Paper 1 and Paper 4 are aggregated at the end of the thesis, and there is a single bibliography for the whole dissertation.

[6] https://github.com/magnolia-lang/magnolia-lang/tree/base-program.
[7] https://github.com/magnolia-lang/magnolia-lang/tree/base-program/examples/pde.

Chapter 2

Magnolia and magnoliac

2.1 Motivation

The Magnolia programming language was originally designed to complement the hard-to-extend C++ as a vehicle for exploring novel language features for programming numerical software in Bagge's work [14, Chapter 1.3]. The language was designed to be semantically close to C++ and the compiler emitted C++ code—so as to both retain full control over resource usage and remain compatible with existing High-Performance Computing (HPC) compilers. Magnolia was built to be

1. easy to transform and extend;
2. able to harness the power of the existing C++ high-performance compilers;
3. able to express HPC-oriented optimisations (i.e. produce HPC-grade programs at the end of the compilation process).

The overarching goal for exploring these novel language features through Magnolia has always been to propose and support a development method geared towards developing high quality software [14, Introduction]. Key attributes of high quality software include *reliability*, *robustness*, *flexibility*, *portability*, and *efficiency*. Magnolia attempts to provide the keys to building high quality software by providing powerful generic facilities through a flexible intermix of specifications and implementations. As alluded to in the introduction, Magnolia makes enabling Stepanov-style generic programming [102] its core design goal. Bagge emphasizes the importance of *abstraction* (or *generalization*) for high-quality software development to be a fundamental insight guiding the whole approach.

Magnolia naturally grew over time to implement many experimental language features, such as *functionalisation* and *mutification* [13], *generated types*, *type partitions*, and more. While this proliferation of language features was consistent with the stated goals of Magnolia, it was not accompanied by any kind of centralized documentation—providing a full specification was considered to be a hindrance to the stated objective of exploring and implementing novel language features. When picking Magnolia up again many years later, this context—or lack thereof—made developing a shared understanding of what Magnolia was extremely difficult. We were—ironically—bitten by our own reluctance to build formal specifications!

The first implementation of a compiler for Magnolia was extremely slow. This is largely due to a design

that required constructing and traversing chunks of the underlying Abstract Syntax Tree (AST) an inordinate number of times [16]. As a result, the compilation of the original Magnolia basic library (around 8000 lines of code) took over 30 minutes. The compiler was also specifically constructed to work as a plugin for the Eclipse IDE, and did not expose facilities for command-line compilation. This excluded developing Magnolia code in other IDEs, and later on caused issues when—due to dependencies on older packages and lack of maintenance—the compiler became accessible only through many-years-old versions of Eclipse.

This lack of formal specification and difficulty of using the implementation motivated the development of *magnoliac*, a new command-line compiler for Magnolia [35]. Our goals and philosophy for developing magnoliac ended up being as follows:

1. magnoliac needs to be decoupled from any particular IDE, and to support command-line compilation;
2. magnoliac needs to be reasonably fast;
3. magnoliac must be easy to reuse and extend by future students and researchers. That is to say that the compiler must be documented well, and clear extension points must be defined;
4. magnoliac will produce good and well-located error diagnostics;
5. magnoliac will emit structured code easily parsable by humans. This so that generated code can be more or less edited by hand and included in a project without making that project depend on Magnolia. This additionally allows debugging performance issues (among others) easily in the generated code;
6. magnoliac will allow compiling Magnolia code to several target languages, including C++;
7. every non-trivial language design choice must be documented and justified;
8. every new feature must be implemented in its most restricted useful form unless there is a (justifiably) better choice. It is possible to relax strict restrictions on a feature made *a priori* and to remain backwards compatible, but the opposite (restricting features *a posteriori*) is not.

Though we initially intended for magnoliac to become able to successfully compile the original Magnolia basic library, this effort was hindered by the lack of available documentation. As it became clear that some of the features necessary to accomplish this would be difficult to design in a principled and usable way, we gave up on this goal. To this day, the compiler presented here is only compatible with a subset of the code that the initial compiler accepted. This is largely by design, but also due to our work over the last few years focusing only on a subset of the previously implemented features.

In the absence of a previous specification, the language becomes most accurately defined by its compiler implementations. As a result, we should be cognizant that the Magnolia programming language as presented in the following is a reflection of magnoliac's implementation—which diverges from the previous compiler implementation, but doesn't betray the spirit of the language. This thesis does not redesign the Magnolia programming language, but merely tweaks and clarifies some aspects of it. Specifying the language and building a compiler for it were pre-requisites for the rest of the work in this thesis, on MoA, and on generic programming.

We first give a short primer on the Magnolia programming language. We follow up with a rigorous presentation of the language, complete with a formal syntax and accompanying typing rules. Magnoliac's implementation of the language is extremely close to the formal description provided here, but there are slight variations. This is because this formal description postdates the initial development of the compiler, and uncovers a few minor weaknesses in magnoliac. These discrepancies will be resolved when the necessary changes make it into the compiler's code base. Lastly, we briefly

discuss some of the interesting implementation choices made in magnoliac.

The current chapter is highly technical—Paper 1 synthesizes core Magnolia concepts more digestibly.

2.2 A Primer on Magnolia

The content of this primer largely follows that of our paper *"Revisiting Language Support for Generic Programming: When Genericity Is a Core Design Goal"* [38] (Paper 1 of the present dissertation).

Algebraic specifications are at the core of Stepanov's work on generic programming [49; 104; 137; 165]. Highly influential early work in the field is Goguen's parameterized programming that emphasizes code reuse and modularity [60; 63]. Siek characterizes parameterized programming as similar to Stepanov's notion of generic programming, but without the same emphasis on efficiency [159]. Parameterized programming thus also aims at expressing algorithms in their most general form, making both their syntactic and semantic requirements explicit, and well organized.

As the embodiment of a language for Stepanov-style generic programming, Magnolia's lineage naturally traces back to parameterized programming, and to an approach to language design rooted in algebraic specifications.

2.2.1 Designing for Generic Programming with Algebraic Specifications

Algebraic specifications and Goguen and Burstall's theory of institutions [67] have guided the design of the OBJ language family [71] (OBJ2, OBJ3, CafeOBJ, Maude…). The OBJ language family was (and still is) a highly influential representative for programming languages based on the algebraic approach. We use these languages to introduce various important concepts, and later explain how the same concepts manifest in Magnolia.

Languages in the OBJ family provide extensive support for parameterized programming by design. OBJ2 and OBJ3 are both implementations of the OBJ logical programming language that differ in their operational semantics [70]. Maude incorporates most features of OBJ3 and significantly expands the capabilities of OBJ2 and OBJ3 for parameterized programming. Maude and CafeOBJ are still under active development. We describe below the general design of languages intended to support generic programming using algebraic specifications, and explain how it is concretized in Maude. Maude is based on rewriting logic [42; 66], and uses *membership equational logic* as its underlying equational logic. Our discussion only touches upon the fragment of Maude related to membership equational logic, where Maude's support for parameterized programming is concretized. Note that grounding Maude in rewriting logic is purely a design choice, and not a requirement for institution-based languages.

The general approach relies on a bilevel module system, with modules that allow for specifying generic APIs on the one hand and modules that allow for writing concrete programs on the other hand [69]. Modules of the same kind may be composed, and program modules can be parameterized by specification modules. Specifications consist of an algebraic signature defining sorts and (total and partial) operations, along with semantic requirements on their behaviour called *axioms*. Satisfaction relations can be expressed which describe how a program (or a specification) satisfies the requirements

of a given specification.

Specifications are given in Maude through *functional theories*—Goguen introduced the notion of types as theories [65]. Functional theories allow expressing semantic requirements using *equations* and *conditional equations*. In addition, Maude allows the specification of *subsorting relations* along with *membership axioms*. This approach allows flexible control of partiality and declaring relationships between types, e.g., natural numbers and integers.

Maude's *functional modules* allow for writing programs using the same constructs as functional theories—where equations and conditional equations define functions and data types in lieu of functional theories' semantic requirements, and where the rewriting system engendered by these equations must be confluent and terminating. The semantics of a functional module in Maude is the initial algebra defined by the module's equations, and evaluation is performed using an equational rewriting engine. Functional modules can be parameterized by functional theories: we speak of *parameterized functional modules*. Maude programs can be metarepresented as data and manipulated to produce new programs. This powerful mechanism of reflection allows generating so-called *dependent parameterized modules* such as n-tuples containing n sorts and n projection functions [45, Section 21.3.1]. Maude's built-in types are efficiently implemented in C++. Contrarily to the previous OBJ2 and OBJ3, Maude does not allow the user to implement custom primitive types in an external language.

Satisfaction relations in Maude are stated through *views*. Every sort (respectively function) in the view's source theory must be mapped (renamed) to a corresponding sort (respectively function) in the view's target module, and the mappings must preserve the subsorting structure of the source theory in the target module. It is also possible to implement functions on the fly to resolve signature mismatches.

2.2.2 Magnolia in a Nutshell

As alluded to above, the Magnolia programming language is designed for Stepanov-style generic programming—i.e. parameterized programming with an added emphasis on efficiency. The language takes the same general approach based on algebraic specifications as described above, and its module system is likewise based on Goguen and Burstall's theory of institutions.

Listing 2.1 shows uses of the different module types. A **signature** allows defining types and operations. A **concept** is a **signature** augmented with **axiom**s that restrict the properties of the types and operations. A **concept** serves the same purpose as a functional theory in Maude, and the **signature** and **concept** modules constitute the specification layer of the module system. An **implementation** allows the same declarations as a **signature**, but also the definition of generic operation implementations; it is the equivalent of a parameterized functional module in Maude. A **program** is a specific kind of **implementation** in which all the specified operations and types are matched with (non-generic) concrete implementations; either Magnolia code that has a concrete implementation or an implementation in the base library in the host language. The **implementation** and **program** modules constitute the program layer of the module system. Constructs analogous to Maude's metaprogramming facilities are under investigation for Magnolia [87].

Types (sorts) in Magnolia are opaque identifiers. One cannot explicitly parameterize them, nor can one define relations such as subtyping relations between them. Types and operations that

construct or destructure them need to come from a host language. Operations can be **function**s, **procedure**s, or **predicate**s. Procedure calls are prefixed with the **call** keyword, while function calls follow the usual uncurried call syntax. Predicates are treated as functions with a built-in, non-reimplementable return type. Magnolia's approach to partiality is based on guarded algebras [88]: an operation can be guarded by a predicate, which then acts as a precondition. In addition to their types, a **procedure** associates modes to its arguments: **obs** (read-only), **upd** (can be read and written to), and **out** (write-only, and must be written to) [18]. `ExampleProgram` in Listing 2.1 shows two implementations of a multiplication by three, one as a **procedure** (`timesThreeUpdateRef`) and the other as a **function** (`timesThree`). In the example's program, the `int` type and `add` function are externally defined in Python and come from `PyConcreteSemigroup`. The line **use** `Magma[ T => int, bop => add ]` applies a renaming function to the content of the `Magma` **signature** and brings it into scope. The renaming maps `T` to a new name `int`, and `bop` to a new name `add`. It is assumed that the primitives implemented in the host language do not have side-effects, except for the modification of arguments to procedures where the argument mode is **out** or **upd**.

A **satisfaction** allows defining a modeling relation between an **implementation** and a **concept**; or between two **concept**s—it is the equivalent of a view in Maude. Signature mismatches are resolved through the renaming mechanism.

Magnolia semantics are tightly coupled to abstracting over hardware features: primitive types and operations may directly represent characteristics of the underlying hardware architecture, such as instruction sets, memory layout, etc. This enables Magnolia code to run efficiently on a variety of hardware, and to explore software for high-performance computing (HPC) [39]—making it suitable to address also the efficiency aspect of generic programming. This feature enables the user to utilize features of new hardware, e.g., posit numbers [80] by writing code directly in the targeted host language. Magnolia's notions of *statements* and *expressions* have semantics compatible with their counterpart in e.g. C++ and Python.

The notion of concepts, around which specifications in Magnolia are constructed, is from Stepanov and Musser [137].

Listing 2.1: The main Magnolia building blocks.

```
signature Magma = {
  type T;
  function bop(t1: T, t2: T): T;
}

concept Semigroup = {
  use Magma;
  axiom bopIsAssociative(t1: T, t2: T, t3: T) {
    assert bop(t1, bop(t2, t3)) == bop(bop(t1, t2), t3);
  }
}

implementation PyConcreteSemigroup =
  external Python lib.int_impl {
    use Magma[ T => int, bop => add ];
    use Magma[ T => int, bop => mul ];
  }

program ExampleProgram = {
  use PyConcreteSemigroup;
  procedure timesThreeUpdateRef(upd i: int) {
    i = add(add(i, i), i);
  }

  function timesThree(i: int): int {
    var mutable_i = i;
    call timesThreeUpdateRef(mutable_i);
    value mutable_i;
  }
}

satisfaction ExampleProgramHasAddSemigroup =
  ExampleProgram models Semigroup[ T => int, bop => add ];

satisfaction ExampleProgramHasMulSemigroup =
  ExampleProgram models Semigroup[ T => int, bop => mul ];
```

2.3 Specifying Magnolia(c)

In the following, we present each of Magnolia's constructs and provide the complete typing rules for the language as they are concretized (or intended to be concretized) in magnoliac. We discuss some differences compared with the previous main compiler of the language. We don't delve into parsing and related details. Likewise, we do not discuss dynamic semantics of the language.

2.3.1 Grammar of Magnolia

Figure 2.1 shows the syntax of Magnolia.

Magnolia code features two kinds of identifiers:

1. *unqualified names*, which match the regex `[a-zA-Z0-9_]+`. Unqualified names correspond to e.g., names of modules, sources and targets of renamings, variables, types or operations;
2. *fully qualified names*, which match the regex (*unqualified name*`'.'`)`*` *unqualified name*. Fully qualified names are used for paths on the file system, and for package and module dependencies, where unqualified names may be ambiguous.

2.3.2 Static Semantics of Magnolia

package	::=	*package-head top-level…*	package
package-head	::=	`package` *package-name* `imports` *package-name…*;	package header with imports
	|	`package` *package-name* ;	package header
top-level	::=	*module-type id* = *module-expr*	module declaration
	|	`renaming` *id* = *renaming-block*	renaming declaration
	|	`satisfaction` *id* = *satisfaction-expr*	satisfaction declaration
satisfaction-expr	::=	*module-expr* `models` *module-expr*	satisfaction expression
	|	*module-expr* `with` *module-expr* `models` *module-expr*	parameterized satisfaction expression
module-type	::=	`signature` | `concept` | `implementation` | `program`	module kinds
module-expr	::=	*id*	module reference
	|	{ *construct…* }	inline module expression
	|	`external` *host-language package-name module-expr*	external module expression
	|	`signature` (*module-expr*)	*signature* module transformation
	|	*module-expr renaming-block*	*rename* module transformation
	|	`rewrite` *module-expr* `with` *module-expr int*	*rewrite* module transformation
	|	`implement` *module-expr* `in` *module-expr*	*implement* module transformation
renaming-block	::=	[*renaming*, …]	total renaming block
	|	[[*renaming*, …]]	partial renaming block
renaming	::=	*id* => *id*	inline renaming
	|	*id*	renaming reference
host-language	::=	`C++` | `Cuda` | `Python`	host languages

construct	::=	`require` *decl*	required declaration
	|	*decl*	declaration
	|	`require` *module-expr*	module requirement
	|	`use` *module-expr*	module import
prototype	::=	*op-sig*	total prototype
	|	*op-sig* `guard` *expr*	guarded prototype
op-sig	::=	`function` *id* (*arg*, …) : *id*	function signature
	|	`predicate` *id* (*arg*, …)	predicate signature
	|	`procedure` *id* (*mode arg*, …)	procedure signature
	|	`axiom` *id* (*arg*, …)	axiom signature
stmt	::=	`call` *id*(*expr*, …)	procedure call
	|	{ *stmt* ; …}	effectful block
	|	`var` *id* : *id*	variable declaration
	|	`var` *id* : *id* = *expr*	variable definition
	|	`var` *id* = *expr*	variable definition
	|	*id* = *expr*	variable assignment
	|	`if` *expr* `then` *stmt* `else` *stmt*	effectful conditional
	|	`assert` *expr*	assertion
	|	`value` *expr*	value statement
	|	`skip`	no-op

decl	::=	`type` *id* ;	type declaration
	|	*prototype* ;	operation declaration
	|	*prototype* = *expr* ;	operation definition
	|	*prototype* { *stmt* ; …}	operation definition
arg	::=	*id* : *id*	typed argument
mode	::=	`obs`	read mode
	|	`out`	write mode
	|	`upd`	read/write mode
expr	::=	*id*	variable
	|	*id*(*expr*, …)	function call
	|	*id*(*expr*, …) : *id*	typed function call
	|	`if` *expr* `then` *expr* `else` *expr*	conditional
	|	{ *stmt…*; `value` *expr*; }	value block

Figure 2.1: The syntax of Magnolia as supported by magnoliac [35]. Each file is parsed as a *package*. Support for the *rewrite* and *implement* module transformations is experimental and not part of the main development branch of magnoliac at the time of writing. These transformations are discussed in details in Paper 4.

2.3.2.1 Packages

Magnoliac parses each input file as a *package* whose name must correspond to the path to the file—with ‘ . ’ used to denote between directory names instead of classical directory separators (à la Java). A package contains a header, which determines its name and dependencies on other packages, and of a sequence of top-level declarations.

Magnoliac takes in a path towards a target package through its command line interface, and compiles it following these coarse-grained steps:

1. resolve the package's dependencies (see Import system below);
2. verify that the target package's dependencies' top-level declarations are correctly constructed (see Top-level declarations further down);
3. verify that the target package itself is correctly constructed;
4. for each program module in the target package, emit corresponding code in a given host language (today one of C++, Python, or CUDA).

Import system The dependencies of a package are the packages imported in the header along with their transitive dependencies. The compiler parses only package headers during dependency analysis (as opposed to whole packages). This decision is driven by the desire to limit parsing failure cases, in order to report only errors that are meaningful in the context of dependency analysis. This is also more efficient than parsing whole files, but this is not a primary concern.

Dependency analysis can fail in only three ways (excluding out-of-memory errors and the likes): (1) the target package or one of its dependencies does not have a corresponding file, (2) the target package's or one of its dependencies' package header can't be parsed correctly, or (3) the dependency graph contains cycles.

Cyclic package dependencies are theoretically well-behaved thanks to Magnolia's modularity—so long as there are no cyclic dependencies between top-level declarations. However, performing this analysis across package boundaries is more complicated than performing it only within a single package, and it is dubious whether any potential benefit would be worth the added complexity. Dependency analysis between top-level declarations is discussed further in the discussion on top-level declarations.

Magnolia's import system relies on two classes of top-level declarations: *local* declarations, and *imported* declarations. Local declarations are declarations made directly in the target package, while imported declarations are declarations accessible in the target package's context by virtue of being imported from other packages. Importing a package makes that package's local declarations accessible in the local context, but not its imported declarations. Imports can not (yet) be qualified, but references can. Importing only local declarations allows for a simple import system, where name collision resolution is only seldom necessary. When attempting to resolve an (unqualified) top-level reference, the name is first checked against all the local declarations of the package. If no match is found, imported declarations are checked next. In that case, it may be necessary to fully qualify the reference if several imported packages define the same name.

Figure 2.2 presents the typing-rules relevant at the package level, and Figure 2.3 presents some necessary helper functions. We define informally a helper function `content` such that given the name π of a package, `content`(π) returns the content of the file with a matching name relative to the root

Well-formed packages $\boxed{p\ \texttt{ok}}$

$$\textsc{WF-Package}\ \frac{(\textbf{package}\ \pi)\ \texttt{ok} \quad \overline{T\ \texttt{ok in}\ \pi}}{(\textbf{package}\ \pi;\ \overline{T})\ \texttt{ok}} \qquad \textsc{WF-PackageImports}\ \frac{(\textbf{package}\ \pi\ \textbf{imports}\ \overline{P})\ \texttt{ok} \quad \overline{T\ \texttt{ok in}\ \pi}}{(\textbf{package}\ \pi\ \textbf{imports}\ \overline{P};\ \overline{T})\ \texttt{ok}}$$

Well-formed package headers $\boxed{\eta\ \texttt{ok}}$

$$\textsc{WF-PackageHeader}\ \frac{(\texttt{file}\ \pi)\ \texttt{ok}}{(\textbf{package}\ \pi)\ \texttt{ok}} \qquad \textsc{WF-PackageHeaderImports}\ \frac{(\texttt{file}\ \pi)\ \texttt{ok} \quad \texttt{package-header}(P)\ \texttt{ok} \quad \pi \notin \texttt{dependencies}_{\texttt{pkg}}(\pi)}{(\textbf{package}\ \pi\ \textbf{imports}\ \overline{P})\ \texttt{ok}}$$

Figure 2.2: Typing rules for packages.

Definition of $\texttt{dependencies}_{\texttt{pkg}}(\pi)$, and $\texttt{package-header}(\pi)$

$$\frac{\texttt{content}(\pi) = (\textbf{package}\ \pi; \overline{T})}{\texttt{dependencies}_{\texttt{pkg}}(\pi) = \emptyset} \qquad \frac{\texttt{content}(\pi) = (\textbf{package}\ \pi\ \textbf{imports}\ \overline{P}; \overline{T})}{\texttt{dependencies}_{\texttt{pkg}}(\pi) = \bigcup_{p \in \overline{P}} \texttt{dependencies}_{\texttt{pkg}}(p) \cup \overline{P}}$$

$$\frac{\texttt{content}(\pi) = (\textbf{package}\ \pi; \overline{T})}{\texttt{package-header}(\pi) = (\textbf{package}\ \pi)} \qquad \frac{\texttt{content}(\pi) = (\textbf{package}\ \pi\ \textbf{imports}\ \overline{P}; \overline{T})}{\texttt{package-header}(\pi) = (\textbf{package}\ \pi\ \textbf{imports}\ \overline{P})}$$

Figure 2.3: Helper functions at the package level. $\texttt{dependencies}_{\texttt{pkg}}(\pi)$ returns all the (transitive) dependencies of π. $\texttt{package-header}(\pi)$ extracts the header of π. $\texttt{content}(\pi)$ returns the content of the file with name matching the package name π relative to the root directory in which compilation happens.

directory in which compilation happens. If there is no matching file, the function does not succeed. The judgment $(\texttt{file}\ \pi)\ \texttt{ok}$ holds if the name of the file matches the name of the package π that it contains.

2.3.2.2 Top-level declarations

A top-level declaration in Magnolia binds one of three different kinds of expressions to an identifier, i.e. module expressions (the identifier denotes a *module*), renaming expressions (the identifier denotes a *renaming*), and satisfaction expressions (the identifier denotes a *satisfaction relation*). Each kind of top-level declaration comes with its own namespace, allowing e.g. a renaming to have the same name as a module. References to top-level declarations are never ambiguous with regards to which kind of declaration is referred to, and having separate namespaces thus does not add any complexity when resolving references.

Modules Modules constitute Magnolia's primary building blocks. Modules come in four flavours: *signatures*, *concepts*, *programs*, and *implementations*. Each kind of module poses different restrictions regarding what can be expressed.

A *signature* allows the declaration of types and operation prototypes. A *concept* allows the same declarations as a *signature*, but also the specification of semantic requirements on types and operations through *axioms*. A *program* demands the concrete definition of all its types and operations. An *implementation* allows the same declarations as a *signature*, but also the provision of concrete definitions for types and operations. An *implementation* can be seen as a program parameterized by a signature—or equivalently as a function from programs to programs.

Concepts and their axioms are at the heart of Magnolia's design, and allow the expression of powerful specifications. The discussion on module-level declarations gives a detailed explanation of axioms and their role.

The code generated by magnoliac when compiling a package does not reproduce the package's hierarchy of modules. Instead, programs are the only modules for which (self-contained) code is generated. Concepts may however be used to create and apply transformations on programs at compile-time. The discussion on module expressions provides more information about such module transformations.

Renamings A top-level renaming defines a renaming function over module expressions, and consists of a single renaming block. A renaming block is a mapping from names to names, and can be either *partial*, or *total*.

Listing 2.2 shows the difference between total and partial renaming blocks. A total renaming block can only be applied on a module expression successfully if all the "source" names in the block exist in the module expression. A (well-constructed) partial renaming block always applies successfully.

Listing 2.2: Applying a renaming as a total and as a partial renaming block

```
renaming Renaming = [ Source => Target ];

signature Sig1 = { type Type; }[ Renaming ] // Total, fails!
signature Sig2 = { type Type; }[[ Renaming ]] // Partial, succeeds!
```

Total renaming blocks and the syntax for partial renaming blocks constitute one of the new language features introduced in magnoliac. The original Magnolia compiler treated all renaming blocks as partial. This approach makes it easy to define large renaming blocks of common renamings and to apply them whenever relevant—e.g. to map an infix function like `_*_` to a corresponding prefix function like `mul`. The drawback of that approach is that a renaming will silently fail to apply whenever its source name contains a typo—or whenever the name of the declaration intended to be renamed contains a typo. The error then surfaces much later, when one attempts to use the name that one expects the renamed module to define. Because this can happen after many layers of reuse, such a mistake is typically difficult to track down. Adding total renaming blocks to Magnolia while retaining partial renaming blocks allows for better error locations in the ad-hoc renaming case, while still allowing the use of partial renaming blocks whenever convenient.

Renamings are crucial to enabling a powerful reuse mechanism in Magnolia. This is discussed in more details in Paper 1.

Figure 2.4 gives the typing rules for a renaming block, and Figure 2.5 provides definitions for the relevant helper functions.

Well-formed renaming blocks $\boxed{R \texttt{ ok in } \overline{T}}$

WF-TotalRenamingBlock

$$\frac{\texttt{distinct}(\texttt{sources}(\texttt{flatten-renamings}(\overline{r}, \overline{T}))) \qquad \overline{(\exists id, r = id) \Rightarrow (\exists t \in \overline{T}, (t = (\textbf{renaming } id = r')))}}{[\overline{r}] \texttt{ ok in } \overline{T}}$$

WF-PartialRenamingBlock

$$\frac{\texttt{distinct}(\texttt{sources}(\texttt{flatten-renamings}(\overline{r}, \overline{T}))) \qquad \overline{(\exists id, r = id) \Rightarrow (\exists t \in \overline{T}, (t = (\textbf{renaming } id = r')))}}{[[\overline{r}]] \texttt{ ok in } \overline{T}}$$

Figure 2.4: Typing rules for renaming blocks. $\overline{T}$ here is a set of available top-level declarations. Renaming blocks can exist independently of any module expression. As such, the condition of *totality* particular to total renaming blocks needn't be checked here, but rather directly in the typing rules of the relevant module transformation.

Definition of $\texttt{distinct}(\overline{N})$

$$\texttt{distinct}(\overline{N}) = \forall n_1 \in \overline{N}\ \forall n_2 \in \overline{N} \setminus n_1, n_1 \neq n_2$$

Definition of $\texttt{sources}(\overline{R})$

$$\texttt{sources}(\overline{R}) = [\texttt{source}(r) \mid r \in \overline{R}] \textbf{ where } \texttt{source}(\texttt{src => tgt}) = \texttt{src}$$

Definition of $\texttt{flatten-renamings}(\overline{R},)$

$$\texttt{flatten-renamings}(\overline{R}, \overline{T}) = \sum_{r \in \overline{R}} \texttt{flatten-renaming}(r, \overline{T})$$

$$\texttt{flatten-renaming}(\texttt{src => tgt}, \overline{T}) = [\texttt{src => tgt}]$$

$$\frac{\exists \overline{r'}, (\textbf{renaming } \rho = [\overline{r'}] \in \overline{T}) \cup (\textbf{renaming } \rho = [[\overline{r'}]] \in \overline{T})}{\texttt{flatten-renaming}(\rho, \overline{T}) = \overline{r'}}$$

Figure 2.5: Helper functions necessary to check the well-formedness of renaming blocks. $\texttt{distinct}(\overline{N})$ returns true if all the names in a sequence of names are unique, and false otherwise. $\texttt{sources}(\overline{R})$ traverses a sequence of renamings, and for each of them, captures its source name. $\texttt{flatten-renamings}(\overline{R}, \overline{T})$ takes a sequence of renamings, and returns the corresponding sequence of inline renamings after resolving all references to top-level named renamings. The function fails if a named renaming can not be resolved.

Figure 2.6 outlines Magnolia's renaming algorithm all the way down from module-level declarations.

Satisfaction relations Satisfaction relations record facts about programs and specifications. A satisfaction relation describes how and when the requirements of a given specification are satisfied by a given implementation or specification. The body of a satisfaction relation is a *satisfaction expression*.

Satisfaction expressions may refer to modules, module expressions may refer to other modules and renamings, and renaming blocks may refer to other renamings. The correctness of all the top-level declarations in a package is checked by ensuring that the set of renamings and the set of modules each form a directed acyclic graph, and then verifying—in this order—the correctness of all renamings, modules, and satisfactions.

The formal typing rules for Magnolia's top-level declarations are given in Figure 2.7. Figures 2.8 and 2.9 define additional relevant helper functions.

Definition of renaming on module-level declarations $\texttt{rename}(d, \overline{r})$

$\texttt{rename}(\textbf{require}\ d, \overline{r})$	=	$\textbf{require}\ \texttt{rename}(d, \overline{r})$
$\texttt{rename}(\textbf{type}\ \tau, \overline{r})$	=	$\textbf{type}\ \texttt{rename}(\tau, \overline{r})$
$\texttt{rename}(\mathit{prototype} = e, \overline{r})$	=	$\texttt{rename}(\mathit{prototype}, \overline{r}) = \texttt{rename}(e, \overline{r})$
$\texttt{rename}(\mathit{prototype}\ \textbf{guard}\ g, \overline{r})$	=	$\texttt{rename}(\mathit{prototype}, \overline{r})\ \textbf{guard}\ \texttt{rename}(g, \overline{r})$
$\texttt{rename}(\mathit{prototype}\ \textbf{guard}\ g = e, \overline{r})$	=	$\texttt{rename}(\mathit{prototype}, \overline{r})\ \textbf{guard}\ \texttt{rename}(g, \overline{r}) = \texttt{rename}(e, \overline{r})$
$\texttt{rename}(\textbf{function}\ f(\overline{x:\tau}):\tau_r, \overline{r})$	=	$\textbf{function}\ \texttt{rename}(f, \overline{r})(\overline{x : \texttt{rename}(\tau, \overline{r})}) : \texttt{rename}(\tau_r, \overline{r})$
$\texttt{rename}(\textbf{predicate}\ p(\overline{x:\tau}), \overline{r})$	=	$\textbf{predicate}\ \texttt{rename}(p, \overline{r})(\overline{x : \texttt{rename}(\tau, \overline{r})})$
$\texttt{rename}(\textbf{procedure}\ p(\overline{\omega\, x:\tau}), \overline{r})$	=	$\textbf{procedure}\ \texttt{rename}(p, \overline{r})(\overline{\omega\, x : \texttt{rename}(\tau, \overline{r})})$
$\texttt{rename}(\textbf{procedure}\ p(\overline{\omega\, x:\tau})\ b, \overline{r})$	=	$\texttt{rename}(\textbf{procedure}\ p(\overline{\omega\, x:\tau}), \overline{r})\ \texttt{rename}(b, \overline{r})$
$\texttt{rename}(\textbf{procedure}\ p(\overline{\omega\, x:\tau})\ \textbf{guard}\ g\ b, \overline{r})$	=	$\texttt{rename}(\textbf{procedure}\ p(\overline{\omega\, x:\tau})\ \textbf{guard}\ g, \overline{r})\ \texttt{rename}(b, \overline{r})$
$\texttt{rename}(\textbf{axiom}\ a(\overline{x:\tau})\ b, \overline{r})$	=	$\textbf{axiom}\ \texttt{rename}(a, \overline{r})(\overline{x:\texttt{rename}(\tau, \overline{r})})\ \texttt{rename}(b, \overline{r})$

Definition of renaming on statements $\texttt{rename}(s, \overline{r})$

$\texttt{rename}(\{\overline{s}\}, \overline{r})$	=	$\{\overline{\texttt{rename}(s, \overline{r})}\}$
$\texttt{rename}(\textbf{call}\ p(\overline{e}), \overline{r})$	=	$\textbf{call}\ \texttt{rename}(p, \overline{r})(\overline{\texttt{rename}(e, \overline{r})})$
$\texttt{rename}(\textbf{var}\ x:\tau, \overline{r})$	=	$\textbf{var}\ x:\texttt{rename}(\tau, \overline{r})$
$\texttt{rename}(\textbf{var}\ x:\tau = e, \overline{r})$	=	$\textbf{var}\ x:\texttt{rename}(\tau, \overline{r}) = \texttt{rename}(e, \overline{r})$
$\texttt{rename}(\textbf{var}\ x = e, \overline{r})$	=	$\textbf{var}\ x = \texttt{rename}(e, \overline{r})$
$\texttt{rename}(x = e, \overline{r})$	=	$\mathrm{x} = \texttt{rename}(e, \overline{r})$
$\texttt{rename}(\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \overline{r})$	=	$\textbf{if}\ \texttt{rename}(e, \overline{r})\ \textbf{then}\ \texttt{rename}(s_1, \overline{r})\ \textbf{else}\ \texttt{rename}(s_2, \overline{r})$
$\texttt{rename}(\textbf{assert}\ e, \overline{r})$	=	$\textbf{assert}\ \texttt{rename}(e, \overline{r})$
$\texttt{rename}(\textbf{value}\ e, \overline{r})$	=	$\textbf{value}\ \texttt{rename}(e, \overline{r})$
$\texttt{rename}(\textbf{skip}, \overline{r})$	=	$\textbf{skip}$

Definition of renaming on expressions $\texttt{rename}(e, \overline{r})$

$\texttt{rename}(\text{variable}, \overline{r})$	=	variable
$\texttt{rename}(f(\overline{e}), \overline{r})$	=	$\texttt{rename}(f, \overline{r})(\overline{\texttt{rename}(e, \overline{r})})$
$\texttt{rename}(f(\overline{e}):\tau_r, \overline{r})$	=	$\texttt{rename}(f, \overline{r})(\overline{\texttt{rename}(e, \overline{r})}):\texttt{rename}(\tau_r, \overline{r})$
$\texttt{rename}(\textbf{if}\ c\ \textbf{then}\ e_1\ \textbf{else}\ e_2, \overline{r})$	=	$\textbf{if}\ \texttt{rename}(c, \overline{r})\ \textbf{then}\ \texttt{rename}(e_1, \overline{r})\ \textbf{else}\ \texttt{rename}(e_2, \overline{r})$
$\texttt{rename}(\{\overline{s};\ \textbf{value}\ e;\}, \overline{r})$	=	$\{\overline{\texttt{rename}(s, \overline{r})};\ \textbf{value}\ \texttt{rename}(e, \overline{r});\}$

Definition of renaming on names $\texttt{rename}(\nu, \overline{r})$

$$\frac{\exists!\nu_2,\ (\nu_1 \texttt{=>}\ \nu_2) \in \overline{r}}{\texttt{rename}(\nu_1, \overline{r}) = \nu_2} \qquad \frac{\nexists\nu_2,\ (\nu_1 \texttt{=>}\ \nu_2) \in \overline{r}}{\texttt{rename}(\nu_1, \overline{r}) = \nu_1}$$

Figure 2.6: Implementation of Magnolia's renaming algorithm. Renaming applies on module-level declarations and references made to them in statements and expressions. The renaming algorithm always processes an entire renaming block at a time, and assumes that it has been flattened—i.e. it contains only inline renamings. See the definition of $\texttt{flatten-renamings}(\overline{r}, \overline{T})$ in Figure 2.4 for more details. The renaming block is not allowed to map the same source name to several target names.

Well-formed top-level declarations $\boxed{c\ \texttt{ok in}\ \pi}$

$$\frac{c = (\textbf{renaming}\ \rho = r) \quad \rho \notin \texttt{names}_{\text{renaming}}(\texttt{local}(\pi) \setminus c) \quad r\ \texttt{ok in}\ \Delta_{\texttt{pkg}}(\pi) \setminus c}{c\ \texttt{ok in}\ \pi}\ \text{WF-NamedRenaming}$$

$$\frac{\begin{array}{c}c = (\textbf{signature}\ \sigma = m) \quad \sigma \notin \texttt{names}_{\text{module}}(\texttt{local}(\pi) \setminus c) \quad m\ \texttt{ok in}\ \Delta_{\texttt{pkg}}(\pi) \setminus c \\ \Delta_{\texttt{mod}}(m, \Delta_{\texttt{pkg}}(\pi) \setminus c) \subset \Delta_{\texttt{abs}}(m, \Delta_{\texttt{pkg}}(\pi) \setminus c)\end{array}}{c\ \texttt{ok in}\ \pi}\ \text{WF-SignatureModule}$$

$$\frac{\begin{array}{c}c = (\textbf{concept}\ \sigma = m) \quad \sigma \notin \texttt{names}_{\text{module}}(\texttt{local}(\pi) \setminus c) \quad m\ \texttt{ok in}\ \Delta_{\texttt{pkg}}(\pi) \setminus c \\ \forall d \in \Delta_{\texttt{mod}}(m, \Delta_{\texttt{pkg}}(\pi) \setminus c), d \in \Delta_{\texttt{abs}}(m, \Delta_{\texttt{pkg}}(\pi) \setminus c) \cup \texttt{is-axiom}(d)\end{array}}{c\ \texttt{ok in}\ \pi}\ \text{WF-ConceptModule}$$

$$\frac{\begin{array}{c}c = (\textbf{implementation}\ \sigma = m) \quad \sigma \notin \texttt{names}_{\text{module}}(\texttt{local}(\pi) \setminus c) \quad m\ \texttt{ok in}\ \Delta_{\texttt{pkg}}(\pi) \setminus c \\ \forall d \in \Delta_{\texttt{mod}}(m, \Delta_{\texttt{pkg}}(\pi) \setminus c), \neg\texttt{is-axiom}(d)\end{array}}{c\ \texttt{ok in}\ \pi}\ \text{WF-ImplementationModule}$$

$$\frac{\begin{array}{c}c = (\textbf{program}\ \sigma = m) \quad \sigma \notin \texttt{names}_{\text{module}}(\texttt{local}(\pi) \setminus c) \quad m\ \texttt{ok in}\ \Delta_{\texttt{pkg}}(\pi) \setminus c \\ (\textbf{implementation}\ \sigma = m)\ \texttt{ok in}\ \pi \qquad \forall d \in \Delta_{\texttt{abs}}(m, \Delta_{\texttt{pkg}}(\pi) \setminus c), d \in \{\texttt{sig}(d') \mid d' \in \Delta_{\texttt{con}}(m, \Delta_{\texttt{pkg}}(\pi) \setminus c)\}\end{array}}{c\ \texttt{ok in}\ \pi}\ \text{WF-ProgramModule}$$

$$\frac{c = (\textbf{satisfaction}\ \sigma = s) \quad \sigma \notin \texttt{names}_{\text{satisfaction}}(\texttt{local}(\pi) \setminus c) \quad s\ \texttt{ok in}\ \pi}{c\ \texttt{ok in}\ \pi}\ \text{WF-Satisfaction}$$

Figure 2.7: Typing rules for top-level declarations. A top-level declaration can either be a named renaming, a module, or a satisfaction relation. A module is either a signature, a concept, an implementation, or a program.

Definition of $\Delta_{\texttt{pkg}}(\pi)$

$$\Delta_{\texttt{pkg}}(\pi) = \texttt{local}(\pi) + [\texttt{local}(p) \mid p \in \texttt{dependencies}_{\texttt{pkg}}(\pi)]$$

Definition of $\texttt{local}(\pi)$

$$\frac{\texttt{content}(\pi) = (\textbf{package}\ \pi; \overline{T})}{\texttt{local}(\pi) = \overline{T}} \qquad \frac{\texttt{content}(\pi) = (\textbf{package}\ \pi\ \textbf{imports}\ \overline{P}; \overline{T})}{\texttt{local}(\pi) = \overline{T}}$$

Definition of $\Delta_{\texttt{mod}}(m, \overline{T})$

$$\frac{\textbf{kw}\ \sigma = m \in \overline{T} \quad \textbf{kw} \in \{\textbf{signature}, \textbf{concept}, \textbf{implementation}, \textbf{program}\}}{\Delta_{\texttt{mod}}(\sigma, \overline{T}) = \Delta_{\texttt{mod}}(m, \overline{T})}$$

$$\begin{aligned}
\Delta_{\texttt{mod}}(\{\overline{U}; \overline{D}\}, \overline{T}) &= \textstyle\sum_{u \in \overline{U}} \Delta_{\texttt{dep}}(u, \overline{T}) + \overline{D} \\
\Delta_{\texttt{mod}}(m[\overline{r}], \overline{T}) &= [\texttt{rename}(d, \texttt{flatten-renamings}(\overline{r}, \overline{T})) \mid d \in \Delta_{\texttt{mod}}(m, \overline{T})] \\
\Delta_{\texttt{mod}}(m[[\overline{r}]], \overline{T}) &= [\texttt{rename}(d, \texttt{flatten-renamings}(\overline{r}, \overline{T})) \mid d \in \Delta_{\texttt{mod}}(m, \overline{T})] \\
\Delta_{\texttt{mod}}(\textbf{signature}(m), \overline{T}) &= [\texttt{sig}(d) \mid d \in \Delta_{\texttt{mod}}(m, \overline{T})] \\
\Delta_{\texttt{mod}}(\textbf{external}\ h\, p\, m, \overline{T}) &= \Delta_{\texttt{mod}}(m, \overline{T})
\end{aligned}$$

Definition of $\Delta_{\texttt{dep}}(u, \overline{T})$

$$\begin{aligned}
\Delta_{\texttt{dep}}(\textbf{use}\ m, \overline{T}) &= \Delta_{\texttt{mod}}(m, \overline{T}) \\
\Delta_{\texttt{dep}}(\textbf{require}\ m, \overline{T}) &= [\textbf{require}\ \texttt{sig}(d) \mid d \in \Delta_{\texttt{mod}}(m, \overline{T}), (\nexists(a, \overline{x}, \overline{\tau}, b), d = \textbf{axiom}\ a(\overline{x : \tau})\ b)]
\end{aligned}$$

Definition of $\texttt{sig}(d)$

$$\begin{aligned}
\texttt{sig}(\textbf{function}\ f(\overline{x:\tau}) : \tau_r) &= \textbf{function}\ f(\overline{x:\tau}) : \tau_r \\
\texttt{sig}(\textbf{function}\ f(\overline{x:\tau}) : \tau_r = e) &= \textbf{function}\ f(\overline{x:\tau}) : \tau_r \\
\texttt{sig}(\textbf{procedure}\ p(\overline{\omega\, x:\tau})) &= \textbf{procedure}\ p(\overline{\omega\, x:\tau}) \\
\texttt{sig}(\textbf{procedure}\ p(\overline{\omega\, x:\tau})\ b) &= \textbf{procedure}\ p(\overline{\omega\, x:\tau}) \\
\texttt{sig}(o\ \textbf{guard}\ e) &= o\ \textbf{guard}\ e \\
\texttt{sig}(o\ \textbf{guard}\ e\, b) &= o\ \textbf{guard}\ e \\
\texttt{sig}(\textbf{require}\ d) &= \textbf{require}\ d \\
\texttt{sig}(\textbf{type}\ \tau) &= \textbf{type}\ \tau \\
\texttt{sig}(\textbf{predicate}\ p(\overline{x:\tau})) &= \textbf{predicate}\ p(\overline{x:\tau}) \\
\texttt{sig}(\textbf{predicate}\ p(\overline{x:\tau}) = e) &= \textbf{predicate}\ p(\overline{x:\tau}) \\
\texttt{sig}(o\ \textbf{guard}\ e = e') &= o\ \textbf{guard}\ e
\end{aligned}$$

Figure 2.8: $\Delta_{\texttt{pkg}}(\pi)$ returns a sequence of all the top-level declarations accessible within package π. $\texttt{local}(\pi)$ returns the top-level declarations specifically introduced in π. $\Delta_{\texttt{mod}}(m, \overline{T})$ flattens a module expression m to its corresponding set of declarations. The required $\texttt{rename}(d, \overline{r})$ helper function corresponds to the renaming algorithm outlined previously in Figure 2.6. Similarly, $\Delta_{\texttt{dep}}(u, \overline{T})$ flattens a dependency expression into its corresponding set of declarations. $\texttt{sig}(d)$ maps a module-level declaration to its signature.

Definition of $\texttt{names}_{\text{renaming}}(\overline{T})$, $\texttt{names}_{\text{module}}(\overline{T})$, and $\texttt{names}_{\text{satisfaction}}(\overline{T})$

$$
\begin{aligned}
\texttt{names}_{\text{renaming}}(\overline{T}) &= \{\rho \mid \exists r, (\textbf{renaming}\ \rho = r) \in \overline{T}\} \\
\texttt{names}_{\text{module}}(\overline{T}) &= \{\sigma \mid \exists m \exists \textbf{kw} \in \{\textbf{signature}, \textbf{concept}, \textbf{implementation}, \textbf{program}\}, (\textbf{kw}\ \sigma = m) \in \overline{T}\} \\
\texttt{names}_{\text{satisfaction}}(\overline{T}) &= \{\sigma \mid \exists (m_1, m_2), (\textbf{satisfaction}\ m_1\ \textbf{models}\ m_2) \in \overline{T}\} \cup \\
&\quad \{\sigma \mid \exists (m_1, m_2, m_3), (\textbf{satisfaction}\ m_1\ \textbf{with}\ m_3\ \textbf{models}\ m_2) \in \overline{T}\}
\end{aligned}
$$

Definition of $\Delta_{\texttt{abs}}(m, \overline{T})$ and $\Delta_{\texttt{con}}(m, \overline{T})$

$$
\frac{\textbf{kw}\ \sigma = m \in \overline{T} \quad \textbf{kw} \in \{\textbf{signature}, \textbf{concept}, \textbf{implementation}, \textbf{program}\}}{\Delta_{\texttt{abs}}(\sigma, \overline{T}) = \Delta_{\texttt{abs}}(m, \overline{T})}
$$

$$
\begin{aligned}
\Delta_{\texttt{abs}}(\{\overline{U}; \overline{D}\}, \overline{T}) &= \bigcup_{u \in \overline{U}} \Delta'_{\texttt{abs}}(u) \cup \{d \mid d \in \overline{D}, \texttt{is-abstract}(d)\} \\
&\quad \textbf{where}\ \ \Delta'_{\texttt{abs}}(\textbf{use}\ m) = \Delta_{\texttt{abs}}(m, \overline{T}) \\
&\qquad\quad \Delta'_{\texttt{abs}}(\textbf{require}\ m) = \{\textbf{require}\ \texttt{sig}(d) \mid d \in \Delta_{\texttt{mod}}(m, \overline{T})\} \\
\Delta_{\texttt{abs}}(\textbf{signature}(m), \overline{T}) &= \Delta_{\texttt{mod}}(\textbf{signature}(m), \overline{T}) \\
\Delta_{\texttt{abs}}(\textbf{external}\ h\ p\ m, \overline{T}) &= \{d \mid d \in \Delta_{\texttt{mod}}(m, \overline{T}), \texttt{is-required}(d)\} \\
\Delta_{\texttt{abs}}(m[\overline{r}], \overline{T}) &= \Delta_{\texttt{abs}}(\{\Delta_{\texttt{mod}}(m[\overline{r}], \overline{T})\}, \overline{T}) \\
\Delta_{\texttt{con}}(m, \overline{T}) &= \{d \mid d \in \Delta_{\texttt{mod}}(m, \overline{T}), d \notin \Delta_{\texttt{abs}}(m, \overline{T})\}
\end{aligned}
$$

Definition of $\texttt{is-required}(d)$, $\texttt{is-abstract}(d)$, and $\texttt{is-axiom}(d)$

$$
\begin{aligned}
&\texttt{is-required}(\textbf{require}\ d) = \texttt{is-abstract}(d) \qquad \texttt{is-required}(d \mid \nexists d', d = \textbf{require}\ d') = \bot \\
&\\
&\texttt{is-abstract}(\textbf{require}\ d) = \texttt{is-abstract}(d) \qquad \texttt{is-abstract}(\textbf{function}\ f(\overline{x : \tau}) : \tau_r) = \top \\
&\texttt{is-abstract}(\textbf{type}\ \tau) = \top \qquad \texttt{is-abstract}(\textbf{function}\ f(\overline{x : \tau}) : \tau_r = e) = \bot \\
&\texttt{is-abstract}(o\ \textbf{guard}\ e) = \top \qquad \texttt{is-abstract}(\textbf{procedure}\ p(\overline{\omega\, x : \tau})) = \top \\
&\texttt{is-abstract}(o\ \textbf{guard}\ e = e') = \bot \qquad \texttt{is-abstract}(\textbf{procedure}\ p(\overline{\omega\, x : \tau})\ b) = \bot \\
&\texttt{is-abstract}(o\ \textbf{guard}\ e\ b) = \bot \qquad \texttt{is-abstract}(\textbf{predicate}\ p(\overline{x : \tau})) = \top \\
&\qquad\qquad\qquad\qquad\quad \texttt{is-abstract}(\textbf{predicate}\ p(\overline{x : \tau}) = e) = \bot \\
&\qquad\qquad\qquad\qquad\quad \texttt{is-abstract}(\textbf{axiom}\ a(\overline{x : \tau})\ b) = \bot \\
&\\
&\texttt{is-axiom}(\textbf{axiom}\ a(\overline{x : \tau})\ b) = \top \qquad \texttt{is-axiom}(d \mid \nexists (a, \overline{x}, \overline{\tau}, b), d = \textbf{axiom}\ a(\overline{x : \tau})\ b) = \bot
\end{aligned}
$$

Figure 2.9: Second set of helper functions for checking top-level constructs. $\texttt{names}_{\texttt{renaming}}(\overline{T})$, $\texttt{names}_{\texttt{module}}(\overline{T})$, and $\texttt{names}_{\texttt{satisfaction}}(\overline{T})$ respectively return the set of renaming, module, and satisfaction names in a given sequence of declarations $\overline{T}$. $\Delta_{\texttt{abs}}(m, \overline{T})$ and $\Delta_{\texttt{con}}(m, \overline{T})$ respectively retrieve the set of abstract and concrete declarations contained in a module expression m with outer context $\overline{T}$. `is-{required,abstract,axiom}` are utilities that determine where a module-level declaration d is respectively required, abstract, or an axiom.

2.3.2.3 Satisfaction expressions

Satisfaction expressions establish a modeling relation between two module expressions, the *source* (on the right-hand side of the relation) and the *target* (on the left-hand side of the relation). The source module expression must be a valid concept expression (possibly with zero axioms). The target module expression's signature must subsume the source's, and may be a valid concept expression or a valid implementation expression.

A satisfaction expression's target may be parameterized by a third implementation expression. In that case, it is the union of the signatures of the target and of the parameter that must subsume the signature of the source. Figure 2.10 gives the formal typing rules for satisfaction expressions. Figure 2.11 defines the $\Delta_\Sigma(\overline{D})$ helper.

Well-formed satisfaction expressions $\boxed{s \texttt{ ok in } \pi}$

WF-SatisfactionExpr

$$\frac{\begin{array}{c} m_{\texttt{source}} \texttt{ ok in } \Delta_{\texttt{pkg}}(\pi) \qquad m_{\texttt{target}} \texttt{ ok in } \Delta_{\texttt{pkg}}(\pi) \qquad \exists\sigma, (\textbf{concept } \sigma = m_{\texttt{source}}) \texttt{ ok in } \pi \\ \exists\sigma, (\textbf{concept } \sigma = m_{\texttt{target}}) \texttt{ ok in } \pi \cup (\textbf{implementation } \sigma = m_{\texttt{target}}) \texttt{ ok in } \pi \\ \Delta_\Sigma(\Delta_{\texttt{mod}}(\textbf{signature}(m_{\texttt{source}}), \Delta_{\texttt{pkg}}(\pi))) \subset \Delta_\Sigma(\Delta_{\texttt{mod}}(\textbf{signature}(m_{\texttt{target}}), \Delta_{\texttt{pkg}}(\pi))) \end{array}}{(m_{\texttt{target}} \textbf{ models } m_{\texttt{source}}) \texttt{ ok in } \pi}$$

WF-ParameterizedSatisfactionExpr

$$\frac{\begin{array}{c} m_{\texttt{source}} \texttt{ ok in } \Delta_{\texttt{pkg}}(\pi) \qquad m_{\texttt{target}} \texttt{ ok in } \Delta_{\texttt{pkg}}(\pi) \qquad m_{\texttt{parameter}} \texttt{ ok in } \Delta_{\texttt{pkg}}(\pi) \\ \exists\sigma, (\textbf{concept } \sigma = m_{\texttt{source}}) \texttt{ ok in } \pi \qquad \exists\sigma, (\textbf{implementation } \sigma = m_{\texttt{parameter}}) \texttt{ ok in } \pi \\ \exists\sigma, (\textbf{concept } \sigma = m_{\texttt{target}}) \texttt{ ok in } \pi \cup (\textbf{implementation } \sigma = m_{\texttt{target}}) \texttt{ ok in } \pi \\ \Delta_\Sigma(\Delta_{\texttt{mod}}(\textbf{signature}(m_{\texttt{source}}), \Delta_{\texttt{pkg}}(\pi))) \subset \Delta_\Sigma(\Delta_{\texttt{mod}}(\textbf{signature}(\{\textbf{use } m_{\texttt{target}}; \textbf{ use } m_{\texttt{parameter}}\}), \Delta_{\texttt{pkg}}(\pi))) \end{array}}{(m_{\texttt{target}} \textbf{ with } m_{\texttt{parameter}} \textbf{ models } m_{\texttt{source}}) \texttt{ ok in } \pi}$$

Figure 2.10: Typing rules for satisfaction expressions.

Definition of helper environment $\Delta_\Sigma(\overline{D})$

$\Delta_\Sigma(\overline{D})$	$= [\Delta'_\Sigma(d) \mid d \in \overline{D}]$	$\Delta'_\Sigma(\textit{prototype} = e)$	$= \Delta'_\Sigma(\textit{prototype})$
$\Delta'_\Sigma(\textbf{require } d)$	$= \Delta'_\Sigma(d)$	$\Delta'_\Sigma(\textit{prototype } \textbf{guard } g)$	$= \Delta'_\Sigma(\textit{prototype})$
$\Delta'_\Sigma(\textbf{type } \tau)$	$= \tau$	$\Delta'_\Sigma(\textit{prototype } \textbf{guard } g = e)$	$= \Delta'_\Sigma(\textit{prototype})$
$\Delta'_\Sigma(\textbf{function } f(\overline{x{:}\tau}) : \tau_r)$	$= f : \overline{\tau, \textbf{obs}} \to \tau_r$	$\Delta'_\Sigma(\textbf{procedure } p(\overline{\omega\, x{:}\tau})\ b)$	$= p : \overline{\tau, \omega} \to \texttt{Unit}$
$\Delta'_\Sigma(\textbf{predicate } p(\overline{x{:}\tau}))$	$= p : \overline{\tau, \textbf{obs}} \to \texttt{Pred}$	$\Delta'_\Sigma(\textbf{procedure } p(\overline{\omega\, x{:}\tau}) \textbf{ guard } g\ b)$	$= p : \overline{\tau, \omega} \to \texttt{Unit}$
$\Delta'_\Sigma(\textbf{procedure } p(\overline{\omega\, x{:}\tau}))$	$= p : \overline{\tau, \omega} \to \texttt{Unit}$	$\Delta'_\Sigma(\textbf{axiom } a(\overline{x{:}\tau})\ b)$	$= a : \overline{\tau, \textbf{obs}} \to \texttt{Unit}$

Figure 2.11: $\Delta_\Sigma(\overline{D})$ converts a sequence of declarations into a unified environment containing the corresponding types and prototypes. Prototypes retain variable mode information but omit guards. Procedure and axiom prototypes are transformed into functional prototypes returning the `Unit` type.

2.3.2.4 Module expressions

Modules and satisfactions expressions are constructed using *module expressions*. The fundamental module expression is the (anonymous) *module definition*, which consists of *uses* (i.e. "imports" of existing modules) and *declarations of types and operations*. Magnolia does not allow recursion, and the graph of dependencies of implemented operations on each other must therefore be a directed acyclic graph (DAG). Module declarations at the top-level allow for *module references*, which retrieve the (flattened) module expressions corresponding to a module name. Magnoliac also provides a number of built-in endofunctions over module expressions, namely:

1. **signature**(m), which maps a module expression m to its signature;
2. **external** $h\ p\ m$, which assigns an external implementation provided in the file at path p in the host programming language h for the (non-required) abstract declarations of module expressions m;
3. $m[\ldots]$ and $m[[\ldots]]$, which allow for applying total and partial renamings over m's declarations;
4. **rewrite** m_1 **with** $m_2\ n$, which extracts an equational rewriting system from axioms in m_2 and applies it n times on the body of each operation in m_1. The rewriting rules are not applied in a deterministic order, making this transformation more adapted for uses with confluent and terminating rewriting systems.
5. **implement** f **in** m_1 **using** m_2, which allows for deriving in m_1 a default implementation of an operation f from a particular axiomatic description of f in m_2.

These functions over modules are called *module transformations* in the rest of the text.

A flattened module expression is simply a set of module-level declarations, i.e. a module expression in which all module transformations have been fully applied, and all module references have been resolved. All module expressions are flattened during type checking. The signatures of two module expressions are deemed equal if their flattened form corresponds to the exact same set of declarations independently of partiality constraints, i.e., ignoring guards. We intentionally ignore guards when adding declarations to our typing environments. Guards are unnecessary for developing typing rules and complicate the presentation. In practice, if a single module expression contains two declarations for a prototype p with distinct guards g_1 and g_2, the declarations are merged to produce a prototype p with guard $g_1 \wedge g_2$.

Figure 2.12 describes well-formed module expressions in Magnolia. The *rewrite* and *implement* module transformations are the latest experimental additions to magnoliac. They are presented in the text above for the sake of completeness, but are not described in the typing rules. Paper 4 introduces these two transformations and discusses them in more detail.

Well-formed module expressions $\boxed{m \texttt{ ok in } \overline{T}}$

$$\frac{\sigma \in \texttt{names}_{\texttt{module}}(\overline{T})}{\sigma \texttt{ ok in } \overline{T}}\ \text{WF-ModuleRef} \qquad \frac{m \texttt{ ok in } \overline{T} \qquad \Delta_{\texttt{mod}}(m, \overline{T}) \subset \Delta_{\texttt{abs}}(m, \overline{T})}{\textbf{external } h\, p\, m \texttt{ ok in } \overline{T}}\ \text{WF-ModuleExprExternal} \qquad \frac{m \texttt{ ok in } \overline{T}}{\textbf{signature}(m) \texttt{ ok in } \overline{T}}\ \text{WF-ModuleExprSig}$$

$$\frac{\begin{array}{c} m \texttt{ ok in } \overline{T} \qquad [\overline{r}] \texttt{ ok in } \overline{T} \qquad \forall d \in \Delta_{\texttt{mod}}(m[\overline{r}], \overline{T}), d \texttt{ ok in } \Delta_{\texttt{mod}}(m[\overline{r}], \overline{T}) \qquad \Delta_{\texttt{mod}}(m[\overline{r}], \overline{T}) \texttt{ is a DAG} \\ \forall s \in \texttt{sources}(\texttt{flatten-renamings}(\overline{r}, \overline{T})), \\ (s \in \Delta_\Sigma(\Delta_{\texttt{mod}}(m, \overline{T}))) \cup (\exists(\overline{\omega}, \overline{\tau}, \tau_r), s{:}\overline{\tau, \omega} \to \tau_r \in \Delta_\Sigma(\Delta_{\texttt{mod}}(m, \overline{T}))) \end{array}}{m[\overline{r}] \texttt{ ok in } \overline{T}}\ \text{WF-ModuleExprRenameTotal}$$

$$\frac{m \texttt{ ok in } \overline{T} \qquad [[\overline{r}]] \texttt{ ok in } \overline{T} \qquad \forall d \in \Delta_{\texttt{mod}}(m[[\overline{r}]], \overline{T}), d \texttt{ ok in } \Delta_{\texttt{mod}}(m[[\overline{r}]], \overline{T}) \qquad \Delta_{\texttt{mod}}(m[[\overline{r}]], \overline{T}) \texttt{ is a DAG}}{m[[\overline{r}]] \texttt{ ok in } \overline{T}}\ \text{WF-ModuleExprRenamePartial}$$

$$\frac{U \texttt{ ok in } \overline{T} \qquad \forall d \in \Delta_{\texttt{mod}}(\{\overline{U}; \overline{D}\}, \overline{T}), d \texttt{ ok in } \Delta_{\texttt{mod}}(\{\overline{U}; \overline{D}\}, \overline{T}) \qquad \Delta_{\texttt{mod}}(\{\overline{U}; \overline{D}\}, \overline{T}) \texttt{ is a DAG}}{\{\overline{U}; \overline{D}\} \texttt{ ok in } \overline{T}}\ \text{WF-ModuleExprDef}$$

Figure 2.12: Typing rules for module expressions.

2.3.2.5 Module-level dependency expressions

The **use** and **require** dependency expressions allow for inlining a flattened module expressions in the context of another module expression. Their main use is for composing several top-level modules together within the same module expression. The **use** dependency expression inlines a flattened module expression verbatim into its parent module expression, while the **require** dependency expression transforms the flattened module expression into its corresponding signature, and marks all the resulting declarations as *required*. This means that axioms are removed from the required module expression, and that bodies are stripped from declarations that have one (e.g. function or procedure definitions).

Well-formed dependency expressions $\boxed{u \texttt{ ok in } \overline{T}}$

$$\frac{m \texttt{ ok in } \overline{T}}{\textbf{use } m \texttt{ ok in } \overline{T}}\ \text{WF-UseModule} \qquad \frac{m \texttt{ ok in } \overline{T}}{\textbf{require } m \texttt{ ok in } \overline{T}}\ \text{WF-RequireModule}$$

Figure 2.13: Typing rules for module-level dependency expressions.

2.3.2.6 Module-level declarations

Module expressions encapsulate the declarations of Magnolia types and operations. The four different kinds of operations are:

1. pure *functions*;
2. *procedures*, which wrap a sequence of statements and give access to a restricted form of side-effects;
3. *predicates*, which are specific functions which return booleans;
4. *axioms*, which allow for specifying semantic properties of types and operations through assertions.

Declaration modifiers Declarations are treated as either *required*, *abstract*, or *concrete*. A concrete declaration provides an implementation for a type or operation, while an abstract declaration only declares the existence of a type or operation. Required declarations are abstract declarations declared using the **require** modifier. The only difference between required and abstract declarations is how they behave under certain module transformations. For example, the *external* module transformation transforms a module expression's abstract but non-required declarations into concrete (external) definitions, but does not modify required declarations.

Type system Magnolia has evolved to rely only on a rudimentary type system, straying away from the original description of the language in Bagge's dissertation [14]. As shown in Figure 2.1, types can only be declared in Magnolia as opaque identifiers whose implementation must be provided externally. Apart from user-defined types, magnoliac makes use of two built-in types—that can not be accessed explicitly by the user:

1. `Unit`, the return type of procedures, statements, and thus of *"effectful"* computations. When transpiling to C++, `Unit` is mapped to *void*;
2. `Pred`, the return type of predicates. The usual boolean functions are built-in, and can be used to combine expressions of type `Pred`. When transpiling to C++, `Pred` is mapped to *bool*.

There is no way to state relationships between types apart from operations, and type checking does not have to take into account either *polymorphism*, *subtyping* or *coercions*.

Mode system The readability and writability of variables in statements is controlled by the following three modes: **obs**, **upd**, and **out**. An **upd** variable can be both read from and written to; an **obs** variable can be read-from but not written to; an **out** variable can be written to but not (immediately) read from.

The input parameters of *predicates*, *functions* and *axioms* are implicitly assigned the **obs** mode—guaranteeing their immutability—while the input parameters to *procedures* must have their mode explicitly provided. Modes enable procedures to have controlled side-effects. If a parameter to a procedure has mode **out**, then it is guaranteed that all paths through the procedure will set its value. Once an **out** variable has been assigned a value for the first time, it becomes **upd** and can thus be read from.

Variables declared in statement blocks are **out** if they are not assigned a value, or **upd** otherwise.

Partiality As briefly explained previously, operations in Magnolia can be made partial using *guards*. Unguarded operations are implicitly treated as being guarded by a call to the `TRUE()` built-in predicate.

The typing rules for module-level declarations are presented in Figure 2.14. The module-level built-ins are presented in Figure 2.15 and the `const-env` and `update-env` helpers are given in Figure 2.16.

Well-formed module-level declarations and definitions $\boxed{d \texttt{ ok in } \overline{D}}$

$$\frac{\texttt{is-abstract}(d) \qquad d \texttt{ ok in } \overline{D}}{\textbf{require } d \texttt{ ok in } \overline{D}} \text{ WF-RequiredDecl}$$

$$\frac{}{\textbf{type } \tau \texttt{ ok in } \overline{D}} \text{ WF-TypeDecl}$$

$$\frac{\textit{prototype } \textbf{guard } \texttt{TRUE()} \texttt{ ok in } \overline{D}}{\textit{prototype } \texttt{ ok in } \overline{D}} \text{ WF-PrototypeDecl}$$

$$\frac{\overline{\Delta_\Sigma(\overline{D}) \vdash \tau} \qquad \Delta_\Sigma(\overline{D}) \vdash \tau_r \qquad \texttt{distinct}(\overline{x}) \qquad \Delta_\Sigma(\overline{D}); \overline{x : (\tau, \textbf{obs})} \vdash g : (\texttt{Pred}, \textbf{obs})}{\textbf{function } f(\overline{x : \tau}) : \tau_r \textbf{ guard } g \texttt{ ok in } \overline{D}} \text{ WF-GuardedFunctionDecl}$$

$$\frac{\overline{\Delta_\Sigma(\overline{D}) \vdash \tau} \qquad \texttt{distinct}(\overline{x}) \qquad \Delta_\Sigma(\overline{D}); \overline{x : (\tau, \textbf{obs})} \vdash g : (\texttt{Pred}, \textbf{obs})}{\textbf{predicate } p(\overline{x : \tau}) \textbf{ guard } g \texttt{ ok in } \overline{D}} \text{ WF-GuardedPredicateDecl}$$

$$\frac{\overline{\Delta_\Sigma(\overline{D}) \vdash \tau} \qquad \texttt{distinct}(\overline{x}) \qquad \Delta_\Sigma(\overline{D}); \overline{\texttt{const-env}(x : (\tau, \omega))} \vdash g : (\texttt{Pred}, \textbf{obs})}{\textbf{procedure } p(\overline{\omega\, x : \tau}) \textbf{ guard } g \texttt{ ok in } \overline{D}} \text{ WF-GuardedProcedureDecl}$$

$$\frac{\textbf{function } f(\overline{x : \tau}) : \tau_r \texttt{ ok in } \overline{D} \qquad \Delta_\Sigma(\overline{D}); \overline{x : (\tau, \textbf{obs})} \vdash e : (\tau_r, \textbf{obs})}{\textbf{function } f(\overline{x : \tau}) : \tau_r = e \texttt{ ok in } \overline{D}} \text{ WF-FunctionDef}$$

$$\frac{\textbf{predicate } p(\overline{x : \tau}) \texttt{ ok in } \overline{D} \qquad \Delta_\Sigma(\overline{D}); \overline{x : (\tau, \textbf{obs})} \vdash e : (\texttt{Pred}, \textbf{obs})}{\textbf{predicate } p(\overline{x : \tau}) = e \texttt{ ok in } \overline{D}} \text{ WF-PredicateDef}$$

$$\frac{\textbf{procedure } p(\overline{\omega\, x : \tau}) \texttt{ ok in } \overline{D} \qquad \Delta_\Sigma(\overline{D}); \overline{x : (\tau, \omega)} \vdash b \texttt{ ok} \qquad (\omega = \textbf{out}) \Rightarrow (v, \tau, \textbf{upd}) \in \texttt{update-env}(\Delta, \emptyset, \overline{x : (\tau, \omega)}, b)}{\textbf{procedure } p(\overline{\omega\, x : \tau})\ b \texttt{ ok in } \overline{D}} \text{ WF-ProcedureDef}$$

$$\frac{\textit{prototype} = e \texttt{ ok in } \overline{D} \qquad \textit{prototype } \textbf{guard } g \texttt{ ok in } \overline{D}}{\textit{prototype } \textbf{guard } g = e \texttt{ ok in } \overline{D}} \text{ WF-GuardedFunctionalDef}$$

$$\frac{\textbf{procedure } p(\overline{\omega\, x : \tau})\ b \texttt{ ok in } \overline{D} \qquad \textbf{procedure } p(\overline{\omega\, x : \tau}) \textbf{ guard } g \texttt{ ok in } \overline{D}}{\textbf{procedure } p(\overline{\omega\, x : \tau}) \textbf{ guard } g\ b \texttt{ ok in } \overline{D}} \text{ WF-GuardedProcedureDef}$$

$$\frac{\textbf{procedure } a(\overline{\textbf{obs } x : \tau}) \texttt{ ok in } \overline{D} \qquad \Delta_\Sigma(\overline{D}); \overline{x : (\tau, \textbf{obs})} \vdash b \texttt{ ok}}{\textbf{axiom } a(\overline{x : \tau})\ b \texttt{ ok in } \overline{D}} \text{ WF-AxiomDef}$$

Figure 2.14: Typing rules for module-level declarations. We refer to the *prototype* production rule from Figure 2.1 in the *WF-PrototypeDecl* and *WF-GuardedFunctionalDef* typing rules for the purpose of conciseness. The well-formedness rules of module expressions in Figure 2.12 enforce the DAG property of the graph of dependencies of implemented operations on each other, and we do not need to check that it holds here.

Predefined types and operations $\boxed{\Delta \vdash d}$

WF-PredType $\Delta \vdash$ `Pred`

WF-UnitType $\Delta \vdash$ `Unit`

T-True $\Delta \vdash$ `TRUE`$:\longrightarrow$ `Pred`

T-False $\Delta \vdash$ `FALSE`$:\longrightarrow$ `Pred`

T-Or $\Delta \vdash$ `_||_`$:$ (`Pred`, **obs**) $\times$ (`Pred`, **obs**) $\longrightarrow$ `Pred`

T-And $\Delta \vdash$ `_&&_`$:$ (`Pred`, **obs**) $\times$ (`Pred`, **obs**) $\longrightarrow$ `Pred`

T-Implies
$\Delta \vdash$ `_=>_`$:$ (`Pred`, **obs**) $\times$ (`Pred`, **obs**) $\longrightarrow$ `Pred`

T-Equiv
$\Delta \vdash$ `_<=>_`$:$ (`Pred`, **obs**) $\times$ (`Pred`, **obs**) $\longrightarrow$ `Pred`

$$\text{T-Equal}\ \frac{\Delta \vdash \tau}{\Delta \vdash \texttt{\_==\_}: (\tau, \textbf{obs}) \times (\tau, \textbf{obs}) \longrightarrow \texttt{Pred}} \qquad \text{T-Nequal}\ \frac{\Delta \vdash \tau}{\Delta \vdash \texttt{\_!=\_}: (\tau, \textbf{obs}) \times (\tau, \textbf{obs}) \longrightarrow \texttt{Pred}}$$

T-Not $\Delta \vdash$ `!_`$:$ (`Pred`, **obs**) $\longrightarrow$ `Pred`

Figure 2.15: Predefined types and operations in Magnolia. Δ is the typing environment containing the locally available types and operations.

Definition of $\texttt{const-env}(\Gamma)$

$$\texttt{const-env}(\Gamma) = [\texttt{const-var}(v) \mid v \in \Gamma] \qquad \texttt{const-var}((v, \tau, \textbf{obs})) = (v, \tau, \textbf{obs})$$

$$\texttt{const-var}((v, \tau, \textbf{upd})) = (v, \tau, \textbf{obs}) \qquad \texttt{const-var}((v, \tau, \textbf{out})) = (v, \tau, \textbf{unusable})$$

Definition of $\texttt{local-env}(\Delta, \Gamma, \bar{s})$ and $\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \bar{s})$

$$\begin{aligned}
&\texttt{local-env}(\Delta, \Gamma_g, \bar{s}) && = \texttt{update-env}(\Delta, \Gamma_g, \emptyset, \bar{s}) \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \emptyset) && = \Gamma_l \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \textbf{call}\ p(\bar{e}); \bar{s}) && = \texttt{update-env}(\Delta, \Gamma_g, \Gamma_l', \bar{s}) \\
& && \quad \textbf{where}\ \Gamma_l' = [(v, \tau, \textbf{upd}\ \textbf{if}\ v \in \bar{e} \cap (v, \tau, \textbf{out}) \in \Gamma_l\ \textbf{else}\ \omega) \\
& && \qquad\qquad \mid (v, \tau, \omega) \in \Gamma_l] \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \{\overline{s_0}\}; \overline{s_1}) && = \texttt{update-env}(\Delta, \Gamma_g, \texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \overline{s_0}), \overline{s_1}) \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \textbf{var}\ v : \tau; \bar{s}) && = \texttt{update-env}(\Delta, \Gamma_g, (\Gamma_l, (v, \tau, \textbf{out})), \bar{s}) \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \textbf{var}\ v : \tau = e; \bar{s}) && = \texttt{update-env}(\Delta, \Gamma_g, (\Gamma_l, (v, \tau, \textbf{upd})), \bar{s}) \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \textbf{var}\ v = e; \bar{s}) && = \texttt{update-env}(\Delta, \Gamma_g, (\Gamma_l, (v, \tau, \textbf{upd})), \bar{s}) \\
& && \quad \textbf{where}\ \Delta; \Gamma_g \parallel \Gamma_l \vdash e : \tau \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, v' = e; \bar{s}) && = \texttt{update-env}(\Delta, \Gamma_g, [(v, \tau, \textbf{upd}\ \textbf{if}\ v = v'\ \textbf{else}\ \omega) \mid (v, \tau, \omega) \in \Gamma_l], \bar{s}) \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \textbf{if}\ e\ \textbf{then}\ s_t\ \textbf{else}\ s_f; \bar{s}) && = \texttt{update-env}(\Delta, \Gamma_g, \texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, s_t), \bar{s}) \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \textbf{assert}\ e; \bar{s}) && = \texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \bar{s}) \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \textbf{value}\ e; \bar{s}) && = \texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \bar{s}) \\
&\texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \textbf{skip}; \bar{s}) && = \texttt{update-env}(\Delta, \Gamma_g, \Gamma_l, \bar{s})
\end{aligned}$$

Figure 2.16: The $\texttt{const-env}(\Gamma)$ helper function transforms the input variable environment into the corresponding variable environment where all readable variables are **obs** and all unset variables are unusable. $\texttt{local-env}(\Delta, \Gamma, \bar{s})$ traverses a sequence of statements and returns the local environment that this sequence creates. This helper functions works correctly under the assumption that the judgment $\Delta; \texttt{const-env}(\Gamma) \vdash \{\bar{s}\}$ holds. This implies that any variable update within the sequence of statements $\bar{s}$ occurs on a variable defined within the sequence of statements. When calling $\texttt{update-env}_h$ directly instead of `local-env`, one can pre-define an existing local environment—thus allowing to track updates made throughout a block of statement to that pre-populated environment.

2.3.2.7 Functional expressions

Expressions constitute the bodies of functions and predicates. They are free of side-effects—i.e. they are pure. The different kinds of expressions are:

1. *variable references*;
2. *function calls*, which may be accompanied by a type annotation in order to disambiguate calls to functions with overloads where only the return type is different;
3. *conditional (if-then-else) expressions*;
4. *value blocks*, which allow building up an expression from a block of statements terminated by a single *value statement*. Value blocks capture a "frozen" version of the variable context in which they appear, i.e., one in which all readable entities (constants with mode **obs**, and variables with mode **upd**) are considered to have mode **obs**, and all existing but non-readable variables (variables with mode **out**) are unusable but remain in scope. Freezing the external variable context passed to a value block ensures that the expression remains free of side-effects. Retaining non-readable values in scope ensures that the value block does not shadow them—shadowing variables is forbidden in magnoliac's flavour of Magnolia.

Figure 2.17 presents the typing rules for Magnolia's functional expressions.

Typing rules for expressions $\boxed{\Delta; \Gamma \vdash e : (\tau, \omega)}$

$$\textsc{T-VarRef}\ \frac{x : (\tau, \omega) \in \Gamma}{\Delta; \Gamma \vdash x : (\tau, \omega)}$$

$$\textsc{T-FunctionCall}\ \frac{\exists! \tau_r, (f : (\overline{\tau, \textbf{obs}}) \to \tau_r) \in \Delta \quad \overline{\exists \omega \in \{\textbf{obs}, \textbf{upd}\}, \Delta; \Gamma \vdash e : (\tau, \omega)}}{\Delta; \Gamma \vdash f(\overline{e}) : (\tau_r, \textbf{obs})}$$

$$\textsc{T-FunctionCallAnn}\ \frac{(f : (\overline{\tau, \textbf{obs}}) \to \tau_r) \in \Delta \quad \overline{\exists \omega \in \{\textbf{obs}, \textbf{upd}\}, \Delta; \Gamma \vdash e : (\tau, \omega)}}{\Delta; \Gamma \vdash (f(\overline{e}) : \tau_r) : (\tau_r, \textbf{obs})}$$

$$\textsc{T-IfThenElse}\ \frac{\begin{array}{c}\exists \omega_c \in \{\textbf{obs}, \textbf{upd}\}, \Delta; \Gamma \vdash e_c : (\texttt{Pred}, \omega_c) \\ \exists \omega_t \in \{\textbf{obs}, \textbf{upd}\}, \Delta; \Gamma \vdash e_t : (\tau, \omega_t) \qquad \exists \omega_f \in \{\textbf{obs}, \textbf{upd}\}, \Delta; \Gamma \vdash e_f : (\tau, \omega_f)\end{array}}{\Delta; \Gamma \vdash (\textbf{if}\ e_c\ \textbf{then}\ e_t\ \textbf{else}\ e_f) : (\tau, \textbf{obs})}$$

$$\textsc{T-ValueBlock}\ \frac{\begin{array}{c}\nexists e', (\textbf{value}\ e') \in \bar{s} \quad \Delta; \texttt{const-env}(\Gamma) \vdash \{\bar{s}\} \\ \exists \omega \in \{\textbf{obs}, \textbf{upd}\}, \Delta; \texttt{const-env}(\Gamma) \parallel \texttt{local-env}(\Delta, \Gamma, \bar{s}) \vdash e : (\tau, \omega)\end{array}}{\Delta; \Gamma \vdash (\{\bar{s};\ \textbf{value}\ e;\}) : (\tau, \textbf{obs})}$$

Figure 2.17: Magnolia's typing rules at the expression level. Γ is the typing environment containing the type and mode of existing local variables, while Δ is the typing environment containing the available types and operations.

2.3.2.8 Statements

Magnolia allows programming in an imperative fashion using statement blocks (sequences of statements). There are two flavours of statement blocks: the so-called *"value"* block, described previously in the discussion on functional expressions, and the *"effectful"* block, which does not return a value but may modify writeable variables declared in an outer scope. Effectful statement blocks give Magnolia a form of side-effects controlled through variable modes. Unlike the value block, the effectful statement block is itself a statement. The body of a procedure or of an axiom is always an effectful statement block.

Apart from an effectful statement block, a statement can be one of the following:

1. a *procedure call*;
2. a *variable declaration*, which allows for declaring new (typed) variables with mode **out**. Note that once a variable is assigned a value, its mode is automatically set to **upd**;
3. a *variable definition*, which allows for declaring a variable and assigning it a value at the same time. Variable definitions may be given an explicit type;
4. a *variable assignment*, which assigns a new value to an already existing variable. Variable assignment can be thought of as a call to a **procedure** `_=_`(**upd** `var` : *T*, **obs** `val` : *T*) built in for every type *T*;
5. an *effectful conditional statement*. This is the effectful counterpart of the *conditional expression*. For simplicity, we do not allow branches of a conditional statement to affect the existing variable environment in different ways. More concretely, this means that all the pre-existing variables must have the same mode after the conditional statement, regardless of which branch is taken;
6. an *assertion*, which records a predicate expected to hold. Axioms use assertions on universally quantified variables to express semantic constraints on types and operations. The **implement** and **rewrite** module transformations both leverage equational assertions to respectively provide a default implementation for a function, and rewrite terms;
7. a *value statement*, which serves only to return a value from inside value blocks. A value block is always terminated by a single value statement. Conversely, a value statement may only exist as the last statement of a value block;
8. *skip statements*, which are no-ops.

Figure 2.18 formalizes what it means for a statement to be well-formed.

Well-formed statements $\boxed{\Delta; \Gamma \vdash s \ \texttt{ok}}$

$$\textsc{WF-ProcedureCall}\ \frac{(p : (\overline{\tau, \omega}) \to \texttt{Unit}) \in \Delta \quad \overline{\exists \omega' \in \{\omega, \textbf{upd}\}, \Delta; \Gamma \vdash e : (\tau, \omega')}}{\Delta; \Gamma \vdash \textbf{call}\ p(\overline{e})\ \texttt{ok}}$$

$$\textsc{WF-EffectfulBlock}\ \frac{\nexists (v, e, \tau), (s_0 = (\textbf{var}\ v : \tau)) \cup (s_0 = (\textbf{var}\ v = e)) \cup (s_0 = (\textbf{var}\ v : \tau = e)) \cup (s_0 = (v = e)) \cup (s_0 = \textbf{value}\ e) \quad \Delta; \Gamma \vdash s_0\ \texttt{ok} \quad \Delta; \Gamma \vdash \{\bar{s}\}\ \texttt{ok}}{\Delta; \Gamma \vdash \{s_0; \bar{s}\}\ \texttt{ok}}$$

$$\textsc{WF-VarDecl}\ \frac{\nexists e, (\textbf{value}\ e) \in \bar{s} \quad \Delta \vdash \tau \quad \nexists (\tau', \omega'), v : (\tau', \omega') \in \Gamma \quad \Delta; \Gamma, v : (\tau, \textbf{out}) \vdash \{\bar{s}\}}{\Delta; \Gamma \vdash \{\textbf{var}\ v : \tau; \bar{s}\}\ \texttt{ok}}$$

$$\textsc{WF-VarDef}\ \frac{\nexists e', (\textbf{value}\ e') \in \bar{s} \quad \nexists (\tau', \omega'), v : (\tau', \omega') \in \Gamma \quad \exists \tau \exists \omega \in \{\textbf{obs}, \textbf{upd}\}, \Delta \vdash \tau \cap \Delta; \Gamma \vdash e : (\tau, \omega) \cap \Delta; \Gamma, v : (\tau, \textbf{upd}) \vdash \{\bar{s}\}}{\Delta; \Gamma \vdash \{\textbf{var}\ v = e; \bar{s}\}\ \texttt{ok}}$$

$$\textsc{WF-VarTypedDef}\ \frac{\nexists e', (\textbf{value}\ e') \in \bar{s} \quad \nexists (\tau', \omega'), v : (\tau', \omega') \in \Gamma \quad \exists \omega \in \{\textbf{obs}, \textbf{upd}\}, \Delta; \Gamma \vdash \texttt{hint}(\tau, e) : (\tau, \omega) \quad \Delta; \Gamma, v : (\tau, \textbf{upd}) \vdash \{\bar{s}\}}{\Delta; \Gamma \vdash \{\textbf{var}\ v : \tau = e; \bar{s}\}\ \texttt{ok}}$$

$$\textsc{WF-VarAssign}\ \frac{\exists \tau \exists \omega_v \in \{\textbf{out}, \textbf{upd}\} \exists \omega_e \in \{\textbf{obs}, \textbf{upd}\}, \Delta \vdash \tau \cap \Gamma \vdash v : (\tau, \omega_v) \cap \Delta; \Gamma \vdash e : (\tau, \omega_e) \cap \Delta; \Gamma \setminus \{v : (\tau, \omega_v)\}, v : (\tau, \textbf{upd}) \vdash \{\bar{s}\}}{\Delta; \Gamma \vdash \{v = e; \bar{s}\}\ \texttt{ok}}$$

$$\textsc{WF-EffectfulIfThenElse}\ \frac{\exists \omega_c \in \{\textbf{obs}, \textbf{upd}\}, \Delta; \Gamma \vdash e_c : (\texttt{Pred}, \omega_c) \quad \Delta; \Gamma \vdash s_t\ \texttt{ok} \quad \Delta; \Gamma \vdash s_f\ \texttt{ok} \quad \Delta; [v : (\tau, \omega') \mid (v, \tau, \omega') \in \texttt{update-env}(\Delta, \emptyset, \Gamma, s_t), (\exists \omega, v : (\tau, \omega) \in \Gamma)] \vdash \{\bar{s}\} \quad \forall v : (\tau, \textbf{out}) \in \Gamma, \exists \omega \in \{\textbf{out}, \textbf{upd}\}, (v, \tau, \omega) \in \texttt{update-env}(\Delta, \emptyset, \Gamma, s_t) \cap (v, \tau, \omega) \in \texttt{update-env}(\Delta, \emptyset, \Gamma, s_f)}{\Delta; \Gamma \vdash \{\textbf{if}\ e_c\ \textbf{then}\ s_t\ \textbf{else}\ s_f; \bar{s}\}\ \texttt{ok}}$$

$$\textsc{WF-Assert}\ \frac{\exists \omega \in \{\textbf{obs}, \textbf{upd}\}, \Delta; \Gamma \vdash e : (\texttt{Pred}, \omega)}{\Delta; \Gamma \vdash \textbf{assert}\ e\ \texttt{ok}}$$

$$\textsc{WF-Value}\ \frac{\exists \tau \exists \omega \in \{\textbf{obs}, \textbf{upd}\}, \Delta \vdash \tau \cap \Delta; \Gamma \vdash e : (\tau, \omega)}{\Delta; \Gamma \vdash \textbf{value}\ e\ \texttt{ok}}$$

$$\textsc{WF-Skip}\ \Delta; \Gamma \vdash \textbf{skip}\ \texttt{ok}$$

Definition of $\texttt{hint}(\tau, e)$

$$\texttt{hint}(\tau, f(\bar{e})) = f(\bar{e}) : \tau \qquad \texttt{hint}(\tau, e' \mid e' \neq f(\bar{e})) = e'$$

Figure 2.18: Magnolia's typing rules at the statement level. Γ is the typing environment containing the type and mode of existing local variables, and Δ is the typing environment containing the available types and operations. Variable declaration, definition, and assignment only make sense in the context of blocks which affects the presentation of these particular rules. In practice, statements are always in a block.

2.4 Notes on the Implementation of magnoliac

The magnoliac compiler is implemented in Haskell. This choice is motivated partly by features of the language itself (e.g. algebraic data types and pattern matching), partly by existing tooling (e.g. parser combinator libraries), and partly by the host of published compiler-related research that uses Haskell. The implementation of magnoliac draws inspiration from several other compiler codebases, including that of Futhark [**?**], Dex [**?**], and the Glasgow Haskell Compiler (GHC) itself.

2.4.1 Compilation Phases and Passes

The compiler pipeline consists of four main phases (with corresponding passes): *dependency analysis*, *parsing*, *checking*, and *code generation*.

Dependency analysis Let the *main* file for a call to the compiler be the source file containing the **package** whose **program** definitions the user wants to generate code for. The dependency analysis pass takes in an input file path corresponding to the main file for the compilation run, and tries to produce a directed acyclic graph (DAG) containing all the dependencies for the corresponding **package**. We achieve this by parsing only the package header in each file once, and recursively reading new files as needed. Compilation fails if a cycle is detected.

Parsing Parsing is implemented using Megaparsec, a popular Haskell parser combinator library [**?**]. For each package identified as necessary by the dependency analysis pass, we produce a corresponding abstract syntax tree (AST) by parsing the corresponding file—the *parsed AST*. Each node in the parsed AST is annotated with its source location within the file for future error reporting.

Checking The checking pass attempts to construct a *checked AST* from each parsed AST. The pass checks that each package follows the typing and consistency rules outlined earlier in this chaper. Module transformations are applied and top-level references are resolved during the checking phase. All the top-level constructs in the checked AST (i.e. modules, named renamings, and satisfaction relations) are completely inlined. The package itself is also self-contained, i.e., all the modules defined in other packages that are necessary for checking the package are locally reproduced. Nodes in the checked AST are annotated with relevant information derived during the checking phase, e.g. a type for functional expressions, or whether a top-level declaration is locally defined or imported from another package, or all the locations where a given module-level declaration has been defined following **use**s, **require**s, and renamings.

Code generation Code generation transforms each checked **program** that was declared in the main file of the compilation run into a host language-specific AST. The resulting AST can later be pretty printed as source code in the chosen language. Three such languages are currently available: C++, Python, and CUDA—the last only in an experimental branch.

The phases above are executed in order, and the compiler does not move to the next phase if one or more errors have been encountered while processing any of the inputs in the current phase.

Listing 2.3: The monad stack used throughout the compiler passes in magnoliac. While the declaration of `MgMonadT` is generic, `MgMonad` is its only instantiation currently used in practice. In `MgMonad`, the exception type is the unit type, and compiler errors are accumulated in the state of type `Set Err`. This allows us to throw exceptions that we can recover from at any point in the compilation pipeline and to accumulate the encountered errors for reporting at the end of the ongoing compilation pass. The provided environment of type `[Name]` is used to store the names of the parent scopes (e.g. the parent package and module names) in order to construct helpful error messages.

```
newtype MgMonadT e r s m a =
  MgMonadT (ExceptT e (ReaderT r (StateT s m)) a)
  deriving (Functor, Applicative, Monad)

instance MonadIO m => MonadIO (MgMonadT e r s m) where
  liftIO = MgMonadT . liftIO

type MgMonad = MgMonadT () [Name] (Set Err) IO
```

Within a particular phase of compilation, errors are recovered from wherever possible. For example, encountering an error while checking the consistency of a module does not prevent us from checking the consistency of another independent module. Although we know compilation will fail, we aggregate as many useful errors as possible before returning. Listing 2.3 shows the monad stack used to achieve that throughout the compiler.

2.4.2 Definition of the AST

As mentioned above, the *parsed AST* and the *checked AST* both represent Magnolia code—but are not quite the same. First and foremost, the ASTs are annotated with different kinds of information—source information for the parsed AST, but also additional data for the checked AST. Second, the checked AST is more constrained. E.g., it no longer contains any module reference—all top-level declarations are flattened.

The most natural solution for defining these two ASTs is simply to define two different data types for each nesting level (package, module, …). This is frustrating though: since the data types are very similar, the definitions are nearly identical, and utility functions are harder to reuse. The approach does not scale well either: adding more phases with slightly different representations or different annotations causes further duplication.

The problem of needing several similar variants of a given AST is known as the *"tree-decoration problem"*. The problem is solved in magnoliac similarly to how it is solved in GHC—as described in the excellent *"Trees that Grow"* paper [**?**]. The ASTs in magnoliac are defined using a single data type for each nesting level, and using two of the techniques described in the paper: *phase-indexed fields*, and *alternating data types*.

Phase-indexed fields A phase-indexed field is a field within a data type that is of arbitrarily different types during different phases of the compilation. Phase-indexed fields allow changing the type of information associated with nodes throughout the compilation, but also allows enabling or disabling

arbitrary constructors. Phase-indexed fields are the central piece to enabling the reuse of the AST data type throughout the compilation phases. Such fields are implemented using type families [32]. Listing 2.4 shows one use of a phase-indexed field in Magnolia that ensures that renamings are always fully inlined in the checked AST.

Listing 2.4: Definition of the `XRef` type family. `PhParse` and `PhCheck` are used as phantom types corresponding respectively to the parsing phase and to the checking phase of compilation. The `MRenaming'` data type describes a renaming. A renaming may be defined inline, or reference a named renaming. Because `XRef PhCheck` returns `Void`, it is impossible to (sanely) construct a checked `MRenaming'` that is a reference to a named renaming. The checked AST thus enjoys the additional static guarantee that no such unresolved reference remains compared to the parsed AST.

```
-- data FullyQualifiedName = ...
-- data InlineRenaming = ...

data PhParse
data PhCheck

type family XRef p where
  XRef PhParse = FullyQualifiedName
  XRef PhCheck = Void

data MRenaming' p = InlineRenaming InlineRenaming
                  | RefRenaming (XRef p)
```

Alternating data types In order to provide relevant annotations for each node in the ASTs (e.g., the source information for each node in the parsed AST), each nesting level in the AST has two type definitions: one for the data type itself, and one for the annotated version of the data type. Listing 2.5 shows an excerpt of the code.

Listing 2.5: Definition of the `Ann` annotation data type and example use to annotate the `MModule'` data type. The `e` type parameter corresponds to a phase-indexed node type to annotate, and the `p` type parameter is a phantom type corresponding to a compilation phase. The type of the annotation is itself a phase-indexed field with type `XAnn p e`. By making the annotation type a phase-indexed type instance of the `XAnn` type family, we are free to arbitrarily change the type of the annotation as needed throughout the compilation pipeline.

```
type family XAnn p (e :: * -> *)

data Ann p e = Ann { _ann :: XAnn p e
                   , _elem :: e p
                   }

type MModule p = Ann p MModule'
data MModule' p = ...
```

Using a single definition for the AST has its own drawbacks. Namely, it makes compilation slower[1],

[1] Haskell type families are painfully slow, see the longstanding issue at https://gitlab.haskell.org/ghc/ghc/-/issues/8095.

makes the code somewhat more complex, and forces us to write a bit of code to handle unavailable constructors. All in all, the trade-off seems worth it here. Such a design makes the AST reusable and extensible without duplication—and making magnoliac easy to reuse and extend is one of our design goals.

Part II

Scientific Results

Paper 1

Revisiting Language Support for Generic Programming: When Genericity Is a Core Design Goal[1]

Benjamin Chetioui[a], Jaakko Järvi[b], Magne Haveraaen[a]

[a]Department of Informatics, University of Bergen, Norway, [b]Department of Computing, University of Turku, Finland

Abstract

Context Generic programming, as defined by Stepanov, is a methodology for writing efficient and reusable algorithms by considering only the required properties of their underlying data types and operations. Generic programming has proven to be an effective means of constructing libraries of reusable software components in languages that support it. Generics-related language design choices play a major role in how conducive generic programming is in practice.

Inquiry Several mainstream programming languages (e.g. Java and C++) were first created without generics; features to support generic programming were added later, gradually. Much of the existing literature on supporting generic programming focuses thus on retrofitting generic programming into existing languages and identifying related implementation challenges. Is the programming experience significantly better, or different when programming with a language designed for generic programming without limitations from prior language design choices?

Approach We examine Magnolia, a language designed to embody generic programming. Magnolia is representative of an approach to language design rooted in algebraic specifications. We repeat a well-known experiment, where we put Magnolia's generic

[1]Benjamin Chetioui, Jaakko Järvi, and Magne Haveraaen. Revisiting language support for generic programming: When genericity is a core design goal. *The Art, Science, and Engineering of Programming*, 7(2), oct 2022. doi:10.22152/programming-journal.org/2023/7/4

1

programming facilities under scrutiny by implementing a subset of the Boost Graph Library, and reflect on our development experience.

Knowledge We discover that the idioms identified as key features for supporting Stepanov-style generic programming in the previous studies and work on the topic do not tell a full story. We clarify which of them are more of a means to an end, rather than fundamental features for supporting generic programming. Based on the development experience with Magnolia, we identify variadics as an additional key feature for generic programming and point out limitations and challenges of genericity by property.

Grounding Our work uses a well-known framework for evaluating the generic programming facilities of a language from the literature to evaluate the algebraic approach through Magnolia, and we draw comparisons with well-known programming languages.

Importance This work gives a fresh perspective on generic programming, and clarifies what are fundamental language properties and their trade-offs when considering supporting Stepanov-style generic programming. The understanding of how to set the ground for generic programming will inform future language design.

1.1 Introduction

It is routine in programming to parameterize algorithms and data structures by type to make them reusable in different contexts. The mechanisms for implementing generic code, however, vary from one language to the other. These details matter: Garcia et al. [61] evaluated and compared the level of support for generic programming in several programming languages (C++, SML, OCaml, Haskell, Eiffel, Java, C#, and Cecil), and showed that many language design choices related to generics significantly influence how conducive that language is in practice to generic programming. This work has had an influence on the design of programming languages (see, e.g., C++'s Concepts [76], Haskell's associated types [32], and Siek and Lumsdaine's idealized *G* language [159; 163] for generic programming).

Generic features are now common features of most widely used languages, and for many of them, these features were an afterthought. The list of such languages has kept growing—examples of languages with recent or planned generic features include Fortran [91], Go [77], ECMAScript, TypeScript, and FlowType. Retrofitting tends to lead to compromises, which raises the questions whether the set of features for generic programming would look the same for languages that incorporate support for generic programming as part of their initial design, and how such potentially different designs support generic programming in practice. This paper sheds light on these questions by examining language designs rooted in algebraic specification. In particular, we conduct a case study and analyse in details the features and programmability of one language representative of the approach, and discuss the findings in the general context of languages that follow the same algebraic design principles. The goal is to inform future language designs, so that new languages could support generic programming without pitfalls identified by Garcia et al.

Interpretations of generic programming vary depending on what kind of parameterization a programming language supports. Gibbons gives a taxonomy for some interpretations of genericity [62] which we reuse here. Programs parameterized by type constructors give rise to *genericity by shape*, through *datatype-generic programming* (also called *polytypism*) [11; 62]. This is the interpretation chosen by, e.g., Generic Haskell [93]. In the object-oriented world, generic programming refers

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.
- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized to the concrete case, the result is just as efficient as the original algorithm.
- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.
- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.

Figure 1.1: Definition of generic programming from Jazayeri, Musser, and Loos [102].

primarily to *generics* or *parametric polymorphism* [29; 97], that is *genericity by type*. We add qualifiers such as *bounded* or *constrained* to these terms, and mean roughly the same things. Algebraic specifications are the basis for another approach to generic programming called *parameterized programming*. Parameterized programming has been concretized prominently in the OBJ family of languages, e.g. in OBJ2 [59], OBJ3 [70], CafeOBJ [50], and Maude [43; 45]. C++ *concepts* (as proposed for C++11), which describe syntactic and semantic requirements on data structures and algorithms [76], also descend from this approach based on algebraic specifications. Concepts, as implemented in C++20, only support syntactic requirements: we talk about *genericity by structure*. In the fully-fledged version of concepts, when both syntactic and semantic requirements are supported, we talk about *genericity by property*. C++ concepts were born out of Stepanov's work on generic programming [49; 104]. This paper, following Garcia et al., takes the notion of generic programming as introduced by Musser and Stepanov in their seminal work in 1988 [137]. Figure 1.1 reproduces their structured definition of generic programming, taken from Jazayeri, Musser, and Loos [102].

We employ Garcia et al.'s framework for evaluating languages for generic programming to assess the approach based on algebraic specifications through an experiment with the Magnolia programming language [14]. This research language was first developed more than a decade ago, and is now again under active development [35]. We repeat Garcia et al.'s experiment of implementing a subset of the Boost Graph Library [164] (BGL), rich in generic definitions, to put the generic programming facilities of a language under rigorous scrutiny.

Magnolia is designed as an embodiment of a language for Stepanov-style generic programming. Magnolia's main type of genericity is thus *genericity by property*, as the language allows the specifications of algebraic signatures along with semantic requirements on their behavior, i.e., concepts. Magnolia does not offer any primitive type (beyond predicates), and it is designed to be parameterized by a host programming language and data structures implemented in that language. In the style of Gibbons's taxonomy, we coin the term *genericity by host language* to refer to the type of generic programming enabled by this axis of parameterization. One can implement composite operations in

Magnolia—all base types and their operations, even loop structures, come from libraries written in the host language. Magnolia has a transpiler architecture, where the boundary between the language (Magnolia) and the base library (written in the host language) is not predefined, but rather the programmer can freely place it where convenient.

Garcia et al.'s work [61] to implement the same generic library in a variety of languages led the authors to identify several language properties that are useful and/or necessary for effective generic programming. Siek and Lumsdaine, in the context of developing the *G* programming language, extended this set of properties [163]. These sets of properties served as the language evaluation framework in the above two works. We adopt this framework on the one hand to assess Magnolia's support for generic programming and on the other hand to relate its somewhat unorthodox language design to (more) mainstream languages. The listing of the identified properties, with our additions, is shown in Figure 1.2.

Following the recipe of the prior works, we implement a fragment of a generic graph library modeled after the BGL, in Magnolia, and analyze the result with regards to each identified property. This experiment allows us to extract several insights into generic programming, which we discuss in the paper. We highlight two particularly noteworthy aspects of the Magnolia BGL fragment. First, we implement both C++ and Python backend libraries. The same generic Magnolia code that captures the essence of graph algorithms can then be transpiled to either of these languages. We achieve an additional level of genericity, i.e., genericity by host language. Second, we show how a (seemingly) sequential generic algorithm can be transformed into one that is parallel, by picking appropriate backend data structures. This is achieved by abstracting the iteration mechanism. Magnolia does not offer any built-in looping constructs, and repetitions are thus necessarily expressed as generic abstractions.

The paper is structured as follows. Section 1.2 describes the landscape of languages designed for generic programming based on algebraic specifications, and explains how the approach is concretized, first in Maude and then in Magnolia. Section 1.3 presents our small graph library and discusses its Magnolia implementation. Section 1.4 situates Magnolia within the landscape of generic languages. It also makes connections and comparisons with other languages, and discusses related work. Section 1.5 discusses the performance of our implementation. Section 1.6 reflects on our approach and the insights we gained by developing the graph library. All the code discussed in this paper is made available [35].

1.2 Languages Designed for Generic Programming: The Approach of Algebraic Specifications

Algebraic specifications are at the core of Stepanov's work on generic programming [49; 104; 137; 165]. Highly influential early work in the field is Goguen's parameterized programming that emphasizes code reuse and modularity [60; 63]. Siek characterizes parameterized programming as similar to Stepanov's notion of generic programming, but without the same emphasis on efficiency [159]. Parameterized programming thus also aims at expressing algorithms in their most general form, making both their syntactic and semantic requirements explicit, and well organized.

1.2.1 Algebraic Specifications and Maude

Algebraic specifications and Goguen and Burstall's theory of institutions [67] have guided the design of the OBJ language family [71] (OBJ2, OBJ3, CafeOBJ, Maude…). These languages provide extensive support for parameterized programming by design. OBJ2 and OBJ3 are both implementations of the OBJ logical programming language that differ in their operational semantics [70]. Maude incorporates most features of OBJ3 and significantly expands the capabilities of OBJ2 and OBJ3 for parameterized programming. Maude and CafeOBJ are still under active development. We describe below the general design of languages intended to support generic programming using algebraic specifications, and explain how it is concretized in Maude. Maude is based on rewriting logic [42; 66], and uses *membership equational logic* as its underlying equational logic. Our discussion only touches upon the fragment of Maude related to membership equational logic, where Maude's support for parameterized programming is concretized.

The general approach relies on a bilevel module system, with modules that allow for specifying generic APIs on the one hand and modules that allow for writing concrete programs on the other hand [69]. Modules of the same kind may be composed, and program modules can be parameterized by specification modules. Specifications consist of an algebraic signature defining sorts and (total and partial) operations, along with semantic requirements on their behaviour called *axioms*. Satisfaction relations can be expressed which describe how a program (or a specification) satisfies the requirements of a given specification.

Specifications are given in Maude through *functional theories*—Goguen introduced the notion of types as theories [65]. Functional theories allow expressing semantic requirements using *equations* and *conditional equations*. In addition, Maude allows the specification of *subsorting relations* along with *membership axioms*. This approach allows flexible control of partiality and declaring relationships between types, e.g., natural numbers and integers. The choice of implementing partiality using subsorting has consequences on other language features. For example, it poses restrictions on overloading and thus also on the ability to compose two arbitrary theories—see Listing 2 (in Appendix 3.3) for an example. Note the similarity of this approach to *refinement types*—where the refined type $\{t : T \mid P\}$ is the subset of type T for which the formula P holds [57; 73]. Refinement types are closely related to subtyping.

Maude's *functional modules* allow for writing programs using the same constructs as functional theories—where equations and conditional equations define functions and data types in lieu of functional theories' semantic requirements, and where the rewriting system engendered by these equations must be confluent and terminating. The semantics of a functional module in Maude is the initial algebra defined by the module's equations, and evaluation is performed using an equational rewriting engine. Functional modules can be parameterized by functional theories: we speak of *parameterized functional modules*. Maude programs can be metarepresented as data and manipulated to produce new programs. This powerful mechanism of reflection allows generating so-called *dependent parameterized modules* such as n-tuples containing n sorts and n projection functions [45, Section 21.3.1]. Maude's built-in types are efficiently implemented in C++. Contrarily to the previous OBJ2 and OBJ3, Maude does not allow the user to implement custom primitive types in an external language.

Satisfaction relations in Maude are stated through *views*. Every sort (respectively function) in the view's source theory must be mapped (renamed) to a corresponding sort (respectively function) in the view's target module, and the mappings must preserve the subsorting structure of the source

1

theory in the target module. It is also possible to implement functions on the fly to resolve signature mismatches.

1.2.2 Magnolia

As alluded to above, the Magnolia programming language is designed for Stepanov-style generic programming—i.e. parameterized programming with an added emphasis on efficiency. The language takes the same general approach based on algebraic specifications as described above, and its module system is likewise based on Goguen and Burstall's theory of institutions.

Listing 1.1 shows uses of the different module types. A **signature** allows defining types and operations. A **concept** is a **signature** augmented with **axiom**s that restrict the properties of the types and operations. A **concept** serves the same purpose as a functional theory in Maude, and the **signature** and **concept** modules constitute the specification layer of the module system. An **implementation** allows the same declarations as a **signature**, but also the definition of generic operation implementations; it is the equivalent of a parameterized functional module in Maude. A **program** is a specific kind of **implementation** in which all the specified operations and types are matched with (non-generic) concrete implementations; either Magnolia code that has a concrete implementation or an implementation in the base library in the host language. The **implementation** and **program** modules constitute the program layer of the module system. Constructs analogous to Maude's metaprogramming facilities are under investigation for Magnolia through Syntactic Theory Functors (STFs) [87] but the Magnolia compiler supports only specific instances of STFs at the moment [39].

Types (sorts) in Magnolia are opaque identifiers. One cannot explicitly parameterize them, nor can one define relations such as subtyping relations between them. Operations can be **function**s, **procedure**s, or **predicate**s. Procedure calls are prefixed with the **call** keyword, while function calls follow the usual uncurried call syntax. Predicates are treated as functions with a built-in, non-reimplementable return type. Magnolia's approach to partiality is based on guarded algebras [88]: an operation can be guarded by a predicate, which then acts as a precondition. In addition to their types, a **procedure** associates modes to its arguments: **obs** (read-only), **upd** (can be read and written to), and **out** (write-only, and must be written to) [18]. `ExampleProgram` in Listing 1.1 shows equivalent implementations of a multiplication by three as a **procedure** (`timesThreeUpdateRef`) and as a **function** (`timesThree`). In the example's program, the `int` type and `add` function are externally defined in Python and come from `PyConcreteSemigroup`. The line **use** `Magma[ T => int, bop => add ]` applies a renaming function to the content of the `Magma` **signature** and brings it into scope. The renaming maps `T` to a new name `int`, and `bop` to a new name `add`. It is assumed that the primitives implemented in the host language do not have side-effects, except for the modification of arguments to procedures where the argument mode is **out** or **upd**.

A **satisfaction** allows defining a modeling relation between an **implementation** and a **concept**; or between two **concept**s—it is the equivalent of a view in Maude. Signature mismatches are resolved through the renaming mechanism.

Magnolia semantics are tightly coupled to abstracting over hardware features: primitive types and operations may directly represent characteristics of the underlying hardware architecture, such as instruction sets, memory layout, etc. This enables Magnolia code to run efficiently on a variety of

Listing 1.1: The main Magnolia building blocks.

```
signature Magma = {
  type T;
  function bop(t1: T, t2: T): T;
}

concept Semigroup = {
  use Magma;
  axiom bopIsAssociative(t1: T, t2: T, t3: T) {
    assert bop(t1, bop(t2, t3)) == bop(bop(t1, t2), t3);
  }
}

implementation PyConcreteSemigroup =
  external Python lib.int_impl {
    use Magma[ T => int, bop => add ];
    use Magma[ T => int, bop => mul ];
  }

program ExampleProgram = {
  use PyConcreteSemigroup;
  procedure timesThreeUpdateRef(upd i: int) {
    i = add(add(i, i), i);
  }

  function timesThree(i: int): int {
    var mutable_i = i;
    call timesThreeUpdateRef(mutable_i);
    value mutable_i;
  }
}

satisfaction ExampleProgramHasAddSemigroup =
  ExampleProgram models Semigroup[ T => int, bop => add ];

satisfaction ExampleProgramHasMulSemigroup =
  ExampleProgram models Semigroup[ T => int, bop => mul ];
```

hardware, and to explore software for high-performance computing (HPC) [39]—making it suitable to address also the efficiency aspect of generic programming. This feature enables the user to utilize features of new hardware, e.g., posit numbers [80] by writing code directly in the targeted host language.

The notion of concepts, around which specifications in Magnolia are constructed, is from Stepanov and Musser [137]. These foundational building blocks of generic specifications and programs manifested in C++ first as mere documentation, then as library "hacks" [162; 169], and later as a language feature. The first proposals, see Siek's account of the history [160], were quite ambitious, including, e.g., semantic constraints (like Magnolia's axioms) and concept-based overloading, but their current form is somewhat scaled back. It is clear that concepts as a notion and language feature has been a highly influential contribution.[2]

We use Magnolia as a representative for languages designed for generic programming based on algebraic specifications throughout the remainder of the paper. The design of Magnolia and languages in the OBJ family draw from the same foundations, and the conclusions we draw about Magnolia should apply to these languages as well.

1.3 Graph Library in Magnolia

The subset of the BGL we implemented is a bit larger than the subset that Garcia et al. used. It consists of the six generic algorithms implemented by Garcia et al., i.e. Graph Search, Breadth-First Search (BFS), Dijkstra's single-source shortest paths, Bellman-Ford's single-source shortest paths, Johnson's all-pairs shortest paths, and Prim's minimum spanning tree, as well as a seventh algorithm, namely Depth-First Search (DFS). Like in Garcia et al.'s first study, we omit discussion of most algorithms for the sake of brevity—and discuss mainly our implementation of the BFS algorithm. This implementation is at the core of the library we implemented, and follows the same pattern as BGL's sequential implementation of BFS, whose pseudo-code is given in Listing 1.2.

We later show how careful choices in the instantiation of backend data structures allow using the same (seemingly) sequential BFS code to realize a parallel breadth-first graph traversal algorithm.

1.3.1 Implementing the Graph Algorithms

The BGL's implementation is based on the textbook BFS algorithm from Cormen et al.'s "Introduction to Algorithms" [46] that maintains the state of the traversal using a color map indexed by vertices. BGL's version adds to the algorithm various user-parameterizable visitor events, shown by the commented out actions in Listing 1.2. E.g., the "discover vertex" action is performed every time a vertex is encountered for the first time. The visitor events may modify the vertex queue (worklist) as well as an arbitrary user-provided state. By carrying around the right state and providing appropriate actions for each event, many algorithms can be built on top of the generic BFS implementation.

[2] As a case in point, the 2021 ACM SIGPLAN International Conference on Software Language Engineering's "Most Influential Paper Award" was given to *Design of Concept Libraries for C++* by Sutton and Stroustrup [170], https://twitter.com/bcombemale/status/1449743946268221440.

Listing 1.2: Pseudo-code for the BFS algorithm implemented in the BGL [164]. Taken from https://www.boost.org/doc/libs/1_79_0/libs/graph/doc/breadth_first_search.html with minor stylistic changes.

```
1  BFS(G, s)
2    for each vertex u in V[G] // initialize vertex u
3      color[u] := WHITE
4      d[u] := infinity
5      p[u] := u
6    end for
7    color[s] := GRAY
8    d[s] := 0
9    ENQUEUE(Q, s) // discover vertex s
10   while (Q != ∅)
11     u := DEQUEUE(Q) // examine vertex u
12     for each vertex v in Adj[u] // examine edge (u,v)
13       if (color[v] = WHITE) // tree edge (u,v)
14         color[v] := GRAY
15         d[v] := d[u] + 1
16         p[v] := u
17         ENQUEUE(Q, v) // discover vertex v
18       else // non-tree edge (u,v)
19         if (color[v] = GRAY)
20           ... // gray target (u,v)
21         else
22           ... // black target (u,v)
23     end for
24     color[u] := BLACK // finish vertex u
25   end while
26   return (d, p)
```

1

Dijkstra's algorithm, for instance, can be implemented by carrying a state containing edge costs and vertex costs, and by updating the vertex costs every time an edge is examined.

The corresponding Magnolia implementation is split up into several functions across several modules and is rather lengthy. To improve readability, we put the full listings accompanying this section in Appendix 3.3, and intersperse only excerpts with our text here. Listing 3 presents the `GenericBFSUtils` module, corresponding to lines 7 to 26 in Listing 1.2.

```
procedure breadthFirstVisit(obs g: Graph,
    obs s: VertexDescriptor, upd a: A, upd q: Queue,
    upd c: ColorPropertyMap) {
  call discoverVertex(s, g, q, a);
  call push(s, q);
  call put(c, s, gray());
  call bfsOuterLoopRepeat(a, q, c, g);
}
```

The entry point in `GenericBFSUtils` is `breadthFirstVisit`, which discovers the initial vertex and adds it to the queue, before calling `bfsOuterLoopRepeat`. The `bfsOuterLoopRepeat` procedure corresponds to the outer while loop in Listing 1.2 (lines 10 to 25), with the body of the loop implemented in `bfsOuterLoopStep` (reproduced below); we discuss this in more detail in Subsection 1.3.3.

```
procedure bfsOuterLoopStep(upd x: A, upd q: Queue,
    upd c: ColorPropertyMap, obs g: Graph) {
  var u = front(q);
  call pop(q);
  call examineVertex(u, g, q, x);
  var edgeItr: OutEdgeIterator;
  call outEdges(u, g, edgeItr);
  call bfsInnerLoopRepeat(edgeItr, x, q, c, g, u);
  call put(c, u, black());
  call finishVertex(u, g, q, x);
}
```

Next is `bfsInnerLoopRepeat`, which corresponds to the inner for-each loop in Listing 1.2 (lines 12 to 23). The inner loop's body is implemented in `bfsInnerLoopStep` (see Appendix 3.3).

The initialization of the queue and the color map is done in `search`, which is part of the `GraphSearch` module presented in Listing 1.3.

The `search` function is an entry point for simple graph searches in which an empty constructor that takes no argument exists for the queue. This is the case for a FIFO queue for instance, but not necessarily for a priority queue. For example, to implement Dijkstra's algorithm, we might want to use a priority queue that stores the shortest measured distance from the source to each vertex. The empty constructor for such a queue would take this information as a parameter—thus exposing a different API.

Listing 1.3: Implementation of a graph search entry point in Magnolia.

```
implementation GraphSearch = {
  use GenericBFSUtils;
  require function empty(): Queue;

  function search(g: Graph, start: VertexDescriptor,
      init: A): A = {
    var q = empty(): Queue;
    var vertexItr: VertexIterator;
    call vertices(g, vertexItr);
    var c = initMap(vertexItr, white());
    var a = init;

    call breadthFirstVisit(g, start, a, q, c);
    value a;
  }
}
```

Listing 1.4 completes the implementation of the BFS: the types and operations are renamed and the underlying queue data structure is set to be a FIFO queue.

Listing 1.4: Implementation of a BFS in Magnolia.

```
implementation BFS = {
  use GraphSearch[ search => breadthFirstSearch,
    Queue => FIFOQueue ];
  use FIFOQueue[ A => VertexDescriptor,
    isEmpty => isEmptyQueue ];
}
```

By keeping the requirements on the queue implementation loose in the `GraphSearch` module, we can produce a DFS implementation following the same pattern as in Listing 1.4—but using a LIFO queue (i.e., a stack) instead of a FIFO queue, and with appropriate renamings. Listing 4 (in Appendix 3.3) shows how.

Dijkstra's algorithm is also implemented reusing the code in `GenericBFSUtils`, this time using a priority queue.

1.3.2 Specifying and Instantiating Data Structures

Both the FIFO queue and the stack concepts are easily derived from the generic `Queue` concept in Listing 1.5; the stack case is shown in Listing 1.6. Note that the concept of a stack exposes an operation named `top` instead of one named `front`. Thanks to the use of Magnolia's powerful renaming mechanism, this is not a problem: we can instantiate generic algorithms with data structures that provide the expected API up to renaming.

The concept in Listing 1.6 describes a stack by virtue of the axioms that refine a generic queue

Listing 1.5: Specification of a generic queue in Magnolia.

```
concept Queue = {
    require type A;
    type Queue;

    predicate isEmpty(q: Queue);
    procedure push(obs a: A, upd q: Queue);
    procedure pop(upd q: Queue) guard !isEmpty(q);
    function front(q: Queue): A guard !isEmpty(q);
}
```

Listing 1.6: Specification of a generic stack in Magnolia.

```
concept Stack = {
  use Queue[ Queue => Stack, front => top ];

  function empty(): Stack;
  axiom pushPopTopBehavior(s: Stack, a: A) {
    var mut_s = s;
    call push(a, mut_s);
    assert top(mut_s) == a;

    call pop(mut_s);
    assert mut_s == s;
  }
  axiom emptyIsEmpty() {
    assert isEmpty(empty());
  }
}
```

concept's behavior. Magnolia allows specifying axioms as part of concepts. They place restrictions on the behavior of operations' implementations. The `pushPopTopBehavior` axiom, for example, tells us that whenever a value `a` is pushed to any stack `s`, calling `top` on the resulting stack `s'` yields `a`; similarly, calling `pop` on `s'` yields `s`.

Possible (hand-coded) user-provided backend data structure implementations for the stack concept of Listing 1.6 are given in Appendix 3.3 in Listings 5 (for C++) and 6 (for Python).

1.3.3 Abstracting the Schedule of the Algorithms

When comparing the Magnolia implementation to the pseudo-code in Listing 1.2, one can notice that the former has no loop structure. The outer (while) loop in the pseudo-code is implemented by a triplet of operations: `bfsOuterLoopCond`, which corresponds to the condition of the loop, `bfsOuterLoopStep`, which corresponds to the body of the loop, and `bfsOuterLoopRepeat`, which is called to start the loop. The inner for-each loop is implemented by a pair of operations, `bfsInnerLoopRepeat` and `bfsInnerLoopStep`.

Though this may seem tedious, it is by design that Magnolia provides no loop structure. The ideal manner to schedule and allocate computations (in a loop or otherwise) depends heavily on the hardware architecture, and by not having loops Magnolia forces this choice to remain a parameter, defined in a base library in the host language.

A generic specification of a while loop in Magnolia is presented in Listing 1.7, and a corresponding C++ backend data structure implementation is shown in Listing 7—the latter listing can be found in Appendix 3.3. The `WhileLoop` concept describes an API that takes two types (`Context` and `State`) and two operations (`cond` and `step`), and provides a `repeat` procedure whose behavior must correspond to the constraints expressed in the `whileLoopBehavior` axiom. By implementing projections on the opaque `Context` and `State` types, and updates on `State`, we can carry around arbitrarily complex contexts and states.

For the experiments described in Section 1.5, we implemented the loops of Listing 3 differently, to carry several state and context arguments. This was done for performance reasons, to avoid the overhead of packing and unpacking the `State` and `Context` objects. Magnolia lacks *variadics*, definitions that are generic on arity, i.e. on the number of arguments. Such a feature could let us avoid the packing and unpacking without the need to specify different concepts. We discuss this further in Section 1.4.

Abstracting away the loop structure (instead of providing a native Magnolia construct) has advantages: repetition can be implemented differently for different data structures or different target architectures. In our small BGL fragment, we exploited this aspect of Magnolia to provide two different backend implementations (in C++) for the inner for-each loop of the BFS algorithm: one that uses a sequential for loop and another that uses a parallel for loop based on OpenMP [143]. This is possible because the algorithm does not enforce a processing order on not-yet-visited vertices adjacent to the current vertex—the iterations of the inner loop are independent. The parallel version of the code also requires using explicitly thread-safe data structures for the vertex queue (and for the user-provided state, depending on how it is modified by the visitor events).

The difference in code when going from sequential to parallel is minimal: when concretizing our generic BFS algorithm, it suffices to **use** three modules exposing the same API as their sequential

1

Listing 1.7: Specification of a generic while loop in Magnolia.

```
concept WhileLoop = {
    require type Context;
    require type State;

    require predicate cond(s: State, c: Context);
    require procedure step(upd s: State, obs c: Context);
    procedure repeat(upd s: State, obs c: Context);

    axiom whileLoopBehavior(s: State, c: Context) {
        if cond(s, c) then {
            // if the condition holds, then doing one step
            // and completing the loop is the same as just
            // completing the loop
            var mutableState1 = s;
            var mutableState2 = s;
            call repeat(mutableState1, c);
            call step(mutableState2, c);
            call repeat(mutableState2, c)
            assert mutableState1 == mutableState2;
        }
        else {
            // otherwise, the state shouldn't change
            var mutableState1 = s;
            call repeat(mutableState1, c);
            assert mutableState1 == s;
        };
    }
};
```

counterpart, but with different properties. By not committing to a looping mechanism too early, we gain a new powerful axis of parameterization.

1

1.4 Generic Features: Evaluation

With an understanding of the Magnolia implementation of the generic graph library, we can relate the code to the important language properties for generic programming identified by Garcia et al. [61] and Siek and Lumsdaine [163]. Figure 1.2 summarizes how Magnolia fares.

	C++	SML	OCaml	Haskell	Java	C#	Cecil	C++0X	*G*	Magnolia
Multi-type concepts	-	●	○	●	○	○	◒	●	●	●
Multiple constraints	-	◒	◒	●	●	●	●	●	●	●
Associated type access	●	●	◒	●	◒	◒	◒	●	●	●
Constraints on associated types	-	●	●	●	◒	◒	●	●	●	●
Retroactive modeling	-	●	●	●	○	○	●	●	●	●
Type aliases	●	●	●	●	○	○	○	●	●	●
Separate compilation	○	●	●	●	●	●	◒	○	●	○
Implicit argument deduction	●	○	●	●	●	●	◒	●	●	○
Modular type checking	○	●	◒	●	●	●	◒	◒	●	●
Lexically scoped models	○	●	○	○	○	○	○	●	●	●
Concept-based overloading	●	○	○	○	○	○	●	●	◒	○
Same-type constraints	-	●	○	●	○	○	○	●	●	○
First-class functions	○	●	●	●	○	◒	●	●	◒	○
Property-based specifications	○	○	○	◒	○	○	○	●	○	●
Variadics	●	○	●	●	◒	◒	○	●	○	○

Figure 1.2: The level of support in Magnolia for properties for generic programming. For the reader's convenience, we reproduce here the original characterization of C++, SML, OCaml, Haskell, Java, C#, Cecil, C++ox, and *G* from Siek and Lumsdaine [163] (omitting footnotes with detailed commentary). ● indicates full support, ○ indicates poor support, and ◒ indicates partial support. The rating of "-" for C++ indicates that while C++ does not explicitly support the feature, one can still program as if the feature were supported. The level of support for *property-based specifications* and *variadics* is indicated for the latest release of each language at the time of writing, i.e. respectively, for the first seven columns, C++20, SML'97, OCaml 4.14, Haskell 2010, Java 18, C# 11, and Cecil 3.2 [33]. We evaluate C++ox as it was envisioned, as opposed to its eventual partial adoption in C++11. To the best of the authors' knowledge, *G* has only had one release.

We should be cognizant that the list of properties is a reflection of the desire to express generic programs well in mainstream multi-paradigm languages, and maybe even based on experiences and programming idioms of C++. This is understandable: while Stepanov's and Musser's generic programming notions evolved through many languages, including Scheme and Ada, they materialized most prominently in C++. It is thus the case that even though the evaluation with the listed properties revealed shortcomings in programming languages, the properties arose from a C++-centric view of generic programming. Some are artifacts of this view and others more of a means to an end, rather than an essential part of a foundation for generic programming. Indeed, despite the several empty bullets in Magnolia's column in Figure 1.2, the BGL experiment was successful, likely because Magnolia builds its generics on somewhat different foundations than any of the languages studied by Garcia et al. (we like to think that it is closer to Stepanov's and Musser's ideals).

We also note that while the list of properties is rather comprehensive, we did end up adding two new items: variadics and property-based specifications. These are not relevant only to Magnolia, but would have been interesting topics of study in the original evaluation as well: variadic templates were studied after Garcia et al.'s evaluation [75] and the feature is today part of standard C++; property-based specifications (axioms) were proposed to be included in C++, e.g., for enabling optimizations, and the Haskell GHC compiler supports such specifications (in compiler pragmas) for rewriting [145].

For each of the properties listed in Figure 1.2, we give below its definition as given by Siek and Lumsdaine [163], motivate it briefly, and discuss its relevance and realization in Magnolia. We do not do any reimplementation for the previously studied languages. However, for the new properties we introduced, we also discuss their realization in the most recent release of the previously studied languages at the time of writing.

Multi-type concepts

A concept can be implemented by a collaboration of several types.

Multi-type concepts in generic programming correspond to multi-sorted signatures with axioms in algebraic specifications, and both arise naturally and often. Magnolia's concepts can declare any number of types and define syntactic and semantic requirements on any combination of them. Further, the partial order of concepts that arises from Magnolia concept definitions and their **use** declarations is not in any way constrained by the types declared in the concept that *uses* or the concept that is being *used*. Any name conflicts that might arise are easily resolved with renaming types and operations. Magnolia thus fully supports multi-type concepts.

By contrast, in many other languages, in particular in object-oriented ones, concepts are approximated by interfaces/classes, which are types. These interface/class types are treated differently from other types of the concept (defined as type parameters to the generic interface), which introduces many restrictions, and obstacles for the clean expression of generic programs [61; 101].

Multiple constraints

More than one constraint can be placed on a type parameter.

A few of the languages studied by Garcia et al. had restrictions when constraining types by more than one concept; the reasons are technical, and discussed in prior work [61]. In Magnolia there are no restrictions: *multiple constraints* merely mean that a particular type appears in more than one **use** declaration. All definitions from used concepts are brought to the same scope; the type's constraints are thus a union of its requirements in these concepts.

Associated type access

Types can be mapped to other types within the context of a generic function.

In most object-oriented languages studied by Garcia et al., the only way to declare types of a concept is as type parameters of a generic. The evaluation called for a mechanism for defining type members,

1

"associated" types of the "main" type(s) of the concept. One could do this in C++ with trait classes and in Haskell, at the time with the *functional dependencies* extension, later with associated types [32]. Today, Rust and Swift also have similar notions. Associated types solve problems of instantiating concepts with positional type parameters, e.g., that they can shorten the parameter lists considerably. Järvi et al. [101] detail these problems in Java and C#.

In Magnolia, there is no distinction between main types and associated types. All types are opaque, and accessible by their name. Magnolia **use**s of concepts only mention the types (and operations) that the programmer needs or wants to rename—there is no need to anticipate which types are better expressed as main types, which as associated types.

Constraints on associated types

Concepts may include constraints on associated types.

Declaring constraints on associated types leads to problems in several languages; Java and C#, for example, require redundant constraints (for complex technical reasons [101]). As discussed above, all types in a concept are treated the same in Magnolia, and hence Magnolia supports constraints on associated types as it does for any type.

Retroactive modeling

New modeling relationships can be added after a type has been defined.

Problematic languages concerning this property are languages where the declaration that a data type satisfies a certain set of requirements (a concept or concepts) takes place at the site of definition of the data type. This is the case for object-oriented languages that fix the bases of a class when the class is defined. But even in Haskell, where an *instance declaration* is distinct from both a datatype and a type class definition, retroactive modeling can be limited. An example is changes to Haskell's standard library and its type class hierarchy. In 2014 the `Applicative` typeclass was suggested to be made a superclass of the `Monad` typeclass [2]. Such a change breaks `Monad` instances (models) where the corresponding `Functor` and `Applicative` instances are not implemented. This change occurred after the study of Garcia et al. [61], and thus likely was not considered when evaluating the support of Haskell for retroactive modeling—the study characterized Haskell as fully supporting retroactive modeling.

Listing 1.8 builds up to the concepts of a commutative magma with a left absorbing element, and of a commutative magma with a right absorbing element. The two concepts are equivalent, i.e., each concept models the other. Each modeling relationship is expressed through a satisfaction relation (see `CommutativeZeroLR` and `CommutativeZeroRL`). Satisfaction relations can be added at any point in the program, hence Magnolia satisfies the retroactive modeling property.

Type aliases

A mechanism for creating shorter names for types is provided.

Listing 1.8: An example of equivalent specifications in Magnolia.

```
concept CommutativeMagma = {
  type T;
  function bop(t1: T, t2: T): T;
  axiom commutativity(t1: T, t2: T)) {
    assert bop(t1, t2) == bop(t2, t1);
  }
}

concept CommutativeMagmaWithLeftZero = {
  use CommutativeMagma;
  function zero(): T;
  axiom leftAbsorption(t: T) {
    assert bop(zero(), t) == zero();
  }
}

concept CommutativeMagmaWithRightZero = {
  use CommutativeMagma;
  function zero(): T;
  axiom rightAbsorption(t: T) {
    assert bop(t, zero()) == zero();
  }
}

satisfaction CommutativeZeroLR =
  CommutativeMagmaWithLeftZero models
    CommutativeMagmaWithRightZero;

satisfaction CommutativeZeroRL =
  CommutativeMagmaWithRightZero models
    CommutativeMagmaWithLeftZero;
```

The problem of long names in generic programming often arise from a large number of type parameters (due to the representation of associated types as type parameters). In Magnolia, concepts are not represented by types—type names are not parameterized, their names thus stay atomic. Magnolia does support type aliases too, however, through the mechanism of renaming — see once again lines 15 and 16 in Listing 1.1. Magnolia does not allow declaring a new alias for a type (or operation) in the same module expression. That being said, it is possible to make declarations with different names in a module expression, and to later merge them using renaming, therefore "retroactively" aliasing the two declarations.

Separate compilation

Generic functions can be compiled independently of calls to them.

The motivation behind separate compilation is attaining better compilation speed by avoiding recompiling generic definitions every time their uses are compiled. All languages but C++ in Garcia et al.'s evaluation have this compilation model; in C++ the compiler generates a distinct piece of code for each different template instantiation. While Garcia et al. bundled separate compilation and modular type checking under one language property, Siek and Lumsdaine [163] split them into two distinct properties. This allows to more precisely characterize C++ after the concepts feature was added—C++ today partially supports modular type checking but not separate compilation.

In Magnolia, generic operations are type checked where they are declared. They may undergo name changes during renamings, but after these are resolved, a call to a generic function needs only to be checked against the function's declaration, so Magnolia supports modular type checking. Adhering strictly to the definition given above, Magnolia could be said to support separate compilation: each monomorphic operation is transpiled to the host language independently of calls to it (and the compilation of transpiled Magnolia code to executable code is host language-dependent). However, a distinct piece of code is emitted for each instantiation of a generic function definition — resulting in a compilation model similar to C++'s, and not one which achieves the goals of the property.

Implicit argument deduction

The arguments for the type parameters of a generic function can be deduced and do not need to be provided by the programmer. Also, the finding of models to satisfy the constraints of a generic function is automated by the language implementation.

Garcia et al. describe the lack of implicit type argument deduction to result in verbose generic algorithm invocations. Most languages avoid problems by deducing the type parameters of a generic function from the types of its function arguments. Magnolia avoids this problem in a different way: there are no implicit arguments to deduce in the first place. Operations are always monomorphic, and each argument's type is resolved whenever renaming occurs. Whenever a call occurs, there is almost always a single corresponding prototype in scope. The exception is when a call can resolve to several functions overloaded solely on their return type. In that case, a type annotation must be provided by the user to disambiguate between the matches.

Modular type checking

Generic functions can be type checked independently of calls to them.

Because of C++'s lack of modular type checking, debugging type errors in generic code in C++ is often very difficult. The C++ concepts language feature fixes this problem partially: uses of templates are checked against type parameter constraints but definitions of templates are not checked. The bodies of C++ template functions are (still) type checked only after their instantiation, which can delay catching a type errors in the implementation of a generic library until it is used in client code. Magnolia supports modular type checking of both the uses and definitions of generic code, as described above in the discussion of *Separate compilation*.

Lexically scoped models

Model declarations are treated like any other declaration, and are in scope for the remainder of the enclosing namespace. Models may be explicitly imported from other namespaces.

Siek and Lumsdaine implement lexically scoped models in *G* [163], in order to be explicit about which models are in scope. One motivation for this feature is to avoid the problem of overlapping models (corresponding to overlapping instances in Haskell, with concepts corresponding to typeclasses, models to typeclass instances). Suppose we want to define an **instance** of the `Semigroup` typeclass for `Int` in Haskell. The Haskell 2010 Language Report [1, Chapter 4] dictates that *"A type may not be declared as an instance of a particular class more than once in the program."*. However, there is more than one intuitive instance of `Semigroup` for `Int`, as shown in Listing 1.9.

Listing 1.9: Overlapping instances in Haskell.

```
-- A.hs
instance Semigroup Int where
  (<>) = (+)
-- B.hs
instance Semigroup Int where
  (<>) = (*)
-- C.hs (imports A, B)
-- error: Overlapping instances for Semigroup Int
val = (2 :: Int) <> 3
```

Attempting to call `(<>)` with both of these definitions in scope results in an error. This can be worked around in awkward ways, e.g. using **newtype**s, or wrappers around class methods [180]. The crux of the issue here is that typeclass instances are not first-class in Haskell.

It is not clear if the *lexically scoped models* property is actually sufficient to solve the problem of overlapping models. The approach works well when the different models are used in different scopes, but does not seem to offer a solution when one wants to have them in the same scope. One example of such a use case is *united monoids*, an algebraic structure involving two monoids with the same unit element [126]. Approaches such as named instances [103] or the *CONCEPT* pattern in Scala [141] can address this issue.

The issue does not arise in Magnolia either: one can bring two different models of the same concept into the same scope and resolve the overlap explicitly using the renaming mechanism—see `ConcreteSemiGroup` in Listing 1.1 for an example. The property is thus supported by design.

Concept-based overloading

There can be multiple generic functions with the same name but differing constraints. For a particular call, the most specific overload is chosen.

The C++ standard library's hierarchy of iterator concepts includes two concepts, *InputIterator* and *ForwardIterator*, whose signatures agree—they differ only on their operations' semantic requirements. In particular, the former does not admit restarting the iteration. There are well-motivated cases for

overloading a function where the overloads should be differentiated based only on whether their argument types model one or both of these concepts [168]. However, overloading on semantics is problematic in the general case. To specialize based on the semantics of two concepts with exactly the same API, the compiler needs to partially order them. Consider the two concepts `CommutativeMagmaWithRightZero` and `CommutativeMagmaWithLeftZero` from Listing 1.8. If one specializes an algorithm on their semantics, the following happens:

- if the compiler is unable to deduce that the specifications are equivalent, one specialization is picked at each call site and compilation succeeds;
- if the compiler is able to deduce that the specifications are equivalent, the compiler cannot specialize and compilation fails.

This implies that once correct code may become incorrect as the compiler's reasoning abilities get more powerful and it can deduce more properties from the same axioms. And indeed, C++ does not really do overloading on semantics. It equips each iterator concept with a *tag*-type, creating thus a syntactic difference between the two concepts' requirements, which is really what is used as a criterion in overload resolution.

Magnolia does not provide support for concept-based overloading. Arguments to an operation are always instances of the exact types specified in the operation's prototype. In the absence of subtyping, classic overloading is sufficient to resolve every call to the right implementation. The modular structure of Magnolia code allows the programmer to defer the implementation of types and operations as long as is necessary to sufficiently refine their semantic requirements and explicitly determine which implementation should be chosen.

Same-type constraints

The notion of *same-type constraints* lacks a precise definition in Siek and Lumsdaine's work. We give it the following definition: *It is possible to force two type parameters to refer to the same type*.

In Magnolia, two types are the same if they have the same name. The renaming mechanism allows forcing different concepts to depend on the same type, by bringing them into the same scope and renaming their type parameters to the same name. This is slightly different from a type constraint: instead of requiring two constrained type parameters, the resulting module has a single type parameter. This mechanism is demonstrated lines 15 and 16 in Listing 1.1: the `Magma` module is brought into scope twice, and its type parameter `T` is forced to the same name `int`.

First-class functions

Supporting anonymous functions with lexical scoping as first-class citizens of the language.

Magnolia does not support higher-order functions. This is intentional: it keeps Magnolia programs simpler to reason about. We do not lose out on expressivity—in lieu of higher-order functions, the Magnolia programmer can use a parameterized module [64], and deal with potential naming conflicts by leveraging once more the renaming mechanism. This is, however, certainly a trade-off on convenience. For example, the most cumbersome aspects of implementing the graph library were

the looping structures, split into several concepts and functions (see e.g. Listings 1.7 and 7), which with higher-order functions could have been implemented with a single function parameterized by a function parameter.

Property-based specifications

We define the property as follows: *Arbitrary semantic constraints on types and operations can be defined.*

Property-based specifications are a desirable feature that can enable strong correctness guarantees and formal verification of code. Such semantic constraints are mentioned in Garcia et al.'s study [61], but they are not evaluated due to the lack of support for them in the studied languages. The way we specify properties in Magnolia axioms is through assertions—and in fact, any programming language with assertions can produce some sort of library support for property-based specifications. We explicitly do not consider this to qualify as language support for property-based specifications in Figure 1.2, but we mention some such libraries below.

C++20 implements a scaled back version of C++ox's concepts which does not provide support for semantic constraints—but only for same-type constraints and API modeling constraints. Bagge et al. previously built a testing system atop concepts and axioms implemented using template metaprogramming in C++11 [20].

SML does not support property-based specifications. We note that property-based specifications for SML programs can be expressed in Sannella and Tarlecki's Extended ML [152]. However, the semantics of Standard ML are not fully compatible with the theory of algebraic specifications, and the approach suffers from a semantic gap common in many approaches to formal verification of software [153].

OCaml does not support property-based specifications. Xu showed how OCaml could be augmented with a contract declaration construct, along with both static and dynamic contract checking features [178]. However, to the best of the authors' knowledge, this research did not lead to the implementation of such a feature in OCaml. Design by contract (DbC) is a common approach to software correctness made popular by Eiffel [124; 125]. DbC has roots in Floyd-Hoare logic [55; 94] and uses assertions to specify *preconditions*, *postconditions*, and *invariants* on programs. Bagge et al. point out limitations with pre/postconditions for specifying generic APIs, e.g., difficulties of capturing properties like *associativity* or *transitivity*, and show how they are subsumed by axioms [19].

Haskell's support for property-based specifications is limited. One visible consequence of this is that typeclass laws are typically stated only as documentation, and it is up to the programmer of a typeclass instance to ensure that they hold. However, the language's powerful type system and extensions allow specifying and enforcing sophisticated invariants. For example, Bailey and Gale encoded the full FIDE ruleset at the type level [21]. Noonan shows a design concept for validating preconditions at compile time by constructing proofs inhabiting phantom type parameters [139]. Haskell also offers good support for property-based testing, through the QuickCheck library [40]. Like for OCaml, some work on enabling static contract checking in Haskell was initiated, but did not lead to the implementation of such a feature in the language to the best of the authors' knowledge [179]. Also worthy of note is LiquidHaskell, a static verifier for Haskell based on liquid types [151]. Liquid types are refinement types [57] with logical predicates coming from a decidable sublanguage—allowing decidable type checking and inference.

1

Java itself does not support property-based specifications. However, there exist a number of tools extending Java to provide varying levels of support for property-based specifications. One such tool is the Java Modeling Language (JML), which draws from the design by contract approach and from algebraic specifications to allow for specifying the behavior of Java modules [116]. Another one is JAxT, a tool that generates JUnit test cases from static methods representing axioms [85; 86]. In addition, there exists several libraries implementing property-based testing à la QuickCheck for Java—one example is junit-quickcheck [95].

C# itself does not provide support for property-based specifications. Similarly to Java, several tools exist that address this shortcoming. For example, Spec# is a superset of C# which allows specifying and verifying method contracts (pre- and postconditions), object invariants, and loop invariants [22]. Code contracts are another approach that enables design-by-contract programming in .NET programming languages [53].

Cecil does not provide support for property-based specifications.

Axioms were supported in **C++ox** concepts, and the language had full support for property-based specifications.

Siek and Lumsdaine's $\mathcal{G}$ [163] does not implement such semantic constraints, restricting itself to same-type constraints and API modeling constraints (similarly to the concepts implemented in C++20).

Magnolia supports property-based specifications in the form of axioms in concepts. These formulas are boolean expressions with free variables, thus encompassing equational and conditional equational specifications as common in algebraic specifications [26]. Boolean expressions have the benefit of being readily handled by programming language compilers and tools, allowing us to compile axioms as test oracles and systematically test a program's compliance with its specification [20]. The axiom formalism and the program code are semantically compatible, thus avoiding the semantic gap mentioned previously [153]. Magnolia axioms can be leveraged in practice for program optimizations [39] and for proving the correctness of Magnolia specifications [83].

Variadics

We define the property as follows: *Operations can have a variable number of arguments of different types.*

C++ and C++ox allow generic operations to take in a variable number of arguments of different types through variadic templates [75]. In a variadic context, any type expression can be repeated, including expressions containing the *const* qualifier, the lvalue and rvalue reference declarators, or any *concept* constraint. We note that variadic templates were introduced in C++11, a version of the language that postdates both Garcia et al.'s and Siek and Lumsdaine's studies. The version of C++ evaluated in the previous studies did not support variadics, but C++ox did.

It is possible to implement functions that support a variable number of arguments of different types in **OCaml**, as demonstrated in the current implementation of the *Format* module [174]. This implementation of variadics relies on heterogeneous lists, which are in turn implemented in OCaml with difference lists leveraging Generalized Algebraic Data Types (GADTs) [148; 154]. There are limitations to this approach, related to the mixing of GADTs and subtyping [148; 155].

Like for OCaml, there is no obvious way to define functions that take a variable number of arguments of different types in **Haskell** [92]. It is however possible to define variadic functions through type hackery, as demonstrated by the HList library [106; 107]. Haskell's expressivity allows for several reasonable ways to create such functions—as noted in the source code of HList [108]. Since 2015, heterogeneous lists are implemented using a *data family* [156].

Support for variadics in **Java** is only partial. It is possible to define operations that take a variable number of arguments of different, arbitrary types in Java by adding an argument of type `Object...` to the end of their argument list. The ellipsis syntax is syntactic sugar for passing in a single-dimensional array of the specified type as an argument. `Object` is a superclass for every defined class in Java, ensuring that the function can be called with parameters of any object type. Note that this excludes primitive types, which are not subclasses of `Object`, and for which there is no obvious solution.

Support for variadics in **C#** is limited. It is possible to define functions that take a variable number of arguments of different types in C# by using the `params` keyword to pass in an arbitrary number of arguments of type either `object` or `dynamic`. The `params` keyword is syntactic sugar for passing in an array as a parameter. Arguments given the `dynamic` type can not be type checked at compile time, and will cause run time exceptions if used inappropriately. Arguments converted to the `object` type must eventually be unboxed to the correct type to be used. In this case, some errors can be caught at compile time, and others at run time. It is also not possible to pass a variable number of generic type parameters to a function.

None of **SML**, **Cecil**, and *G* [159; 163] (to the best of our knowledge) offer support for variadics.

Because all types are opaque in **Magnolia**, data structures are characterized only on the set of externally-implemented functions that construct or consume them. To define a record-like type with n fields in Magnolia requires one type definition, along with $2n + 1$ function definitions (one projection and one update for each field, and a constructor). This quickly leads to a large number of functions. These projections and updates may also be expensive, as discussed in Subsection 1.3.3; there, we solved both problems by defining several loop concepts and backend implementations, each with a carefully chosen number of state and context types and parameters. This solution has its own drawbacks though: concepts and implementations are (mostly) duplicated, including axioms. Adding support for variadics to Magnolia would achieve the same outcomes, while eliminating the need for code duplication. We briefly discuss an approach for supporting variadics in Magnolia in Section 1.6.

1.5 Performance

Another key idea in generic programming is that abstracting an algorithm should have no impact on performance: when a generic algorithm is specialized to the concrete case, it should be just as efficient as if the algorithm had been written directly as the non-generic case. We tested whether Magnolia and its BFS implementation satisfy this criterion. Figure 1.3 compares the performance of two instantiations of our BFS implementations; for both, C++ was the host language. The figure also shows the performance of BGL's BFS implementation.

The two Magnolia implementations use the same generic algorithm with different backend data structures. The red bars show the performance of an instantiation that uses the same data structures as the BGL's algorithm (blue bars). The yellow bars show an instantiation of the Magnolia code that

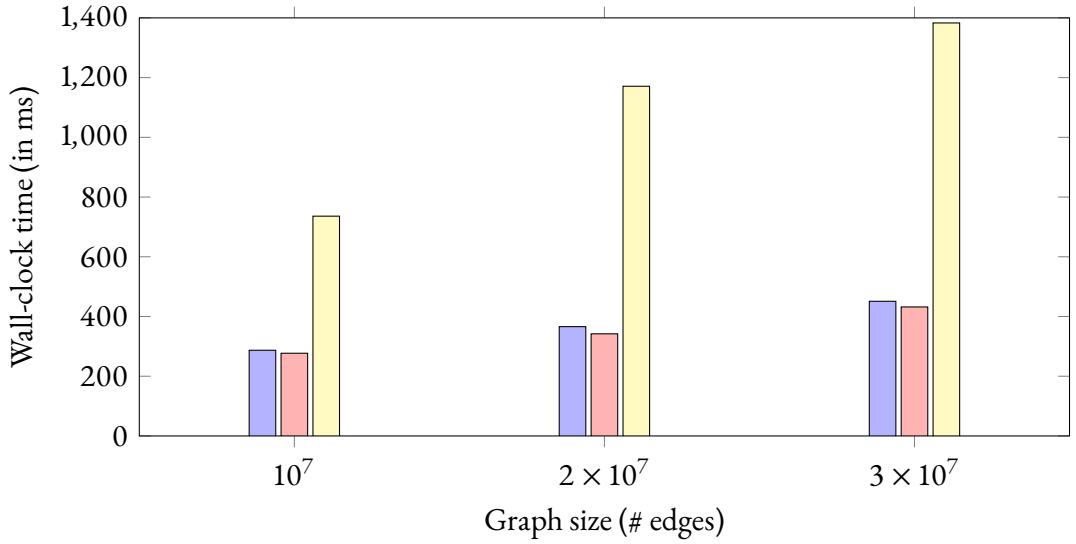

Figure 1.3: Performance comparison of running the sequential version of the BFS algorithm: the leftmost columns (blue) are for BGL's C++ implementation, the middle (red) ones for the Magnolia implementation where the base library uses the same data structures as BGL, and the rightmost (yellow) ones for the Magnolia implementation that moves the language-library border further towards the language, providing highly parameterizable data structures. Both Magnolia implementations used C++ as the host language. Each implementation is run 10 times in total, and the running times are averaged. Every implementation is tested against the same 10 randomly generated graphs. Each graph is directed and contains 10^6 vertices. The test programs are compiled using g++ 10.2.0, with optimization level O3, on an Intel(R) Xeon(R) Silver 4112 CPU @ 2.60GHz.

uses our own ad-hoc, prototype data structures. The transpiled algorithms are identical to the one implemented in the BGL. When using the same underlying data structures, the Magnolia and BGL C++ implementations perform equally well, showing that our generic abstraction in Magnolia is indeed cost-free.

Instantiating the algorithm with our prototype data structures produces code that runs roughly 2.5–3.5 times slower, depending on the number of edges in the graph. At the same time, these data structures offer more flexibility to the user by virtue of being more parameterizable—highlighting a trade-off between parameterization and performance here. Magnolia allows fine-grained choice of level of abstraction on the host language. This makes exploring the possible combinations of backend data structures easy. No particular effort was put in tuning our prototype data structures for performance: it is entirely possible that a more careful and as parameterizable design could match the BGL implementation's performance.

The transpilation of our Magnolia code to a host language does not add much overhead: it takes less than a second to transpile the whole fragment of the BGL we implemented. In contrast, compiling the final binary from the C++ code which imports the BGL takes more than seventeen seconds[3].

We did not run performance tests with Python as the host language. We can expect the current implementation to be slow, because we left overload resolution to be performed in Python (out of convenience), using multiple dispatch.

[3]Compilation times reported for an Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz.

1.6 Discussion and Conclusion

Garcia et al.'s study [61] paved the way for evaluating support for generic programming of languages. The properties the authors identify as important point out issues of retrofitting generic programming into existing languages. This in fact led language implementors to address these issues and improve their language's support for generic programming [31; 32]. Siek and Lumsdaine's *G* demonstrates how a language based on these properties enables generic programming. Our work takes a step back from these works and looks at generic programming from the angle of algebraic specifications, repeating the experiment [61] with Magnolia—a language representative of the algebraic approach. Magnolia is not shoehorned into the properties that Garcia et al. identified, yet provides extensive support for generic programming.

Our evaluation in Section 1.4 shows that the renaming mechanism plays a crucial role in enabling generic programming in Magnolia. Renaming is Magnolia's pragmatic version of signature morphisms. It allows control over the naming of types and operations, both to keep them separate as needed for implementations, but also to coordinate naming within concepts when joining them together. This is somewhat less powerful than the signature morphisms supported by CASL [26], yet powerful enough to enable a high level of reuse between modules. Carette et al. recently investigated union and renaming as a reuse mechanism for modular specification of mathematical concepts [30].

Every programming language is its own formal system with its own advantages and inconvenients, and Magnolia is no exception. For example, while the algebraic approach gives extreme flexibility when it comes to parameterizing and combining modules, this flexibility comes at a usability cost: when developing in Magnolia, it is hard to keep track of what is in scope at a given line and where declarations come from. The problem is further exacerbated by the renaming mechanism. Tool support (e.g., in the form of an IDE) is crucial for Magnolia development. Bagge described an implementation of an IDE for Magnolia integrated with Eclipse [15]. The newer magnoliac compiler provides a basic interactive toplevel that allows users to inspect the content of loaded modules [35]. The design and development of a fully-fledged IDE for Magnolia will inform on whether the reasoning problems we faced when implementing the BGL in Magnolia can be mitigated, and is a topic of future work for us.

In Gibbons' taxonomy of generic programming [62] we characterized Magnolia as supporting genericity by property. This axis of genericity has been an inspiration of new features for C++ (concepts), and there has been expectations that proper language support for expressing semantic properties (axioms in concepts) will lead to domain-specific optimization opportunities, more precise static checking of code for semantic errors, and more flexible (concept-based) overloading. This experiment with Magnolia accentuates some challenges that will remain, even with full language support for properties. In particular, in our evaluation we discuss concept-based overloading and why overloading based on semantic properties is problematic. Further, there are challenges with the expression of semantic constraints in concepts. Sometimes axioms are not expressible by using solely the operations that a concept is meant to expose—additional operations need to be added to the concept just to be able to express a semantic property [24]. Listing 8 gives an example: given `g: Graph`, `vertices(g)` returns the collection of all vertices in `g`. Given `v: Vertex` a vertex of `g`, `adjacentVertices(g, v)` returns the collection of all the vertices adjacent to `v` in `g`. There is a subset relation between `adjacentVertices(g, v)` and `vertices(g)`. To state this property through an axiom, we would need additional operations on `VertexCollection`, e.g., the ability to check whether a `Vertex` is a member of a `VertexCollection`. These operations and the axiom are shown as commented out.

1

Another challenge we identified with Magnolia is the inability to express variadic generic definitions. The general mechanism of syntactic theory functors [87] seems well-suited for implementing variadics in Magnolia. In fact, also renaming can be expressed as STFs. These connections are topics for future work for us.

Acknowledgements We offer our thanks to the anonymous reviewers for their thoughtful insights on our paper. We also thank Mikhail Barash for proofreading, and providing insightful comments about our manuscript in early stages.

Paper 2

Finite Difference Methods Fengshui: Alignment through a Mathematics of Arrays[1]

Benjamin Chetioui[a], Lenore Mullin[b], Ole Abusdal[a], Magne Haveraaen[a], Jaakko Järvi[a], Sandra Macià[c]

[a]Department of Informatics, University of Bergen, Norway, [b]College of Engineering and Applied Sciences, University at Albany, SUNY, USA, [c]Barcelona Supercomputing Center, Spain

Abstract

Numerous scientific-computational domains make use of array data. The core computing of the numerical methods and the algorithms involved is related to multi-dimensional array manipulation. Memory layout and the access patterns of that data are crucial to the optimal performance of the array-based computations. As we move towards exascale computing, writing portable code for efficient data parallel computations is increasingly requiring an abstract productive working environment. To that end, we present the design of a framework for optimizing scientific array-based computations, building a case study for a Partial Differential Equations solver. By embedding the Mathematics of Arrays formalism in the Magnolia programming language, we assemble a software stack capable of abstracting the continuous high-level application layer from the discrete formulation of the collective array-based numerical methods and algorithms and the final detailed low-level code. The case study lays the groundwork for achieving optimized memory layout and efficient computations while preserving a stable abstraction layer independent of underlying algorithms and changes in the architecture.

[1]Benjamin Chetioui, Lenore Mullin, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Sandra Macià. Finite difference methods fengshui: Alignment through a Mathematics of Arrays. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, page 2–13, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367172. doi:10.1145/3315454.3329954

2.1 Introduction

Given an address space, the data layout and the pattern of accessing that data are fundamental for the efficient exploitation of the underlying computer architecture. The access pattern is determined by a numerical algorithm, which may have been tuned to produce a particular pattern. The data layout may have to be adjusted explicitly to a given pattern and the computer hardware architecture. At the same time, high-performance environments are evolving rapidly and are subject to many changes. Moreover, numerical methods and algorithms are traditionally embedded in the application, forcing rewrites at every change. Thus the efficiency and portability of applications are becoming problematic. Under this scenario, software or hardware modifications usually lead to a tedious work of rewriting and tuning throughout which one must ensure correctness and efficiency. To face this scenario, the scientific community suggests a separation of concerns through high-level abstraction layers.

Burrows et al. identified a Multiarray API for Finite Difference Method (FDM) solvers [28]. We investigate the fragment of the Mathematics of Arrays (MoA) formalism [127; 130] that corresponds to this API. MoA gives us the ψ-calculus for optimizing such solvers. We present a full system approach from high level coordinate-free Partial Differential Equations (PDEs) to preparing for the layout of data and code optimization, using the MoA as an intermediate layer and the Magnolia programming language [14] to explore the specifications. In this framework, a clean and natural separation occurs between application code, the optimization algorithm and the underlying hardware architecture, while providing verifiable components. We fully work out a specific test case that demonstrates an automatable way to optimize the data layout and access patterns for a given architecture in the case of FDM solvers for PDE systems. We then proceed to show that our chosen fragment of the rewriting system defined by the ψ-calculus makes up a canonical rewriting subsystem, i.e. one that is both strongly normalizing and confluent.

In the proposed system, algorithms are written against a stable abstraction layer, independent of the underlying numerical methods and changes in the architecture. Tuning for performance is still necessary for the efficient exploitation of different computer architectures, but it takes place below this abstraction layer without disturbing the high-level implementation of the algorithms.

This paper is structured as follows. Section 2.2 presents the related work, and a concise literature review of the state of the art. Section 2.3 introduces the general software stack composition and design used for our purposes. Section 2.4 details the optimizations and transformation rules. The PDE solver test case showcasing the framework is presented in Section 2.5. Finally, conclusions are given in Section 2.6.

2.2 Related Work

Whole-array operations were introduced by Ken Iverson [100] in the APL programming language, an implementation of his notation to model an idealized programming language with a universal algebra. Ten years later, shapes were introduced to index these operations by Abrams [4]. Attempts to compile and verify APL proved unsuccessful due to numerous anomalies in the algebra [172]. Specifically, $\iota\sigma$ was equivalent to $\iota\langle\ \sigma\ \rangle$, where σ is a scalar and $\langle\ \sigma\ \rangle$ is a one element vector. Moreover, there was no indexing function nor the ability to obtain all indices from an array's shape. This caused Perlis to conclude the idealized algebra should be a Functional Array Calculator based on the λ-calculus [172].

Even with this, no calculus of indexing was formulated until the introduction of MoA [127]. MoA can serve as a foundation for array/tensor operations and their optimization.

Numerous languages emerged with monolithic or whole-array operations. Some were interpreted (e.g. Matlab and Python), some were compiled (e.g. Fortran90 and TACO [109]) and some were Object Oriented with pre-processing capabilities (e.g. C++ with expression templates [47; 161]). Current tensor (array) frameworks in contemporary languages, such as Tensorflow [3] and Tensor Toolbox [12] provide powerful environments to model tensor computations. None of these frameworks are based on the ψ-calculus.

Existing compilers have various optimizations that can be formulated in the ψ-calculus, e.g. loop fusion (equivalent to distributing indexing of scalar operations in MoA) and loop unrolling (equivalent to collapsing indexing based on the properties of ψ and the ψ-correspondence Theorem (PCT) in MoA [130]). Many of the languages mentioned above implement concepts somewhat corresponding to MoA's concept of shape and its indexing mechanism. It is, however, the properties of the ψ-calculus and its ability to obtain a Denotational Normal Form (DNF) for any computation that make it particularly well-suited for optimization.

Hagedorn et al. [81] pursued the goal of optimizing stencil computations using rewriting rules in LIFT.

2.3 Background, Design and Technologies

We present the design of our library-based approach structured by layers. Figure 2.1 illustrates this abstract generic environment. At the domain abstraction layer, code is written in the integrated

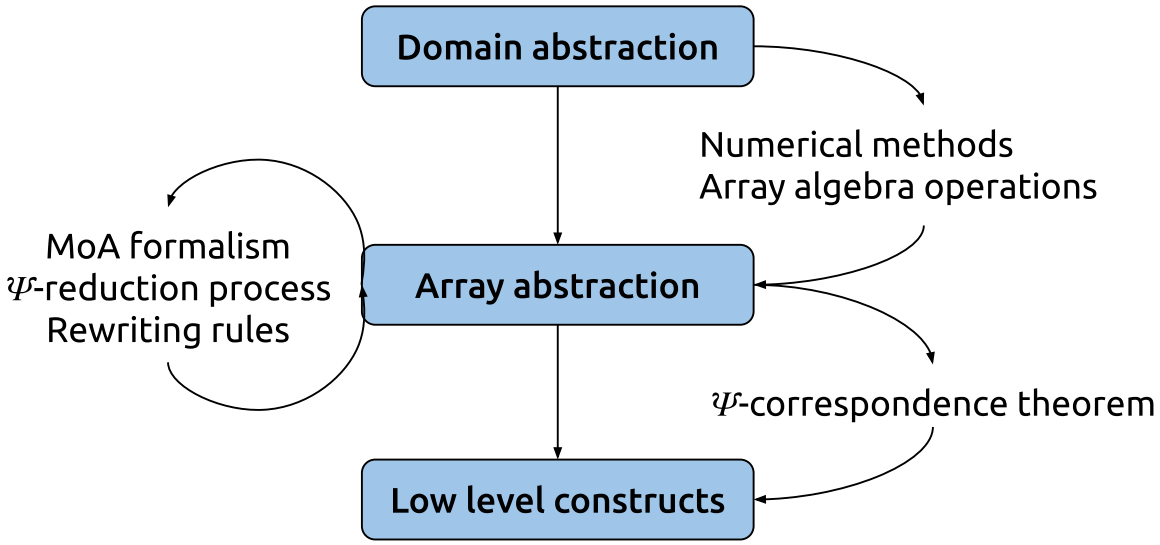

Figure 2.1: Layer abstraction design; generic environment approach.

specification and programming language Magnolia, a language designed to support a high level of abstraction, ease of reasoning, and robustness. At the intermediate level, the MoA formalism describes

multi-dimensional arrays. Finally, through the ψ-correspondence theorem, the array abstraction layer is mapped to the final low-level code.

2.3.1 Magnolia

Magnolia is a programming language geared towards the exploration of algebraic specifications. It is being designed at the Bergen Language Design Laboratory [14]; it is a work in progress and is used to teach the Program Specification class at the University of Bergen, Norway. Magnolia's strength relies in its straightforward way of working with abstract constructs.

Magnolia relies primarily on the *concept* module, which is a list of type and function declarations (commonly called a *signature*) constrained by *axioms*. In Magnolia, an *axiom* defines properties that are assumed to hold; it however differs from the usual axioms in mathematics in that an axiom in Magnolia may define derived properties. Functions and axioms may be given a *guard*, which defines a precondition. The *satisfaction* module serves to augment our knowledge with properties that can be deduced from the premises, typically formatted to indicate that a *concept* models another one.

Magnolia is unusual as a programming language in that it does not have any built-in type or operation, requiring that everything be defined explicitly. Magnolia is transpiled to other languages, and thus, the actual types the programmer intends to use when running their program must be defined in the target language.

2.3.2 Mathematics of Arrays

MoA [127; 130] is an algebra for representing and describing operations on arrays. The main feature of the MoA formalism is the distinction between the *DNF*, which describes an array by its shape together with a function that defines the value at every index, and the *Operational Normal Form* (ONF), which describes it on the level of memory layout. The MoA's ψ-calculus [130] provides a formalism for index manipulation within an array, as well as techniques to reduce expressions of array operations to the DNF and then transform them to ONF.

The ψ-calculus is based on a generalized array indexing function, ψ, which selects a partition of an array by a multidimensional index. Because all the array operations in the MoA algebra are defined using shapes, represented as a list of sizes, and ψ, the reduction semantics of ψ-calculus allow us to reduce complex array computations to basic indexing/selection operations, which reduces the need for any intermediate values.

By the ψ-correspondence theorem [130], we are able to transform an expression in DNF to its equivalent ONF, which describes the result in terms of loops and controls, *starts*, *strides* and *lengths* dependent on the chosen linear arrangement of the items, e.g. based on hardware requirements.

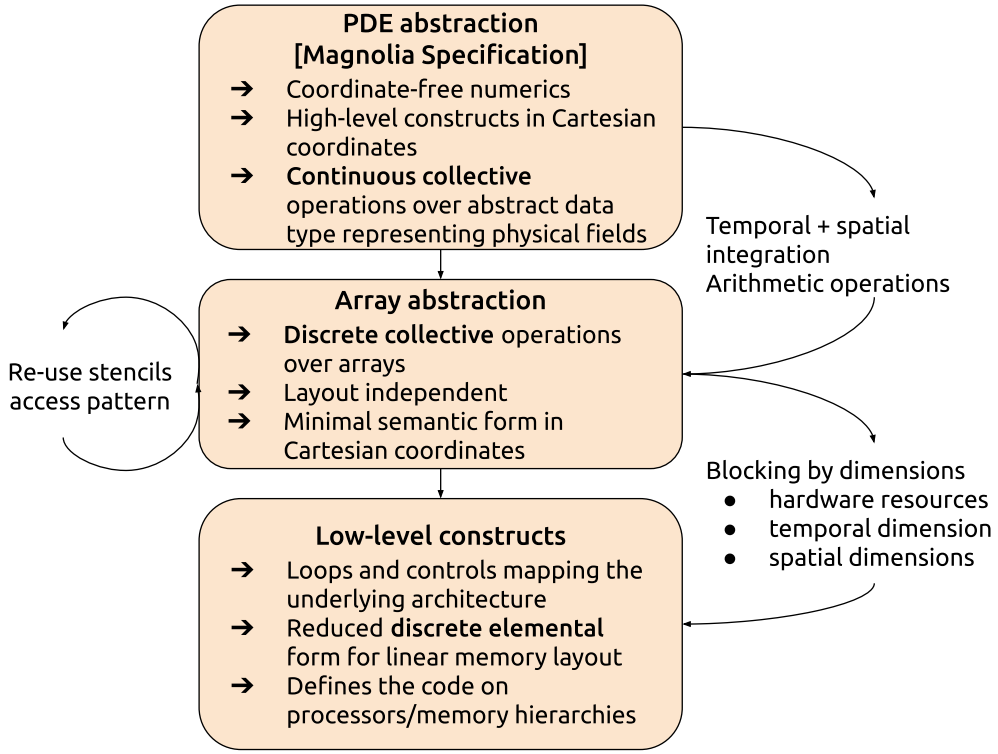

Figure 2.2: Layer abstraction design; detailed environment designed for a PDE solver.

2.3.2.1 Motivation behind DNF and ONF

The goal behind the DNF and the ONF is to create an idealized foundation to define most — if not all — domains that use tensors (arrays). Using MoA, all of the transformations to the DNF can be derived from the definition of the ψ function and shapes.

This view has a long history [4] and, when augmented by the λ-calculus [25], provides an idealized semantic core for all arrays [128; 133]. Array computations are very prevalent. A recent Dagstuhl workshop [7; 8] reported the pervasiveness of tensors in the Internet of things, Machine Learning, and Artificial Intelligence (e.g. Kronecker [131]) and Matrix Products [78]. Moreover, they dominate science [79; 113] in general, especially signal processing [129; 135; 136; 144] and communications [134].

2.3.3 PDE Solver Framework

Figure 2.2 illustrates the design structured by layers for the PDE solver framework we describe. The first abstraction layer defines the problem through the domain's concepts. At this level, PDEs are expressed using collective and continuous operations to relate the physical fields involved. Through the functions encapsulating the numerical methods, the high-level continuous abstraction is mapped to a discrete array-based layer. A Magnolia specification of the array algebra defined by the MoA formalism and the ψ-calculus has been developed at this level. This algebra for arithmetic operations and permutations over multi-dimensional arrays defines the problem through collective array operations in a layout independent manner. At this point, array manipulation functions and operations may be defined in the MoA formalism and reduced according to the ψ-reduction process. This process simplifies an expression through transformational and compositional reduction properties: the rewriting rules.

From the user's array-abstracted expression we obtain an equivalent optimal and minimal semantic form. Finally, the indexing algebra of the ψ-calculus relates the monolithic operations to elemental operations, defining the code on processors and memory hierarchies through loops and controls. The ψ-correspondence theorem is the theorem defining the mapping from the high-level abstracted array expressions to the operational expressions, i.e. from a form involving Cartesian coordinates into one involving linear arranged memory accesses.

2.4 MoA Transformation Rules

2.4.1 ψ-Calculus and Reduction to DNF

Multiarrays, or multidimensional arrays, have a shape given by a list of sizes $\langle s_0 \ \ldots \ s_{n-1} \rangle$. For example, a 6 by 8 matrix A has the shape $\langle 6\ 8 \rangle$. The index for a multiarray is given by a multi-index $\langle i_0 \ \ldots \ i_{n-1} \rangle$. For position j of the multi-index, the index i_j is in the range $0 \leq i_j < s_j$. This sets the vocabulary for talking about multiarrays. In the following Magnolia code and in the rest of the paper, we will assume that the following types are declared:

- **type** `MA`, for Multiarrays;
- **type** `MS`, for Multishapes;
- **type** `MI`, for Multi-indexes;
- **type** `Int`, for Integers.

All these types will have (mapped) arithmetic operators. Important functions on a multiarray are:

- the shape function ρ, which returns the shape of a multiarray, e.g. $\rho A = \langle 6\ 8 \rangle$;
- the ψ function, which takes a submulti-index and returns a submultiarray, e.g. $\langle \rangle \ \psi A = A$ and $\rho(\langle 3 \rangle \ \psi \ A) = \langle 8 \rangle$ is the subarray at position 3;
- the rotate function θ, which rotates the multiarray: $p \ \theta_x \ A$ denotes the rotation of A by offset p along axis x (rotate does not change the shape: $\rho(p \ \theta_x \ A) = \rho A$).

With respect to ψ, rotate works as:

$$\langle i_0 \ \ldots \ i_x \rangle \ \psi \ (p \ \theta_0 \ A) \quad = \quad \langle (i_0 + p) \bmod s_0 \ \ldots \ i_x \rangle \ \psi \ A$$

The rotate operation can be used to calculate, for each element, the sum of the elements in the adjacent columns, $(1 \ \theta_0 \ A) + ((-1) \ \theta_0 \ A)$, which is a multiarray with the same shape as A. Applying ψ to the expression gives the following reduction:

$$\langle i_0 \rangle \ \psi \ ((1 \ \theta_0 \ A) + ((-1) \ \theta_0 \ A)) = \langle (i_0 + 1) \bmod s_0 \rangle \ \psi \ A + \langle (i_0 - 1) \bmod s_0 \rangle \ \psi \ A$$

These above MoA functions can be declared in Magnolia, with axioms stating their properties.

```
/** Extract the shape of an array. */
function rho(a:MA) : MS;
```

```
/** Extract subarray of an array. */
function psi(a:MA, mi:MI) : MA;

/** Rotate distance p along axis. */
function rotate(a:MA, axis:Int, p:Int) : MA ;

axiom rotateShape(a:MA, ax:Int, p:Int) {
  var ra = rotate(a,ax,p);
  assert rho(ra) == rho(a);
}

axiom rotatePsi(a:MA, ax:Int, p:Int, mi:MI) {
  var ra = rotate(a,ax,p);
  var ij = pmod(get(mi,ax)+p,get(rho(a),ax));
  var mj = change(mi,ax,ij);
  assert psi(ra,mi) == psi(a,mj);
}

axiom plusPsi(a:MA, b:MA, mi:MI)
  guard rho(a) == rho(b) {
  assert rho(a+b) == rho(a);
  assert psi(a+b,mi) == psi(a,mi) + psi(b,mi);
}
```

Note how we are using ρ and ψ to define operations on multiarrays. The ρ operator keeps track of the resulting shape. The ψ operator takes a partial multi-index and explains the effect of the operation on the subarrays. In this way the ψ operator moves inward in the expression, pushing the computation outwards towards subarrays and eventually to the element level.

The concatenation property for ψ-indexing is important for this,

$$\langle j \rangle \ \psi \ (\langle i \rangle \ \psi \ A) \equiv \langle i\, j \rangle \ \psi \ A.$$

```
axiom psiConcatenation(ma:MA, q:MI, r:MI) {
  var psiComp = psi( psi( ma,q ), r );
  var psiCat  = psi( ma, cat( q,r ) );
  assert psiComp == psiCat;
}
```

The rules above, for rotation and arithmetic, show how ψ moves inwards towards the multiarray variables. When this process stops, we have reached the DNF. All other multiarray functions have then been removed and replaced by their ψ definitions. What is left to figure out and what we will tentatively show in this paper is how to build the DNF.

Burrows et al [28] made the case that the operations defined above augmented with mapped arithmetic constitute a sufficient basis to work with any FDM solver of PDE systems. It does not matter what language the original expression comes from (Python, Matlab, Fortran, C, etc). With the syntax removed and the tokens expressed as an AST, the DNF denotes the reduced semantic tree and could be returned to the syntax of the originating language, with interpretation or compilation proceeding as usual.

2.4.2 Transformation Rules

The MoA defines many rewriting rules in order to reduce an expression to its DNF. Working with those, we got the insight that the goal of the reduction is to move the call to ψ inwards to apply it as early as possible in order to save computations, and that there are enough rules to allow us to move ψ across any type of operation (Multiarray on Multiarray, scalar on Multiarray).

For the sake of this particular example, we limited ourselves to a subset of the transformation rules in the MoA. We show that this constitutes a rewriting system that is canonical.

Let us first introduce the rules we are using. In the rules, the metavariables $\texttt{index}_i$, u_i and $\texttt{sc}_i$ respectively denote multi-indexes, multiarrays and scalars. The metavariable op is used for mappable binary operations such as $\times$, $+$ and $-$, that take either a scalar and a multiarray or two multiarrays as parameters and return a multiarray.

$$\frac{\texttt{index}\ \psi\ (u_i\ \texttt{op}\ u_j)}{(\texttt{index}\ \psi\ u_i)\ \texttt{op}\ (\texttt{index}\ \psi\ u_j)}\ \text{R1}$$

$$\frac{\texttt{index}\ \psi\ (\texttt{sc}\ \texttt{op}\ u)}{\texttt{sc}\ \texttt{op}\ (\texttt{index}\ \psi\ u)}\ \text{R2}$$

$$\frac{i \leq k \implies \langle \texttt{sc}_0\ \ldots\ \texttt{sc}_i\ \ldots\ \texttt{sc}_k \rangle\ \psi\ (\texttt{sc}\ \theta_i\ u)}{\langle \texttt{sc}_0\ \ldots\ ((\texttt{sc}_i + \texttt{sc})\ \bmod\ (\rho u)[i])\ \ldots\ \texttt{sc}_k \rangle\ \psi\ u}\ \text{R3}$$

Proving that a rewriting system is canonical requires proving two properties [112]:

1. the rewriting system must be confluent;
2. the rewriting system must be strongly normalizing (reducible in a finite number of steps).

For a rewriting system, being confluent is the same as having the Church-Rosser property [112], i.e. in the case when reduction rules overlap so that a term can be rewritten in more than one way, the result of applying any of the overlapping rules can be further reduced to the same result. If a term can be derived into two different terms, the pair of the two derived terms is called a critical pair. Proving that a rewriting system is confluent is equivalent to proving that every critical pair of the system yields the same result for both of its terms.

Our rules above of the rewriting system can not generate any critical pair; the system is thus trivially confluent.

Now, we must prove that the rewriting system is strongly normalizing: the system must yield an irreducible expression in a finite number of steps for any expression. To that end, we assign a weight $w \in \mathbb{N}$ to the expression such that w represents the "weight" of the expression tree. We define the weight of the tree as the sum of the weight of each $(\texttt{index}\ \psi)$ node. The weight of each one of these nodes is equal to 3^h, where h is the height of the node.

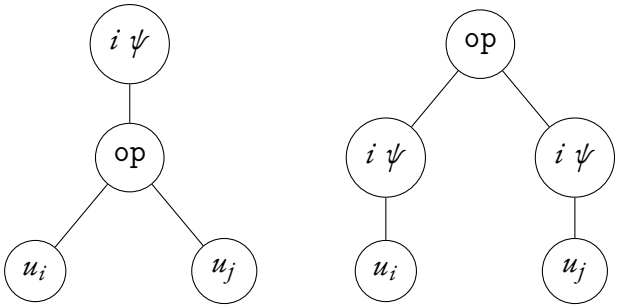

Figure 2.3: Rule 1 and its application.

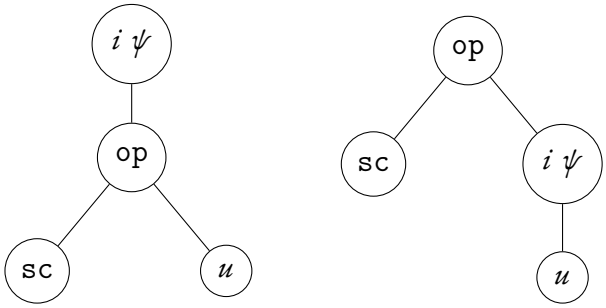

Figure 2.4: Rule 2 and its application.

Since $\mathbb{N}$ is bounded below by 0, we simply need to prove that the application of each rule results in w strictly decreasing to prove that our rewriting system is strongly normalizing.

For each one of our three rules, we draw a pair of trees representing the corresponding starting expression on the left and the resulting expression from applying the rule on the right. Then, we verify that w strictly decreases from the tree on the left to the tree on the right. We call w_l the weight of the left tree and w_r the weight of the right tree. Figures 2.3, 2.4 and 2.5 illustrate these trees.

In the three figures, we assume that the tree rooted in the $i\ \psi$ node has height h'. Since the $i\ \psi$ node has a parameter, it is never a leaf and we have $h' > 0$.

In Figure 2.3, the starting expression has the weight $w_l = 3^{h'}$. The resulting expression from applying $R1$, however, has the weight $w_r = 2 \times 3^{h'-1} = \frac{2}{3} w_l$, which is less than w_l. In Figure 2.4, the starting expression has the weight $w_l = 3^{h'}$. The resulting expression from applying $R2$, however, has the weight $w_r = 3^{h'-1} = \frac{1}{3} w_l$, which is less than w_l. In Figure 2.5, the starting expression has the weight $w_l = 3^{h'}$. The resulting expression from applying $R3$, however, has the weight $w_r = 3^{h'-1} = \frac{1}{3} w_l$, which is less than w_l.

Since w strictly decreases with every rewrite, the system is strongly normalizing. Since it is also confluent, it is canonical.

2.4.3 Adapting to Hardware Architecture using ONF

Once we have reduced an expression to its DNF, if we know about the layout of the data it uses, we can build its ONF. Assuming a row major layout, let us turn $\langle i \rangle\ \psi\ (1\ \theta_0\ A) + ((-1)\ \theta_0\ A)$ into its ONF.

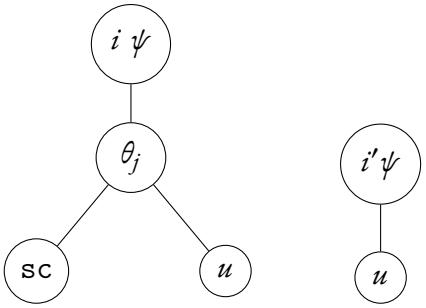

Figure 2.5: Rule 3 and its application.

To proceed further, we need to define three functions: γ, ι and rav.

- rav is short for Ravel, which denotes the *flattening* operation, both in APL and in MoA. It takes a multiarray and reshapes it into a vector. We therefore use rav to deal with the representation of the array in the memory of the computer.
- γ takes an index and a shape and returns the corresponding index in the flattened representation of the array[2]. γ is not computable unless a specific memory layout is assumed, which is why this decision has to be taken before building the ONF.

  One can note that rav and γ are tightly connected in defining flattened array accesses as γ encodes the layout while rav is defined in terms of γ. For FDM, it is important therefore to figure out the right memory layout such that rotations are completed in an efficient fashion.
- ι is a unary function, which takes a natural number n as its parameter and returns a 1-D array containing the range of natural numbers from 0 to n excluded. It is used to build *strides* of indexes needed by the ONF.

With these operations defined, we can proceed. We first apply the ψ-correspondence theorem followed by applying γ.

$\forall i$ such that $0 \leq i < 6$,

$$\begin{aligned}
&\langle i \rangle \ \psi \ ((1\ \theta_0\ A) + (-1\ \theta_0\ A)) \\
\equiv\ &(\mathsf{rav}\, A)[\gamma(\langle (i+1) \bmod 6 \rangle ; \langle 6 \rangle) \times 8 + \iota 8] \ + \ (\mathsf{rav}\, A)[\gamma(\langle (i-1) \bmod 6 \rangle ; \langle 6 \rangle) \times 8 + \iota 8] \\
\equiv\ &(\mathsf{rav}\, A)[(\langle (i+1) \bmod 6 \rangle) \times 8 + \iota 8] \ + \ (\mathsf{rav}\, A)[(\langle (i-1) \bmod 6 \rangle) \times 8 + \iota 8].
\end{aligned}$$

Secondly, we apply rav and turn ι into a loop to reach the following generic program:

$\forall j$ such that $0 \leq j < 8$,

$$A[((i+1) \bmod 6) \times 8 + j] \ + \ A[((i-1) \bmod 6) \times 8 + j].$$

The ONF is concerned with performance, and is where cost analysis and *dimension lifting* begins.

Regarding pure cost analysis, at this point, it is still possible to optimize this program: unfolding the loops gives us the insight that the modulo operation is only ever useful on the $0^{\mathtt{th}}$ and $5^{\mathtt{th}}$ row. Thus, by splitting the cases into those that require the modulo operation to be run and those that do not, we may achieve better performance.

[2] Here, only γ on rows is considered, but other γ functions exist

Now imagine breaking the problem over 2 processors. Conceptually, the dimension is lifted. It is important to note that the lifting may happen on any axis, especially in the current case where we are dealing with rotations on a given axis. If we happen to apply dimension lifting on the axis on which we are rotating, we may not be able to split the memory perfectly between the different computing sites. This could require inter-process communication, or duplication of memory.

In this case, since we are rotating on the $0^{\mathtt{th}}$ axis, we pick axis 1 as the candidate to be lifted. The loop on j is then split into 2 loops because we now view the 2-D resultant array as a 3-D array A' with shape $\left\langle 6\ 2\ \frac{8}{2}\right\rangle = \langle 6\ 2\ 4\rangle$ in which axis 1 corresponds to the number of processors. Therefore, we get:

$\forall i, j$ such that $0 \le i < 6, 0 \le j < 2$,

$$\begin{aligned}
& \left\langle i\, j\right\rangle \ \psi\ ((1\,\theta_0\,A') + (-1\,\theta_0\,A')) \\
\equiv\ & (\mathsf{rav}\,A')[\gamma(\left\langle((i+1) \bmod 6)\, j\right\rangle; \langle 6\ 2\rangle) \times 4 + \imath 4] + \\
& (\mathsf{rav}\,A')[\gamma(\left\langle((i-1) \bmod 6)\, j\right\rangle; \langle 6\ 2\rangle) \times 4 + \imath 4] \\
\equiv\ & (\mathsf{rav}\,A')[(((i+1) \bmod 6) \times 2 + j) \times 4 + \imath 4] + \\
& (\mathsf{rav}\,A')[(((i-1) \bmod 6) \times 2 + j) \times 4 + \imath 4].
\end{aligned}$$

This reduces to the following generic program:

$\forall k$ such that $0 \le k < 4$

$$A'[((i+1) \bmod 6) \times 4 \times 2 + j \times 4 + k] + A'[((i-1) \bmod 6) \times 4 \times 2 + j \times 4 + k].$$

As discussed above, there are other ways to achieve splitting of the problem across several computing sites. In general, the size of the array and the cost of accessing different architectural components drive the decision to break the problem up over processors, GPUs, threads, etc. [96; 98].

If a decision was made to break up the operations over different calculation units, the loop would be the same but the cost of performing the operation would be different. This decision is therefore completely cost-driven.

Continuing with *dimension lifting*, a choice might be made to use vector registers. This is, once again, a cost-driven decision, which may however be decided upon statically, prior to execution.

If we were to break our problem up over several processors and using vector registers, it would conceptually go from 2 dimensional to 4 dimensional, using indexing to access each resource. The same process can be applied to hardware components [78], e.g. pipelines, memories, buffers, etc., to achieve optimal throughput.

2.5 PDE Solver Test Case

Coordinate-free numerics [74; 89] is a high-level approach to writing solvers for PDEs. Solvers are written using high-level operators on abstract tensors. Take for instance Burgers' equation [27],

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = \nu \nabla^2 \vec{u},$$

where vector $\vec{u}$ denotes a time and space varying velocity vector, t is time, and the scalar ν is a viscosity coefficient. Burgers' equation is a PDE involving temporal ($\frac{\partial}{\partial t}$) and spatial ($\nabla$) derivative operations.

Applying an explicit second order Runge-Kutta time-integration method, the coordinate-free time-integrated equation can be coded in Magnolia as follows.

```
procedure burgersTimestep(upd u:Tensor1V, obs dt:R,
                          obs nu:R) = {
  var u_t = nu * laplacian(u) - dot(u,gradient(u));
  var u_substep = u + dt/2 * u_t;
  u_t = nu * laplacian(u_substep)
      - dot(u_substep,gradient(u_substep));
  u = u + dt * u_t;
};
```

Note how close this code follows the mathematical high-level formulation (2.5). We can lower the abstraction level of this code by linking it with a library for 3D cartesian coordinates based on continuous ringfields [90]. Next it can be linked with a library for finite difference methods choosing, e.g., stencils $\langle -\frac{1}{2}, 0, \frac{1}{2} \rangle$ and $\langle 1, -2, 1 \rangle$ for first and second order partial derivatives, respectively. This takes us to a code at the MoA level, consisting of rotate and maps of arithmetic operations [28]. With some reorganisation, we end up with the solver code below, expressed using MoA. The code calls the `snippet` six times forming one full time integration step, one call for each of the three dimensions of the problem times two due to the half-step in the time-integration. The variables `dt,nu,dx` are scalar (floating point). The first two come from the code above, while `dx` was introducd by the finite difference method. The variables `u0,u1,u2` are multiarrays (3D each), for each of the components of the 3D velocity vectorfield. These variables will be updated during the computation. The variables `c0,c1,c2,c3` and `c4` are numeric constants. Three temporary multiarray variables `v0,v1,v2` are computed in the first three `snippet` calls, due to the half-step. They are then used in the last three `snippet` calls to update `u0,u1,u2`.

```
procedure step(upd u0:MA, upd u1:MA, upd u2:MA,
               obs nu:Float, obs dx:Float, obs dt:Float) {

  var c0 = 0.5/dx;
  var c1 = 1/dx/dx;
  var c2 = 2/dx/dx;
  var c3 = nu;
  var c4 = dt/2;

  var v0 = u0;
  var v1 = u1;
  var v2 = u2;
  call snippet(v0,u0,u0,u1,u2,c0,c1,c2,c3,c4);
  call snippet(v1,u1,u0,u1,u2,c0,c1,c2,c3,c4);
  call snippet(v2,u2,u0,u1,u2,c0,c1,c2,c3,c4);
  call snippet(u0,v0,v0,v1,v2,c0,c1,c2,c3,c4);
  call snippet(u1,v1,v0,v1,v2,c0,c1,c2,c3,c4);
  call snippet(u2,v2,v0,v1,v2,c0,c1,c2,c3,c4);
};
```

In the actual `snippet` code, `d1a,d2a,d1b,d2b,d1c,d2c` and `shift_v` are temporary multiarray

variables. The `shift` function takes as first argument the multiarray being shifted, then the direction of the shift, and lastly the distance for the rotational shift.

```
procedure snippet(upd u:MA, obs v:MA,
                  obs u0:MA, obs u1:MA, obs u2:MA,
                  obs c0:Float, obs c1:Float, obs c2:Float,
                  obs c3:Float, obs c4:Float) {

  var shift_v = shift ( v, 0, -1 );
  var d1a = -c0  * shift_v;
  var d2a =  c1 * shift_v - c2 * u0;
  shift_v = shift ( v, 0, 1 );
  d1a = d1a + c0  * shift_v;
  d2a = d2a + c1 * shift_v;

  shift_v = shift ( v, 1, -1 );
  var d1b = -c0  * shift_v;
  var d2b = c1 * shift_v - c2 * u0;
  shift_v = shift ( v, 1, 1 );
  d1b = d1b + c0  * shift_v;
  d2b = d2b + c1 * shift_v;

  shift_v = shift ( v, 2, -1 );
  var d1c = -c0  * shift_v;
  var d2c = c1 * shift_v - c2 * u0;
  shift_v = shift ( v, 2, 1 );
  d1c = d1c + c0  * shift_v;
  d2c = d2c + c1 * shift_v;

  d1a = u0 * d1a + u1 * d1b + u2 * d1c;
  d2a = d2a + d2b + d2c;
  u = u + c4 * ( c3 * d2a - d1a);
};
```

In essence, `snippet` is computing 1/3 of the half-step of the PDE, using common calls to rotate to compute one first and one second order partial derivative.

2.5.1 Reduction using MoA

Using the reduction rules defined in the ψ-calculus, and turning our `snippet` code into an expression, we can reduce the code to a DNF representation. In the following, we spell out some of the transformation steps. The equation

$$\begin{aligned}\texttt{snippet} = u + c_4 \times (c_3 \times (c_1 \times \\ ((-1\,\theta_0\,v) + (1\,\theta_0\,v) + (-1\,\theta_1\,v) + (1\,\theta_1\,v) + (-1\,\theta_2\,v) + (1\,\theta_2\,v)) - 3c_2u_0) \; - c_0 \times \\ (((1\,\theta_0\,v) - (-1\,\theta_0\,v))\,u_0 + ((1\,\theta_1\,v) - (-1\,\theta_1\,v))\,u_1 + ((1\,\theta_2\,v) - (-1\,\theta_2\,v))\,u_2))\end{aligned}$$

is a transcription of the `snippet` code above.

We use the notation θ_x to denote a rotation around the $x^{\texttt{th}}$ axis, represented in Magnolia by calls to `shift(multiarray, axis, offset)`.

The Magnolia implementation of the snippet makes heavy use of the multiarrays `d1x` and `d2x`, where `x` denotes the axis around which the multiarray is rotated in lexicographical order (`a` corresponds to the $0^{\texttt{th}}$ axis, `b` to the 1^{st} and so on). For the sake of easing into it, let us start by building a generic DNF representation for `d2x`. All the steps will be detailed explicitly in order to gain insights on what is needed and what is possible.

$$\langle i\,j\,k\rangle \;\psi\, d_{2x} = \langle i\,j\,k\rangle \;\psi\, (c_1 \times (-1\,\theta_x\, v) + c_1 \;\times (1\,\theta_x\, v) - c_2 \times u_0)$$

(distribute ψ over +/-)

$$= \langle i\,j\,k\rangle \;\psi\, (c_1 \times (-1\,\theta_x\, v)) + \langle i\,j\,k\rangle \;\psi\, (c_1 \times (1\,\theta_x\, v)) - \langle i\,j\,k\rangle \;\psi\, (c_2 \times u_0)$$

(extract constant factors)

$$= c_1 \times (\langle i\,j\,k\rangle \;\psi\, (-1\,\theta_x\, v)) + c_1 \times (\langle i\,j\,k\rangle \;\psi\, (1\,\theta_x\, v)) - c_2 \times (\langle i\,j\,k\rangle \;\psi\, u_0)$$

(factorize by c_1)

$$= c_1 \times (\langle i\,j\,k\rangle \;\psi\, (-1\,\theta_x\, v) + \langle i\,j\,k\rangle \;\psi\, (1\,\theta_x\, v)) - c_2 \times (\langle i\,j\,k\rangle \;\psi\, u_0).$$

Using the MoA's concatenation of index property, we can now define $\langle\ i\ \rangle\ \psi\ d_{2x}$. However, this is only reducible if $x = 0$. The reason is that to reduce an expression using a rotation on the $x^{\texttt{th}}$ axis further, one needs to apply ψ with an index of at least $x + 1$ elements. Therefore, to reduce d_{21}, we need an index vector with at least 2 elements, while we need a total index containing 3 elements to reduce d_{22}. With that in mind, we can try to reduce d_{21}:

$$\langle i\,j\rangle \;\psi\, d_{21} = c_1 \times (\langle i\,j\rangle \;\psi\, (-1\,\theta_1\, v) + \langle i\,j\rangle \;\psi\, (1\,\theta_1\, v)) - c_2 \times (\langle i\,j\rangle \;\psi\, u_0)$$

(reducing rotation)

$$= c_1 \times (\langle i\ ((j-1) \bmod s_1)\rangle \;\psi\, v + \langle i\ ((j+1) \bmod s_1)\rangle \;\psi\, v) - c_2 \times (\langle i\,j\rangle \;\psi\, u_0).$$

For $x = 2$, we apply the same process with a total index:

$$\langle i\,j\,k\rangle \;\psi\, d_{22} = c_1 \times (\langle i\,j\,k\rangle \;\psi\, (-1\,\theta_2\, v) + \langle i\,j\,k\rangle \;\psi\, (1\,\theta_2\, v)) - c_2 \times (\langle i\,j\,k\rangle \;\psi\, u_0)$$

(reducing rotation)

$$\begin{aligned} &= c_1 \times (\langle i\,j\ ((k-1) \bmod s_2)\rangle \;\psi\, v + \langle i\,j\ ((k+1) \bmod s_2)\rangle \;\psi\, v) - \\ &\quad c_2 \times (\langle i\,j\,k\rangle \;\psi\, u_0) \end{aligned}$$

Now we can define the ONF of the expression, which is the form we will use in our actual code. Let's define it for d_{21}:

$$\begin{aligned}(\mathsf{rav}\, d_{21})[\gamma(\langle i\, j\rangle ; \langle s_0\, s_1\rangle) \times s_2 + \imath s_2] \;&= c_1 \times ((\mathsf{rav}\, v)[\gamma(\langle i\ ((j-1) \bmod s_1)\rangle ; \langle s_0\, s_1\rangle) \times s_2 + \imath s_2] + \\ &\quad (\mathsf{rav}\, v)[\gamma(\langle i\ ((j+1) \bmod s_1)\rangle ; \langle s_0\, s_1\rangle) \times s_2 + \imath s_2]) - \\ &\quad c_2 \times (\mathsf{rav}\, u_0)[\gamma(\langle i\, j\rangle ; \langle s_0\, s_1\rangle) \times s_2 + \imath s_2]\end{aligned}$$

(apply γ on both sides)

$$\begin{aligned}(\mathsf{rav}\, d_{21})[(i \times s_1 \times s_2 + j \times s_2 + \imath s_2] \;&= c_1 \times ((\mathsf{rav}\, v)[i \times s_1 \times s_2 + ((j-1) \bmod s_1) \times s_2 + \imath s_2] + \\ &\quad (\mathsf{rav}\, v)[i \times s_1 \times s_2 + ((j+1) \bmod s_1) \times s_2 + \imath s_2]) - \\ &\quad c_2 \times (\mathsf{rav}\, u_0)[i \times s_1 \times s_2 + j \times s_2 + \imath s_2].\end{aligned}$$

The optimization can be done similarly for d_{22}. The fact that d_{22} can only be reduced using a total index means that `snippet` too can only be fully reduced using a total index.

$$\begin{aligned}&\langle i\, j\, k\rangle\ \psi\ \texttt{snippet} \\ =\ &\langle i\, j\, k\rangle\ \psi\ (u + c_4 \times (c_3 \times (c_1 \times ((-1\ \theta_0\ v) + (1\ \theta_0\ v) + (-1\ \theta_1\ v) + (1\ \theta_1\ v) + (-1\ \theta_2\ v) + \\ &\qquad (1\ \theta_2\ v)) - 3c_2u_0)\ - c_0(((1\ \theta_0\ v) - (-1\ \theta_0\ v))\ u_0 + ((1\ \theta_1\ v) - (-1\ \theta_1\ v))\ u_1 + \\ &\qquad ((1\ \theta_2\ v) + (-1\ \theta_2\ v))\ u_2)))\end{aligned}$$

(distribute ψ over + and -)

$$\begin{aligned}=\ &\langle i\, j\, k\rangle\ \psi\ u + \\ &\langle i\, j\, k\rangle\ \psi\ (c_4 \times (c_3\ \times (c_1 \times ((-1\ \theta_0\ v) + (1\ \theta_0\ v) + (-1\ \theta_1\ v) + (1\ \theta_1\ v) + (-1\ \theta_2\ v) + \\ &\qquad (1\ \theta_2\ v)) - 3c_2u_0))) - \\ &\langle i\, j\, k\rangle\ \psi\ (c_0 \times (((1\ \theta_0\ v) - (-1\ \theta_0\ v))\ u_0 + ((1\ \theta_1\ v) - (-1\ \theta_1\ v))\ u_1 + ((1\ \theta_2\ v) - \\ &\qquad (-1\ \theta_2\ v))\ u_2))\end{aligned}$$

(extract constant c_4, c_3, and c_0)

$$\begin{aligned}=\ &\langle i\, j\, k\rangle\ \psi\ u + c_4 \times (c_3 \times \\ &(\langle i\, j\, k\rangle\ \psi\ (c_1 \times ((-1\ \theta_0\ v) + (1\ \theta_0\ v) + (-1\ \theta_1\ v) + (1\ \theta_1\ v) + (-1\ \theta_2\ v) + (1\ \theta_2\ v)) - \\ &\qquad 3c_2u_0)) - c_0 \times \\ &(\langle i\, j\, k\rangle\ \psi\ (((1\ \theta_0\ v) - (-1\ \theta_0\ v))\ u_0 + ((1\ \theta_1\ v) - (-1\ \theta_1\ v))\ u_1 + ((1\ \theta_2\ v) - \\ &\qquad (-1\ \theta_2\ v))\ u_2)))\end{aligned}$$

(distribute ψ over +, ×, and -)

$$\begin{aligned}=\ &\langle i\, j\, k\rangle\ \psi\ u + c_4 \times (c_3 \times \\ &(\langle i\, j\, k\rangle\ \psi\ (c_1 \times ((-1\ \theta_0\ v) + (1\ \theta_0\ v)\ + (-1\ \theta_1\ v) + (1\ \theta_1\ v)\ + (-1\ \theta_2\ v) + (1\ \theta_2\ v))) - \\ &\langle i\, j\, k\rangle\ \psi\ (3c_2u_0)) - c_0 \times (\langle i\, j\, k\rangle\ \psi\ ((1\ \theta_0\ v) - (-1\ \theta_0\ v)) \times \langle i\, j\, k\rangle\ \psi\ u_0 +\end{aligned}$$

$$\langle i\,j\,k\rangle\ \psi\ ((1\,\theta_1\,v) - (-1\,\theta_1\,v)) \times \langle i\,j\,k\rangle\ \psi\ u_1 +$$
$$\langle i\,j\,k\rangle\ \psi\ ((1\,\theta_2\,v) - (-1\,\theta_2\,v)) \times \langle i\,j\,k\rangle\ \psi\ u_2))$$

(extract constant factors c_1 and $3 \times c_2$)

$$= \langle i\,j\,k\rangle\ \psi\ u + c_4 \times (c_3 \times (c_1 \times$$
$$(\langle i\,j\,k\rangle\ \psi\ ((-1\,\theta_0\,v) + (1\,\theta_0\,v) + (-1\,\theta_1\,v) + (1\,\theta_1\,v) + (-1\,\theta_2\,v) + (1\,\theta_2\,v))) - 3c_2$$
$$(\langle i\,j\,k\rangle\ \psi\ u_0)) - c_0 \times (\langle i\,j\,k\rangle\ \psi\ ((1\,\theta_0\,v) - (-1\,\theta_0\,v)) \times \langle i\,j\,k\rangle\ \psi\ u_0 +$$
$$\langle i\,j\,k\rangle\ \psi\ ((1\,\theta_1\,v) - (-1\,\theta_1\,v)) \times \langle i\,j\,k\rangle\ \psi\ u_1 +$$
$$\langle i\,j\,k\rangle\ \psi\ ((1\,\theta_2\,v) - (-1\,\theta_2\,v)) \times \langle i\,j\,k\rangle\ \psi\ u_2))$$

(distribute ψ over + and -)

$$= \langle i\,j\,k\rangle\ \psi\ u + c_4 \times (c_3 \times (c_1 \times$$
$$(\langle i\,j\,k\rangle\ \psi\ (-1\,\theta_0\,v) + \langle i\,j\,k\rangle\ \psi\ (1\,\theta_0\,v) + \langle i\,j\,k\rangle\ \psi\ (-1\,\theta_1\,v) +$$
$$\langle i\,j\,k\rangle\ \psi\ (1\,\theta_1\,v) + \langle i\,j\,k\rangle\ \psi\ (-1\,\theta_2\,v) + \langle i\,j\,k\rangle\ \psi\ (1\,\theta_2\,v)) - 3c_2$$
$$(\langle i\,j\,k\rangle\ \psi\ u_0)) - c_0 \times ((\langle i\,j\,k\rangle\ \psi\ (1\,\theta_0\,v) - \langle i\,j\,k\rangle\ \psi\ (-1\,\theta_0\,v)) \times \langle i\,j\,k\rangle\ \psi\ u_0 +$$
$$(\langle i\,j\,k\rangle\ \psi\ (1\,\theta_1\,v) - \langle i\,j\,k\rangle\ \psi\ (-1\,\theta_1\,v)) \times \langle i\,j\,k\rangle\ \psi\ u_1 +$$
$$(\langle i\,j\,k\rangle\ \psi\ (1\,\theta_2\,v) - \langle i\,j\,k\rangle\ \psi\ (-1\,\theta_2\,v)) \times \langle i\,j\,k\rangle\ \psi\ u_2))$$

(translate rotations into indexing)

$$= \langle i\,j\,k\rangle\ \psi\ u + c_4 \times (c_3 \times (c_1 \times$$
$$(\langle((i-1) \bmod s_0)\,j\,k\rangle\ \psi\ v + \langle((i+1) \bmod s_0)\,j\,k\rangle\ \psi\ v +$$
$$\langle i\,((j-1) \bmod s_1)\,k\rangle\ \psi\ v + \langle i\,((j+1) \bmod s_1)\,k\rangle\ \psi\ v +$$
$$\langle i\,j\,((k-1) \bmod s_2)\rangle\ \psi\ v + \langle i\,j\,((k+1) \bmod s_2)\rangle\ \psi\ v) - 3c_2(\langle i\,j\,k\rangle\ \psi\ u_0)) - c_0 \times$$
$$((\langle((i+1) \bmod s_0)\,j\,k\rangle\ \psi\ v - \langle((i-1) \bmod s_0)\,j\,k\rangle\ \psi\ v) \times \langle i\,j\,k\rangle\ \psi\ u_0 +$$
$$(\langle i\,((j+1) \bmod s_1)\,k\rangle\ \psi\ v - \langle i\,((j-1) \bmod s_1)\,k\rangle\ \psi\ v) \times \langle i\,j\,k\rangle\ \psi\ u_1 +$$
$$(\langle i\,j\,((k+1) \bmod s_2)\rangle\ \psi\ v - \langle i\,j\,((k-1) \bmod s_2)\rangle\ \psi\ v) \times \langle i\,j\,k\rangle\ \psi\ u_2))$$

In Magnolia, the DNF can be captured as such:

```
procedure snippetDNF(
    upd u:MA, obs v:MA, obs u0:MA, obs u1:MA, obs u2:MA,
    obs c0:Float, obs c1:Float, obs c2:Float, obs c3:Float,
    obs c4:Float, obs mi:MI) {
  var s0 = shape0(v);
  var s1 = shape1(v);
  var s2 = shape2(v);
  u = psi(mi,u) + c4*(c3*(c1*(
      psi(mod0(mi-d0,s0),v) + psi(mod0(mi+d0,s0),v) +
      psi(mod1(mi-d1,s1),v) + psi(mod1(mi+d1,s1),v) +
      psi(mod2(mi-d2,s2),v) + psi(mod2(mi+d2,s2),v)) -
      3*c2* psi(mi,u0)) - c0 *
      ((psi(mod0(mi+d0,s0),v) - psi(mod0(mi-d0,s0),v)) *
```

```
        psi(mi,u0) +
        (psi(mod1(mi+d1,s1),v) - psi(mod1(mi-d1,s1),v)) *
        psi(mi,u1) +
        (psi(mod2(mi+d2,s2),v) - psi(mod2(mi-d2,s2),v)) *
        psi(mi,u2)));
}
```


Now, we can transform `snippet` into its ONF form:

$$
\begin{aligned}
&(\mathsf{rav}\,\texttt{snippet})[\gamma(\langle i\,j\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] \\
=\ &(\mathsf{rav}\,u)[\gamma(\langle i\,j\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] + c_4 \times (c_3 \times (c_1 \times \\
&(\mathsf{rav}\,v)[\gamma(\langle((i-1) \bmod s_0)\,j\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] + \\
&(\mathsf{rav}\,v)[\gamma(\langle((i+1) \bmod s_0)\,j\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] + \\
&(\mathsf{rav}\,v)[\gamma(\langle i\,((j-1) \bmod s_1)\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] + \\
&(\mathsf{rav}\,v)[\gamma(\langle i\,((j+1) \bmod s_1)\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] + \\
&(\mathsf{rav}\,v)[\gamma(\langle i\,j\,((k-1) \bmod s_2)\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] + \\
&(\mathsf{rav}\,v)[\gamma(\langle i\,j\,((k+1) \bmod s_2)\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)]) - \\
&3c_2(\mathsf{rav}\,u)[\gamma(\langle i\,j\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)]\ - c_0 \times \\
&(((\mathsf{rav}\,v)[\gamma(\langle((i+1) \bmod s_0)\,j\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] - \\
&\quad(\mathsf{rav}\,v)[\gamma(\langle((i-1) \bmod s_0)\,j\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)]) \times (\mathsf{rav}\,u_0)[\gamma(\langle i\,j\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] + \\
&\quad((\mathsf{rav}\,v)[\gamma(\langle i\,((j+1) \bmod s_1)\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] - \\
&\quad(\mathsf{rav}\,v)[\gamma(\langle i\,((j-1) \bmod s_1)\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)]) \times (\mathsf{rav}\,u_1)[\gamma(\langle i\,j\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] + \\
&\quad((\mathsf{rav}\,v)[\gamma(\langle i\,j\,((k+1) \bmod s_2)\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)] - \\
&\quad(\mathsf{rav}\,v)[\gamma(\langle i\,j\,((k-1) \bmod s_2)\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)]) \times (\mathsf{rav}\,u_2)[\gamma(\langle i\,j\,k\rangle\ ;\ \langle s_0\,s_1\,s_2\rangle)]))
\end{aligned}
$$

This is how far we can go without specific information about the layout of the data in the memory and the architecture. The current form is still fully generic, with γ and rav parameterized over the layout. The Magnolia implementation of this generic form is as follows:

```
procedure moaONF (
    upd u:MA, obs v:MA, obs u0:MA, obs u1:MA, obs u2:MA,
    obs c0:Float, obs c1:Float, obs c2:Float, obs c3:Float,
    obs c4:Float, obs mi:MI) {

  var s0 = shape0(v);
  var s1 = shape1(v);
  var s2 = shape2(v);

  var newu = get(rav(u),gamma(mi,s)) + c4*(c3*(c1*
             get(rav(v),gamma(mod0(mi-d0,s0),s)) +
             get(rav(v),gamma(mod0(mi+d0,s0),s)) +
             get(rav(v),gamma(mod1(mi-d1,s1),s)) +
             get(rav(v),gamma(mod1(mi+d1,s1),s)) +
             get(rav(v),gamma(mod2(mi-d2,s2),s)) +
```

```
                get(rav(v),gamma(mod2(mi+d2,s2),s))) -
                3 * c_2 get(rav(u),gamma(mi,s)) - c0 *
                ((get(rav(v),gamma(mod0(mi+d0,s0),s)) -
                get(rav(v),gamma(mod0(mi-d0,s0),s))) *
                get(rav(u0),gamma(mi,s)) +
                (get(rav(v),gamma(mod1(mi+d1,s1),s)) -
                get(rav(v),gamma(mod1(mi-d1,s1),s))) *
                get(rav(u_1),gamma(mi,s)) +
                (get(rav(v),gamma(mod2(mi+d2,s2),s)) -
                get(rav(v),gamma(mod2(mi-d2,s2),s))) *
                get(rav(u2),gamma(mi,s))));

  set(rav(u),gamma(mi,s),newu);
}
```

In Section 2.4.3, we defined the layout of the data as row-major. Thus we can optimize the expression further by expanding the calls to γ:

$$
\begin{aligned}
&(\text{rav snippet})[i \times s_1 \times s_2 + j \times s_2 + k] \\
=\ &(\text{rav } u)[i \times s_1 \times s_2 + j \times s_2 + k] + c_4 \times (c_3 \times (c_1 \times \\
&(\text{rav } v)[((i-1) \bmod s_0) \times s_1 \times s_2 + j \times s_2 + k] + \\
&(\text{rav } v)[((i+1) \bmod s_0) \times s_1 \times s_2 + j \times s_2 + k] + \\
&(\text{rav } v)[i \times s_1 \times s_2 + ((j-1) \bmod s_1) \times s_2 + k] + \\
&(\text{rav } v)[i \times s_1 \times s_2 + ((j+1) \bmod s_1) \times s_2 + k] + \\
&(\text{rav } v)[i \times s_1 \times s_2 + j \times s_2 + ((k-1) \bmod s_2)] + \\
&(\text{rav } v)[i \times s_1 \times s_2 + j \times s_2 + ((k+1) \bmod s_2)]) - \\
&3c_2(\text{rav } u)[i \times s_1 \times s_2 + j \times s_2 + k] \ - c_0 \times \\
&(((\text{rav } v)[((i+1) \bmod s_0) \times s_1 \times s_2 + j \times s_2 + k] - \\
&\quad (\text{rav } v)[((i-1) \bmod s_0) \times s_1 \times s_2 + j \times s_2 + k]) \times (\text{rav } u_0)[i \times s_1 \times s_2 + j \times s_2 + k] + \\
&\quad ((\text{rav } v)[i \times s_1 \times s_2 + ((j+1) \bmod s_1) \times s_2 + k] - \\
&\quad (\text{rav } v)[i \times s_1 \times s_2 + ((j-1) \bmod s_1) \times s_2 + k]) \times (\text{rav } u_1)[i \times s_1 \times s_2 + j \times s_2 + k] + \\
&\quad ((\text{rav } v)[i \times s_1 \times s_2 + j \times s_2 + ((k+1) \bmod s_2)] - \\
&\quad (\text{rav } v)[i \times s_1 \times s_2 + j \times s_2 + ((k-1) \bmod s_2)]) \ \times (\text{rav } u_2)[i \times s_1 \times s_2 + j \times s_2 + k]))
\end{aligned}
$$

At this point, as indicated in section 4.3, we can convert our expression into several subexpressions in order to distinguish the general case from anomalies (i.e cases that require the modulo operation to be applied on any axis). This general case is in ONF and we can use it for code generation or to perform additional transformations, specifically dimension lifting.

2.6 Conclusion

Through the full analysis of an FDM solver of a PDE, we were able to extract a rewriting subsystem most relevant to our specific problem out of the rewriting rules provided by the ψ-calculus. Then, we

proved that this particular set of rewriting rules constitutes a canonical rewriting system, getting one step closer to fully automating the optimization of array computations using the MoA formalism.

We are now working on the implementation of our optimizations to measure their impact on the performance of the solver for different architectures, and can report results in the near future.

By working out an approach from high level coordinate-free PDEs down to preparing for data layout and code optimization using MoA as an intermediate layer through the full exploration of a relevant example, we pave the way for building similar systems for any problem of the same category. High-efficiency code can thus easily be explored and generated from a unique high-level abstraction and potentially different implementation algorithms, layouts of data or hardware architectures.

Because tensors dominate a significant portion of science, future work may focus on figuring out what properties can be deduced from the complete ψ-calculus rewriting system with a goal to extend this currently problem-oriented approach towards a fully automated problem-independent optimization tool based on MoA.

Given the scale of the ecosystem impacted by this kind of work, such prospects are very attractive.

Paper 3

Padding in the Mathematics of Arrays[I]

Benjamin Chetioui[a], Ole Abusdal[b], Magne Haveraaen[a], Jaakko Järvi[a], Lenore Mullin[c]

[a]Department of Informatics, University of Bergen, Norway, [b]Department of Computer science, Electrical engineering and Mathematical sciences, Western Norway University of Applied Sciences, Norway, [c]College of Engineering and Applied Sciences, University at Albany, SUNY, USA

Abstract

Multi-dimensional array manipulation constitutes a core component of numerous numerical methods, e.g. finite difference solvers of Partial Differential Equations (PDEs). The efficiency of such computations is tightly connected to traversing array data in a hardware-friendly way.

The Mathematics of Arrays (MoA) allows reasoning about array computations at a high level and enables systematic transformations of array-based programs. We have previously shown that stencil computations reduce to MoA's Denotational Normal Form (DNF).

Here we bring to light MoA's Operational Normal Forms (ONFs) that allow for adapting array computations to hardware characteristics. ONF transformations start from the DNF. Alongside the ONF transformations, we extend MoA with rewriting rules for padding. These new rules allow both a simplification of array indexing and a systematic approach to introducing halos to PDE solvers. Experiments on various architectures confirm the flexibility of the approach.

[I]Benjamin Chetioui, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Lenore Mullin. Padding in the Mathematics of Arrays. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2021, page 15–26, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384667. doi:10.1145/3460944.3464311

3.1 Introduction

In the past few decades, a large variety of high-performance computing (HPC) architectures has appeared. On the path towards exascale computing, we can expect to see a similar medley of architectures. Software for HPC therefore needs to be highly adaptable. This includes adjusting to, among other things, different memory hierarchies and changing intra- and interprocess communication hardware.

This paper explores the Mathematics of Arrays (MoA) formalism [127] as a tool for optimizing array codes for different hardware architectures. We have previously established a means of transforming stencil-based array code to *Denotational Normal Form* (DNF) [36]—irreducible expressions in the language of MoA. Given knowledge of the targeted parallel distribution and memory layout, one can transform a DNF expression to an architecture-specific normal form, the *Operational Normal Form* (ONF), which we describe in Section 3.4.

ONF transformations include the *dimension lifting* operation for reshaping an array by splitting a given axis of its shape into two or more dimensions. This operation can conveniently divide an array over different computation loci (whether they be threads, cores, or even systems).

The contribution of this paper is a formalization of concepts of MoA's ONF and the extension of MoA with new operations to deal with padding of data. With these operations, MoA provides a framework for transforming regular array stencil code to distributed code with *halo zones* — also referred to as *ghost cells* in the literature [110]. As an example, the paper shows how MoA and its ONF help in the search for more efficient stencil-based array computations in a Partial Differential Equation (PDE) solver based on Finite Difference Methods (FDMs). We obtain a 10% performance improvement with changes easily expressible in MoA.

We have started to implement MoA with our extensions in Coq, so that the formal claims we make, e.g., about the ONF transformations can be machine-checked. This effort is at an early stage; the proofs can be found in the repository at `https://github.com/mathematics-of-arrays/moa-formalization`.

The paper is organised as follows. Next is a motivation section, then a discussion of related work. Section 3.4 covers the required prerequisites in MoA and previous work on the DNF layer. Section 3.5 discusses dimension lifting, and defines and explains padding and data layout. We then briefly report on some experiments and conclude in Section 3.7.

3.2 Motivation

To motivate our work, we ran the PDE solver we presented in [36] on a set of experimental architectures and implemented some of the ONF transformations on the code. Table 3.1 shows a matrix where each column corresponds to a different version of the solver and each row to different hardware. The table makes it plain that different architectures benefit from different transformations. While on CPU 1 the *dimension lifting on* 0^{th} *axis and tiled memory* approach performs best, on CPU 3 it is clearly inefficient compared to the other dimension lifting-based scenarios.

The hard to predict variations in performance, and the sheer number of different memory layouts,

Table 3.1: Execution time (in seconds) of a PDE solver C implementation compiled with GCC 8.2.0 depending on hardware and dimension lifting (DL) parameters. The arrays involved in the computation are cubic, and each axis has length 512. The gray background marks the fastest version(s) of the solver for each row. The labels are as follows: S: Single core (no DL); MDL: Multicore (DL on 0^{th} dimension); MDLSL: Multicore (DL on $(n-2)^{\text{th}}$ dimension); MDLTM: Multicore (DL on 0^{th} dimension using tiled memory); CPU 1: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz; CPU 2: AMD EPYC 7601 32-Core; CPU 3: Intel(R) Xeon(R) Silver 4112 CPU @ 2.60GHz; and CPU 4: ThunderX2 CN9980. The code of the experiments is at https://github.com/mathematics-of-arrays/padding-in-the-mathematics-of-arrays.

	S	MDL	MDLSL	MDLTM
CPU 1	225.74	70.96	66.66	61.81
CPU 2	299.42	59.16	68.14	68.70
CPU 3	172.71	85.97	85.59	117.11
CPU 4	660.53	85.06	72.80	77.86

motivate a vehicle for easy exploration of codes that implement different memory layouts. If exploring different layouts is made easy, programmers can obtain close-to-optimal performance for different architectures with little effort.

In the following, we demonstrate that MoA, with our extension of operations for padding, provides the required level of expressivity to accomplish just that.

3.3 Related

Ken Iverson introduced whole-array operations in the APL programming language [100]. Building on further explorations by Abrams [4], Mullin created the Mathematics of Arrays formalism [127] in order to address various shortcomings of the universal algebra underlying APL (most notably the lack of a calculus for indexing). MoA is intended to serve as a foundation for exploring and optimizing array/tensor operations. Mullin further explored MoA through case studies of scientific algorithms, including QR Decomposition [144] and Fast Fourier Transforms (FFTs) [98]. The latter paper introduced the dimension lifting operation, crucial to this work.

Burrows et al. identified an array API for FDM solvers of PDEs [28]. We explored the MoA fragment corresponding to this API and concluded that stencil computations can systematically be reduced to MoA's DNF [36]. Hagedorn et al. [81] also looked into optimizing stencil computations, and augmented LIFT [167] with the same operations.

Artjom Šinkarovs studied automatic data layout transformations using a type-based approach and demonstrated that carefully chosen data layouts can greatly improve program vectorisation, therefore leading to significant performance improvements [182]. Šinkarovs et al. also implemented a Convolutional Neural Network in APL [176]—a setting in which stencil-related padding operations are relevant. Šinkarovs's formalization in Agda of multiarrays à la APL in Agda (see https://github.com/ashinkarov/agda-array) uses the latter as an example.

3.4 MoA Background and Notation

We give a short introduction to the MoA algebra for representing and describing operations on arrays. For more details, we refer the reader to the relevant works in our bibliography [6; 127; 130].

MoA defines the ψ-calculus, a set of rules for manipulating array shapes and expressions. By systematically applying a set of terminating rewriting rules, we can transform an array expression to a single array with standard layout and operations on the array elements, the Denotational Normal Form (DNF). DNF can further be transformed into a corresponding Operational Normal Form (ONF), which represents array access patterns in terms of *start*, *stride* and *length*. Together with *dimension lifting*, this lets us reorganize the memory layout and data access patterns, and to thus take into account distribution of data and memory hierarchies, data locality, etc., a flexiblity needed for current and future hardware architectures.

The *dimension* of an array A corresponds to the number of axes in A and is denoted by $\mathsf{dim}(A)$. We define the *shape* of an n-dimensional array A as a vector $\langle s_0\,,\ \ldots\ ,\ s_{n-1}\rangle$ containing at index i the length of i^{th} axis in A. The *size* of an array is the number of elements it contains, i.e. $\mathsf{size}(A) = \prod_{i=0}^{n-1} s_i$; we write this product of shape s also as Πs.

We adopt the notation $\mathsf{Fin}\ n$ for the finite set of natural numbers $\{0, \ldots, n-1\}$. An *index* into A is a vector $\langle i_0\,,\ \ldots\ ,\ i_k\rangle$ of length $k+1$ with $k \in \mathsf{Fin}\ (\mathsf{dim}(A))$ such that $i_j \in \mathsf{Fin}\ s_j$ for all $j \in \mathsf{Fin}\ k$. If $\mathsf{dim}(A) = k+1$, we say the index is a *total index*. The indexing function that defines the content of the array at a given index differs depending on the abstraction layer we consider.

For example, a 2-dimensional array M with shape $\langle 2, 3\rangle$ contains 6 elements and corresponds to a 2-by-3 matrix. We represent such an array using the row-major notation

$$M = \begin{pmatrix} e_{0,0} & e_{0,1} & e_{0,2} \\ e_{1,0} & e_{1,1} & e_{1,2} \end{pmatrix},$$

where $e_{i,j}$ is the element of M at total index $\langle i\,j\rangle$.

Scalars are represented as 0-dimensional arrays, i.e. arrays with shape $\langle\rangle$ and size 1. Empty arrays have size 0, i.e. at least one of their shape components is 0.

3.4.1 Relevant MoA Operations at the DNF level

In the following, A is an n-dimensional array with shape $\langle s_0\,,\ \ldots\ ,\ s_{n-1}\rangle$. The rest of the paper makes use of the following core operations at the DNF level:

- the shape function ρ, that returns the shape of an array, e.g. $\rho(M) = \langle 2, 3\rangle$, where M is the 2-by-3 array from the example above;
- the indexing function ψ, that takes an index into A and returns the subarray at the indexed position. Thus, $\langle\rangle\ \psi\, A = A$ holds. For our example, we have

  $$\langle 1\rangle\ \psi\, M = \begin{pmatrix} e_{1,0} & e_{1,1} & e_{1,2} \end{pmatrix}$$

  and $\rho(\langle 1\rangle\ \psi\, M) = \langle 3\rangle$;

When A is 1-dimensional and therefore has shape $\langle s_0 \rangle$, we also use the notation

$$A[u],$$

where the index u is either

- a scalar $u \in \mathsf{Fin}\, s_0$, and $A[u]$ is the element at index u in A; or
- a vector of k indices $\langle u_0\,,\ \ldots\ ,\ u_{k-1} \rangle$ such that $\forall j \in \mathsf{Fin}\, k$, $u_j \in \mathsf{Fin}\, s_0$, and $A[u]$ is the vector whose j^{th} element is the $u_j{}^{\text{th}}$ element in A;

- the reshaping function reshape, that takes a shape s with $\Pi s = \Pi \rho(A)$, and produces an array with the same size and elements as A but with shape s, i.e. $\rho(\mathsf{reshape}(s, A)) = s$. Note that reshape does not move data around in A;
- a slicing function $\triangle$ (read "take"), that takes a positive (respectively negative) integer t such that $|t| \in \mathsf{Fin}\ (s_0 + 1)$ and returns a slice containing the first (respectively last) $|t|$ subarrays of A. Thus,

$$\rho(\triangle(t, A)) = \langle |t|\,,\ s_1\,,\ \ldots\ ,\ s_{n-1} \rangle$$

and $\forall i \in \mathsf{Fin}\, |t|$,

$$\langle i \rangle\ \psi\ \triangle(t, A) = \begin{cases} \langle i \rangle\ \psi\ A & \text{if } t \geq 0 \\ \langle s_0 - |t| + i \rangle\ \psi\ A & \text{otherwise;} \end{cases}$$

- a slicing function $\triangledown$ (read "drop"), that takes a positive (respectively negative) integer t such that $|t| \in \mathsf{Fin}\ (s_0 + 1)$ and returns a slice containing the last (respectively first) $s_0 - |t|$ subarrays of A. Thus,

$$\rho(\triangledown(t, A)) = \langle s_0 - |t|\,,\ s_1\,,\ \ldots\ ,\ s_{n-1} \rangle$$

and $\forall i \in \mathsf{Fin}\ (s_0 - |t|)$,

$$\langle i \rangle\ \psi\ \triangledown(t, A) = \begin{cases} \langle i + t \rangle\ \psi\ A & \text{if } t \geq 0 \\ \langle i \rangle\ \psi\ A & \text{otherwise.} \end{cases}$$

- the concatenation function cat, that takes an additional array B with shape $\left\langle s_0^B,\ s_1\,,\ \ldots\ ,\ s_{n-1} \right\rangle$ such that

$$\rho(\mathsf{cat}(A, B)) = \left\langle s_0 + s_0^B,\ s_1\,,\ \ldots\ ,\ s_{n-1} \right\rangle$$

and

$$\langle i \rangle\ \psi\ \mathsf{cat}(A, B) = \begin{cases} \langle i \rangle\ \psi\ A & \text{if } i < s_0 \\ \langle i - s_0 \rangle\ \psi\ B & \text{otherwise} \end{cases}$$

hold. In order to simplify notation along the paper, we relax the definition of cat to assume its second argument is automatically reshaped to fit the shape requirements. We use this only in cases when the required reshape operation does not require computing a non-trivial shape argument.

$\forall t \in \mathsf{Fin}\, s_0$, $\mathsf{cat}(\triangle(t, A), \triangledown(t, A)) = A$ holds;

- the family of rotation functions θ_j that take a positive (respectively negative) integer o and rotate A by $|o|$ elements to the "right" (respectively left) along axis j. Formally, we have

$\forall j \in \mathsf{Fin}\ (\mathsf{dim}(A))$, $o \in \{o \in \mathbb{Z} : |o| \in \mathsf{Fin}\, s_j\}$,

$$\rho(o\ \theta_j\ A) = \rho(A)$$

and

$$i \psi (o \theta_j A) = \begin{cases} \mathsf{cat}(\triangle(-o, i \psi A), \triangledown(-o, i \psi A)) & \text{if } 0 \leq o \\ \mathsf{cat}(\triangledown(o, i \psi A), \triangle(o, i \psi A)) & \text{otherwise;} \end{cases}$$

where i is a partial index of length j into A.

3.4.2 Relevant MoA Operations at the ONF level

The following core operations are used throughout the paper at the ONF level:

- the family of dimension lifting operations lift_j that take two natural numbers d, q with $(d, q) \in \{(d, q) : d \cdot q = s_j\}$ and split the j^{th} axis of A into two shape components. More specifically,

  $$\mathsf{lift}_j(d, q, A) = \mathsf{reshape}\left(\langle s_0, \ldots, s_{j-1}, d, q, \ldots, s_{n-1}\rangle, A\right)$$

  holds. Dimension lifting is syntactic sugar for a specific reshaping operation. The intent is to use dimension lifting when the goal is to distribute computations below axis j across d computation loci. To the best of the knowledge of the authors, it is the first time a formal definition for dimension lifting in MoA is stated in the literature. We later give a formal definition for dimension lifting compatible with padding operations in Definition 10;

- the flattening function rav, which flattens an array into a unidimensional array, i.e.

  $$\mathsf{rav}\, A = \mathsf{reshape}(\langle \Pi(\rho(A))\rangle, A).$$

  We use rav to transport A into its corresponding linear representation in memory;

- the mapping function γ that takes a shape s with $\Pi s = \Pi(\rho(A))$ and a total index into s and produces the corresponding index into $\mathsf{rav}\, A$. Since $\mathsf{rav}\, A$ is 1-dimensional, the equation

  $$i \psi A = (\mathsf{rav}\, A)[\gamma(\rho(A), i)]$$

  holds for any total index i into A. Intuitively, γ transforms indexing operations into an abstract array representation of A into one that takes into account its concrete memory layout;

- the range function ι that given a positive integer n returns a 1-dimensional array containing the elements of $\mathsf{Fin}\, n$ in ascending order.

We recall informally the ψ-correspondence theorem [130]: $\forall k \in \mathsf{Fin}\, (\mathsf{dim}(A))$ and an index i of length k into A,

$$i \psi A = (\mathsf{rav}\, A)[\gamma(i, \langle s_0, \ldots, s_{k-1}\rangle) \cdot \mathit{stride} + \iota\, \mathit{stride}]$$

holds, with $\mathit{stride} = \Pi(\langle s_k, \ldots, s_{n-1}\rangle)$, and $+$ is the elementwise addition operation with implicit broadcast semantics.

Formal definitions for the operations described above can be found in Mullin's original work [127].

3.5 Memory Layout in the MoA

In the rest of the paper, we use a row-major memory layout. Our running example will be the DNF expression

$$\mathsf{expr} = ((1\,\theta_0\ \mathsf{Arr}) + (-1\,\theta_0\ \mathsf{Arr}))$$

where + is the elementwise addition operator. For the rest of the paper, we set

$$\mathsf{Arr} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{pmatrix},$$

where $\rho(\mathsf{Arr}) = \langle 6, 4\rangle$. More illustratively,

$$\mathsf{expr} = \begin{pmatrix} 21 & 22 & 23 & 24 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \end{pmatrix} + \begin{pmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ 1 & 2 & 3 & 4 \end{pmatrix} = \begin{pmatrix} 26 & 28 & 30 & 32 \\ 10 & 12 & 14 & 16 \\ 18 & 20 & 22 & 24 \\ 26 & 28 & 30 & 32 \\ 34 & 36 & 38 & 40 \\ 18 & 20 & 22 & 24 \end{pmatrix}.$$

The array expression expr is representative of one step of a PDE solver, as considered in the authors' previous work [36] and by Burrows et al. [28].

We can use the ψ-correspondence theorem to transform expr into the following ONF expression:

$$\begin{aligned} &\forall i \in \mathsf{Fin}\,6,\ \langle i\rangle\ \psi\ ((1\,\theta_0\ \mathsf{Arr}) + (-1\,\theta_0\ \mathsf{Arr})) \\ &\quad = (\mathsf{rav}\,\mathsf{Arr})[\gamma(\langle (i+1) \bmod 6\rangle, \langle 6\rangle) \cdot 4 + \iota 4] + \\ &\quad\ \ (\mathsf{rav}\,\mathsf{Arr})[\gamma(\langle (i-1) \bmod 6\rangle, \langle 6\rangle) \cdot 4 + \iota 4]. \end{aligned}$$

We follow up by unfolding γ:

$$\begin{aligned} &= (\mathsf{rav}\,\mathsf{Arr})[((i+1) \bmod 6) \cdot 4 + \iota 4] + \\ &\ \ (\mathsf{rav}\,\mathsf{Arr})[((i-1) \bmod 6) \cdot 4 + \iota 4]. \end{aligned}$$

By unfolding rav and turning ι into a loop we get the following generic program:

$$\begin{aligned} &\forall i \in \mathsf{Fin}\,6, j \in \mathsf{Fin}\,4, \\ &\quad (\mathsf{rav}\,\mathsf{Arr})[((i+1) \bmod 6) \cdot 4 + j] + (\mathsf{rav}\,\mathsf{Arr})[((i-1) \bmod 6) \cdot 4 + j]. \end{aligned}$$

The above program is written assuming, implicitly, that the target architecture is a single-core processor. We can use dimension lifting to establish a correspondence between the shape of the array and a different underlying hardware architecture.

Consider an architecture that consists of two single-core processors. We apply dimension lifting on axis 1 of Arr, to create the array

$$\mathsf{Arr}' = \mathsf{lift}_1(2, 2, A) = \mathsf{reshape}\,(\langle 6, 2, 2\rangle, A)$$

where axis 1 corresponds to the number of available cores. We get the following:

$$
\begin{aligned}
&\forall i \in \mathsf{Fin}\,6,\ j \in \mathsf{Fin}\,2,\\
&\quad \langle i,j \rangle \ \psi\ ((1\ \theta_0\ \mathsf{Arr}') + (-1\ \theta_0\ \mathsf{Arr}'))\\
&= (\mathsf{rav}\,\mathsf{Arr}')[\gamma(\langle((i+1) \bmod 6)\, j\rangle, \langle 6,2\rangle) \cdot 2 + \iota 2] +\\
&\quad (\mathsf{rav}\,\mathsf{Arr}')[\gamma(\langle((i-1) \bmod 6)\, j\rangle, \langle 6,2\rangle) \cdot 2 + \iota 2]\\
&= (\mathsf{rav}\,\mathsf{Arr}')[(((i+1) \bmod 6) \cdot 2 + j) \cdot 2 + \iota 2] +\\
&\quad (\mathsf{rav}\,\mathsf{Arr}')[(((i-1) \bmod 6) \cdot 2 + j) \cdot 2 + \iota 2].
\end{aligned}
$$

This reduces to the following generic program:

$$
\begin{aligned}
&\forall i \in \mathsf{Fin}\,6, j \in \mathsf{Fin}\,2, k \in \mathsf{Fin}\,2,\\
&\quad (\mathsf{rav}\,\mathsf{Arr}')[((i+1) \bmod 6) \cdot 4 \cdot 2 + j \cdot 2 + k] +\\
&\quad (\mathsf{rav}\,\mathsf{Arr}')[((i-1) \bmod 6) \cdot 4 \cdot 2 + j \cdot 2 + k].
\end{aligned}
$$

The programs before and after dimension lifting above are equivalent except for their different looping structures—they are adapted to two different hardware architectures.

Dimension lifting can be carried out across any axis (or on several axes simultaneously). The choice of axes should be guided by both the memory hierarchy and the operations involved in the expression. For example, the rotations above are applied on axis 0; dimension lifting on this axis would not allow perfectly splitting the memory between the two processors.

The example involves a modulo operation on the index. This is an expensive operation even on modern hardware [117]. We describe below a *circular padding* operation on DNF expressions that introduces data redundancy into arrays.

In Section 3.5.1, we define circular padding operations and observe how they eliminate the need for *modulo* operations in a single-core setting for our running example. In Section 3.5.2 we generalize these operations and put them to work to reduce the need for inter-process communication in a distributed computation setting for our running example.

3.5.1 Case of One Core and Constant Memory Access Cost

Padding an array is prepending or appending data to it. For our purposes, we want these data to be specific slices of the array itself. Here we introduce notation to define the circular prepending (referred to as *left padding*) and circular appending (referred to as *right padding*) operations in MoA.

Notation 1. Given an n-dimensional array A and an integer $i \in \mathsf{Fin}\ n$, we use the shorthand notation K_i to represent the index of length i into A whose j^{th} component is bound to variable k_j, i.e.

$$K_i = \langle k_0\,,\ \ldots\ ,\ k_{i-1}\rangle\,.$$

To further simplify the notation of indexing, we also introduce the shorthand notation

$$A_{K_i} = A_{\langle k_0\,,\ \ldots\ ,\ k_{i-1}\rangle} = \langle k_0\,,\ \ldots\ ,\ k_{i-1}\rangle\ \psi\ A.$$

Definition 2. Let A be an n-dimensional array with shape $\langle s_0\,,\,\ldots\,,\,s_{n-1}\rangle$. We can specify an n-dimensional slice B of A by annotating each component s_i of its shape with the (inclusive) beginning of the slice $b_i \in \mathsf{Fin}\ (s_i+1)$ and the (exclusive) end $e_i \in \mathsf{Fin}\ (s_i+1)$, with $b_i \leq e_i$. Concretely, we have

$$\rho(B) = \langle e_0 - b_0\,,\,\ldots\,,\,e_{n-1} - b_{n-1}\rangle$$

and

$$B_{\langle k_0\,,\,\ldots\,,\,k_{n-1}\rangle} = A_{\langle k_0+b_0\,,\,\ldots\,,\,k_{n-1}+b_{n-1}\rangle}.$$

In the rest of this section, we attach a slice annotation to each of our arrays. We write

$$\rho_{\text{ann}}(A) = \left\langle s_0^{b_0,e_0},\,\ldots\,,\,s_{n-1}^{b_{n-1},e_{n-1}}\right\rangle$$

the projection function that extracts both the shape and the slice annotation from an array.

We care about making a difference between padded and unpadded arrays. In the following, it is assumed that if A is unpadded, it carries the slice annotation such that $\forall i \in \mathsf{Fin}\ (\mathsf{dim}(A))$, $b_i = 0$, $e_i = s_i$.

Definition 3. Given an array A such that

$$\rho_{\text{ann}}(A) = \left\langle s_0^{b_0,e_0},\,\ldots\,,\,s_{n-1}^{b_{n-1},e_{n-1}}\right\rangle$$

and an integer $i \in \mathsf{Fin}\ n$ we define the right circular padding operation on axis i as padr_i such that

$$\mathsf{padr}_i(A)_{K_i} = \mathsf{cat}(A_{K_i}, A_{\langle k_0\,,\,\ldots\,,\,k_{i-1}\,,\,b_i+s_i-e_i\rangle})$$

for j, k_j integers such that $0 \leq j < i$, $0 \leq k_j < s_j$. Notice that this uses our overloaded definition of cat, where the second parameter is implicitly reshaped as needed. The shape of the result is given by

$$\rho_{\text{ann}}(\mathsf{padr}_i(A)) = \left\langle s_0^{b_0,e_0},\,\ldots\,,\,(s_i+1)^{b_i,e_i},\,\ldots\,,\,s_{n-1}^{b_{n-1},e_{n-1}}\right\rangle.$$

As an example, assume $\rho_{\text{ann}}(A) = \left\langle 2^{0,2}, 2^{0,2}\right\rangle$,

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

then

$$\mathsf{padr}_0(A)_{K_0} = \mathsf{padr}_0(A)_{\langle\rangle} = \mathsf{cat}(A_{\langle\rangle}, A_{0+2-2}) = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 1 & 2 \end{pmatrix}.$$

In the same way, we define the left circular padding operation on axis i as padl_i such that

$$\mathsf{padl}_i(A)_{K_i} = \mathsf{cat}(A_{\langle k_0\,,\,\ldots\,,\,k_{i-1}\,,\,e_i-b_i-1\rangle}, A_{K_i}),$$

whose shape is given by

$$\rho_{\text{ann}}(\mathsf{padl}_i(A)) = \left\langle s_0^{b_0,e_0},\,\ldots\,,\,(s_i+1)^{b_i+1,e_i+1},\,\ldots\,,\,s_{n-1}^{b_{n-1},e_{n-1}}\right\rangle.$$

Finally, we write padl^{-1} (respectively padr^{-1}) for the left inverse function of padl (respectively padr).

Recall our running example

$$\mathsf{expr} = ((1\ \theta_0\ \mathsf{Arr}) + (-1\ \theta_0\ \mathsf{Arr})),$$

which we reduced to

$\forall i \in \mathsf{Fin}\ 6, j \in \mathsf{Fin}\ 4,$
$\quad (\mathsf{rav}\ \mathsf{Arr})[((i+1) \bmod 6) \cdot 4 + j] + (\mathsf{rav}\ \mathsf{Arr})[((i-1) \bmod 6) \cdot 4 + j].$

using the ψ-correspondence theorem while assuming a single-core processor as the underlying hardware architecture.

We apply Definition 3 to obtain the following:

$$\begin{aligned}\mathsf{expr} &= \mathsf{padr}_0^{-1}(\mathsf{padl}_0^{-1}(\mathsf{padl}_0(\mathsf{padr}_0(\mathsf{expr})))) \\ &= \mathsf{padr}_0^{-1}(\mathsf{padl}_0^{-1}(\mathsf{padl}_0(\mathsf{padr}_0((1\ \theta_0\ \mathsf{Arr}) + (-1\ \theta_0\ \mathsf{Arr})))))).\end{aligned}$$

Proposition 4. For any axis i, padl_i and padr_i commute, i.e.

$$\mathsf{padl}_i \circ \mathsf{padr}_i = \mathsf{padr}_i \circ \mathsf{padl}_i.$$

Proposition 4 follows from the associativity of cat.

Proposition 5. Let A be an array without right padding, i.e. an array such that

$$\rho_{\mathrm{ann}}(A) = \left\langle s_0^{b_0,s_0}, \ldots, s_{n-1}^{b_{n-1},s_{n-1}} \right\rangle.$$

For all $i \in \mathsf{Fin}\ n$ and $m \in \mathsf{Fin}\ (s_i + 1)$,

$$\mathsf{padr}_i^m(A)_{K_i} = \mathsf{cat}(A_{K_i}, \triangle(m, \triangledown(b_i, A_{K_i}))).$$

In the same way, for A an array without left padding, we have

$$\mathsf{padl}_i^m(A)_{K_i} = \mathsf{cat}(\triangledown(e_i - b_i - m, \triangle(e_i - b_i, A_{K_i})), A_{K_i}).$$

Both cases of can be shown by induction on m.

Proposition 6. Let B, C be n-dimensional MoA expressions with $\rho(B) = \rho(C)$ and $\oplus$ a binary map operation. Then $\forall i \in \mathsf{Fin}\ n$, padr_i is distributive over $\oplus$, i.e.

$$\mathsf{padr}_i(B) \oplus \mathsf{padr}_i(C) = \mathsf{padr}_i(B \oplus C)$$

holds. Similarly, for padl_i

$$\mathsf{padl}_i(B) \oplus \mathsf{padl}_i(C) = \mathsf{padl}_i(B \oplus C)$$

holds. This idea is easily extensible to n-ary map operations.

Proof. To improve readability, we write

$$T_i = \langle k_0\,,\ \ldots\ ,\ k_{i-1}\,,\ b_i + s_i - e_i \rangle\,.$$

In the case of padr_i, since $\oplus$ is a binary map operation, we have:

$$\begin{aligned}
&(\mathsf{padr}_i(B) \oplus \mathsf{padr}_i(C))_{K_i} = \mathsf{cat}(B_{K_i}, B_{T_i}) \oplus \mathsf{cat}(C_{K_i}, C_{T_i}) \\
\Leftrightarrow\ &(\mathsf{padr}_i(B) \oplus \mathsf{padr}_i(C))_{K_i} = \mathsf{cat}(B_{K_i} \oplus C_{K_i}, B_{T_i} \oplus C_{T_i}) \\
\Leftrightarrow\ &(\mathsf{padr}_i(B) \oplus \mathsf{padr}_i(C))_{K_i} = \mathsf{padr}_i(B \oplus C)_{K_i} \\
\Leftrightarrow\ &\mathsf{padr}_i(B) \oplus \mathsf{padr}_i(C) = \mathsf{padr}_i(B \oplus C).
\end{aligned}$$

The case for padl_i can be shown using the same reasoning.

□

By applying Proposition 6 in our example, we get:

$$\mathsf{expr} = \mathsf{padr}_0^{-1}(\mathsf{padl}_0^{-1}(\mathsf{padl}_0(\mathsf{padr}_0(1\ \theta_0\ \mathsf{Arr})) + \mathsf{padl}_0(\mathsf{padr}_0(-1\ \theta_0\ \mathsf{Arr})))).$$

Proposition 7. Let B be a n-dimensional unpadded MoA expression, j an axis of B and r an integer. Then, on any axis i of B, we have that

$$r\ \theta_j\ B = \mathsf{padr}_i^{-m_2}(\mathsf{padl}_i^{-m_1}(r\ \theta_j\ \mathsf{padl}_i^{m_1}(\mathsf{padr}_i^{m_2}(B))))$$

holds if either one of the following cases holds:

(i) $j \neq i$;

(ii) $r = 0$;

(iii) $r < 0$ and $m_2 \geq |r|$;

(iv) $r > 0$ and $m_1 \geq r$.

Proof. Cases (i) and (ii) are trivial. In case (i), padding does not affect the rotation. In case (ii), $0\ \theta_j\ B = B$ holds. We thus want to prove that

$$i = j, r < 0, m_2 \geq |r|$$

implies

$$r\ \theta_j\ B = \mathsf{padr}_i^{-m_2}(\mathsf{padl}_i^{-m_1}(r\ \theta_j\ \mathsf{padl}_i^{m_1}(\mathsf{padr}_i^{m_2}(B))))$$

Using Proposition 5, we can write $\mathsf{padl}_i^{m_1}(\mathsf{padr}_i^{m_2}(B))$ as an array A such that

$$A_{K_i} = \mathsf{cat}(\mathrm{p}, \mathsf{cat}(B_{K_i}, \triangle(m_2, \triangledown(b_i, B_{K_i}))))$$

where p represents the left-padding of the array. Since B is originally unpadded, we rewrite A as:

$$A_{K_i} = \mathsf{cat}(\mathrm{p}, \mathsf{cat}(B_{K_i}, \triangle(m_2, \triangledown(0, B_{K_i})))) = \mathsf{cat}(\mathrm{p}, \mathsf{cat}(B_{K_i}, \triangle(m_2, B_{K_i}))).$$

Since $r < 0$, we have:

$$(r\ \theta_i\ A)_{K_i} = \mathsf{cat}(\triangledown(|r|, \mathsf{cat}(\mathrm{p}, \mathsf{cat}(B_{K_i}, \triangle(m_2, B_{K_i})))),$$

$$\begin{aligned} &\triangle(|r|, \mathsf{cat}(\mathrm{p}, \mathsf{cat}(B_{K_i}, \triangle(m_2, B_{K_i})))) \\ =\ &\mathsf{cat}(\triangledown(|r|, \mathsf{cat}(\mathsf{cat}(\mathrm{p}, B_{K_i}), \triangle(m_2, B_{K_i}))), \\ &\triangle(|r|, \mathsf{cat}(\mathrm{p}, \mathsf{cat}(B_{K_i}, \triangle(m_2, B_{K_i}))))). \end{aligned}$$

Using Proposition 4, we get

$$\mathsf{padr}_i^{-m_2}(\mathsf{padl}_i^{-m_1}(r\ \theta_i\ A)) = \mathsf{padl}_i^{-m_1}(\mathsf{padr}_i^{-m_2}(r\ \theta_i\ A)),$$

and since $m_2 \geq |r|$, we have

$$\begin{aligned} (\mathsf{padr}_i^{-m_2}(r\ \theta_i\ A))_{K_i} &= \triangledown(|r|, \mathsf{cat}(\mathsf{cat}(\mathrm{p}, B_{K_i}), \triangle(m_2 - (m_2 - |r|), B_{K_i}))) \\ &= \triangledown(|r|, \mathsf{cat}(\mathsf{cat}(\mathrm{p}, B_{K_i}), \triangle(|r|, B_{K_i}))). \end{aligned}$$

We can thus write:

$$\begin{aligned} (\mathsf{padl}_i^{-m_1}(\mathsf{padr}_i^{-m_2}(r\ \theta_i\ A)))_{K_i} &= \triangledown(m_1, \triangledown(|r|, \mathsf{cat}(\mathsf{cat}(\mathrm{p}, B_{K_i}), \triangle(|r|, B_{K_i})))) \\ &= \triangledown(m_1 + |r|, \mathsf{cat}(\mathsf{cat}(\mathrm{p}, B_{K_i}), \triangle(|r|, B_{K_i}))). \end{aligned}$$

Since r is a valid rotation offset in B, we can write

$$\begin{aligned} \triangledown(m_1 + |r|, \mathsf{cat}(\mathsf{cat}(\mathrm{p}, B_{K_i}), \triangle(|r|, B_{K_i}))) &= \mathsf{cat}(\triangledown(m_1 + |r|, \mathsf{cat}(\mathrm{p}, B_{K_i})), \triangle(|r|, B_{K_i})) \\ &= \mathsf{cat}(\triangledown(|r|, B_{K_i}), \triangle(|r|, B_{K_i})) \\ &= (r\ \theta_i\ B)_{K_i} \end{aligned}$$

and thus

$$r\ \theta_j\ B = \mathsf{padr}_i^{-m_2}(\mathsf{padl}_i^{-m_1}(r\ \theta_j\ \mathsf{padl}_i^{m_1}(\mathsf{padr}_i^{m_2}(B))))$$

by function extensionality.
The proof for case 4 follows the same pattern as case 3 on the opposite side of the array. □

Proposition 8. Given an array expression A with

$$\rho(A) = \langle s_0\ ,\ \ldots\ ,\ s_{n-1} \rangle\ ,$$

some $i \in \mathsf{Fin}\ n$, and two positive integers m_1, m_2,

$$\mathsf{padl}_i^{m_1}(\mathsf{padr}_i^{m_2}(A))_{\langle k_0\ ,\ \ldots\ ,\ m_1+k_i\ ,\ \ldots\ ,\ k_{n-1} \rangle} = A_{\langle k_0\ ,\ \ldots\ ,\ k_{n-1} \rangle}$$

with $\forall j, k_j \in \mathsf{Fin}\ s_j$ holds.

Proposition 8 can be proven by definition of padding.

The insight behind Proposition 8 is that the content of A is maintained in the padded expression B, but that the evaluation of A within B may behave differently due to the shift in indexing and duplication of data brought by padding.

We call the values m_1 and m_2 in Proposition 8 *consumption speed* for a given axis i in the following, and define a function speed_i on expressions such that

$$\mathsf{speed}_i(B) = (m_1, m_2).$$

Note that in practice, the choice of m_1 and m_2 is made by the user of MoA based on their goals.

We apply Proposition 7, and get:

$$\mathsf{expr} = \mathsf{padr}_0^{-1}(\mathsf{padl}_0^{-1}((1\ \theta_0\ \mathsf{padl}_0(\mathsf{padr}_0(\mathsf{Arr}))) + (-1\ \theta_0\ \mathsf{padl}_0(\mathsf{padr}_0(\mathsf{Arr}))))).$$

In order to get rid of the mod operation in the ONF expression we built for expr, we create a new array Arr′ defined by

$$\mathsf{Arr}' = \mathsf{padl}_0(\mathsf{padr}_0(\mathsf{Arr}))$$

From Definition 3, we have that

$$\rho_{\text{ann}}(\mathsf{Arr}') = \left\langle 8^{1,7}, 4 \right\rangle$$

and

$$\mathsf{Arr}' = \left(\begin{array}{cccc} 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \end{array}\right).$$

We rewrite our example with the new definition of Arr′, and get

$$\mathsf{expr} = \mathsf{padr}_0^{-1}(\mathsf{padl}_0^{-1}((1\ \theta_0\ \mathsf{Arr}') + (-1\ \theta_0\ \mathsf{Arr}'))).$$

We can now transform it to ONF again. The bounds of the relevant indices i and j are given by Proposition 8. We obtain the following:

$$\begin{aligned} &\forall i \in \{i \in \mathsf{Fin}\ 8\ :\ 1 \le i < 7\}, \\ &\quad \left\langle i, j \right\rangle\ \psi\ ((1\ \theta_0\ \mathsf{Arr}') + (-1\ \theta_0\ \mathsf{Arr}')) \\ &\equiv (\mathsf{rav}\ \mathsf{Arr}')[\gamma(\langle i+1 \rangle ; \langle 8 \rangle) \times 4 + \iota 4] + (\mathsf{rav}\ \mathsf{Arr}')[\gamma(\langle i-1 \rangle ; \langle 8 \rangle) \times 4 + \iota 4]. \end{aligned}$$

We then apply γ, rav, and turn ι into a loop to get the following generic program:

$$\begin{aligned} &\forall i \in \{i \in \mathsf{Fin}\ 8\ :\ 1 \le i < 7\}, j \in \mathsf{Fin}\ 4, \\ &\quad (\mathsf{rav}\ \mathsf{Arr}')[(i+1) \times 4 + j] + \mathcal{A}'[(i-1) \times 4 + j]. \end{aligned}$$

Finally, we apply the composition $\mathsf{padr}_0^{-1} \circ \mathsf{padl}_0^{-1}$ and retrieve the exact same result as we would have gotten by directly evaluating expr. Notice that thanks to the notion of consumption speed, we avoided performing the computation on irrelevant indices. In the end, both of the expressions had 6 loop iterations, but we managed to get rid of the expensive mod operation by adding data redundancy into Arr.

3.5.2 Case of Non-Uniform Memory Access

Consider now that expr is embedded within a loop and must be executed several times. Then, in order to avoid the mod operation at each iteration, the array must be padded at each iteration as well.

Considering hardware and software implementations, it is reasonable to investigate a case in which the application of a big padding operation $p = \mathsf{padl}_i^n \circ \mathsf{padr}_i^m$ for some natural numbers n, m, i at a given point in the program is cheaper than applying parts of p in different parts of the program.

For example, if the padding operation depends on inter-process communication, it is usually significantly cheaper to open one socket and send four elements than to open two sockets each sending two elements (each opened connection probably also requiring synchronization of some sort, etc). We however consider unpadding to have negligible cost.

To reduce the number of loci where a padding operation is required and to use the resulting padding efficiently, we need to define a slightly more complex padding function as well as further notation. We also need to define the notion of *padding exhaustion*.

Informally, padding exhaustion corresponds to reaching a state where there is not enough unconsumed padding left to evaluate the expression and achieve our goals of using padding. Padding exhaustion is related to consumption speed. In our example, padding is exhausted when both of the equivalent evaluation strategies stated in Proposition 8 fail to get rid of all mod operations.

Ideally, this state is reached at the end of the computation of all the expressions; if reached in the middle of execution, the padding must be replenished to proceed.

Definition 9. Let A be an n-dimensional array with shape $\langle s_0, \ldots, s_{n-1}\rangle$. We can specify a $2n$-dimensional reshaping D of A by annotating each component s_i of its shape with a divisor d_i such that $s_i \equiv 0 \mod d_i$. We then reshape A into an array D such that

$$\rho(D) = \langle d_0, q_0, \ldots, d_{n-1}, q_{n-1}\rangle$$

where $q_i = \frac{s_i}{d_i}$. Assume we have

$$\rho_{\text{ann}}(D) = \left\langle d_0^{0,d_0}, q_0^{b_0,e_0}, \ldots, d_{n-1}^{0,d_{n-1}}, q_{n-1}^{b_{n-1},e_{n-1}}\right\rangle.$$

Then, we can specify an n-dimensional slice B of A such that

$$\rho(B) = \langle (e_0 - b_0) \times d_0, \ldots, (e_{n-1} - b_{n-1}) \times d_{n-1}\rangle$$

and

$$B_{K_n} = D_{\left\langle \frac{k_0}{e_0-b_0}, k_0 \bmod (e_0-b_0), \ldots, \frac{k_{n-1}}{e_{n-1}-b_{n-1}}, k_{n-1} \bmod (e_{n-1}-b_{n-1})\right\rangle}.$$

We write

$$\rho_{\text{ann+}}(A) = \left\langle s_0^{d_0,b_0,e_0}, \ldots, s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}}\right\rangle$$

the projection function that extracts this "distributed slice annotation" from the array.

In the following definitions, we reuse the present definition of q_i.

Definition 10. Let A be an array such that

$$\rho_{\text{ann+}}(A) = \left\langle s_0^{d_0,b_0,e_0}, \ldots, s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}} \right\rangle,$$

We define the padding-compatible dimension lifting operation on axis i $\mathsf{liftp}_i(A) = B$ such that B has $n+1$ dimensions,

$$\rho_{\text{ann+}}(B) = \left\langle s_0^{d_0,b_0,e_0}, \ldots, d_i, q_i^{1,b_i,e_i}, \ldots, s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}} \right\rangle,$$

and

$$B_{K_i} = \triangle(q_i, \triangledown(k_{i-1} \times q_i, A_{K_{i-1}})).$$

liftp is a version of lift that extracts its parameters from and modifies the "distributed slice annotation" of the array.

Definition 11. Consider an array A such that

$$\rho_{\text{ann+}}(A) = \left\langle s_0^{d_0,b_0,e_0}, \ldots, s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}} \right\rangle.$$

In a MoA setting without any notion of padding, any array is implicitly annotated with $d_i = 1$, $b_i = 0$ and $e_i = s_i$ on any given axis i. To properly use liftp as it is defined above, we do the following: assuming $b_i = 0$ and $e_i = s_i$ for a given axis i of A, we define the prelift operation on that axis for any $d \in \{d : s_i \equiv 0 \bmod d\}$ as $\mathsf{prelift}_i(d, A) = B$,

$$\rho_{\text{ann+}}(B) = \left\langle s_0^{d_0,b_0,e_0}, \ldots, s_i^{d,0,\frac{s_i}{d}}, \ldots, s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}} \right\rangle$$

and

$$B_{K_n} = A_{K_n}.$$

The precondition on $\mathsf{prelift}_i$ means that it can only be applied to arrays that are unpadded on axis i.

Recall once again our running example

$$\mathsf{expr} = ((1\ \theta_0\ \mathsf{Arr}) + (-1\ \theta_0\ \mathsf{Arr})),$$

which we previously reduced to

$\forall i \in \mathsf{Fin}\ 6, j \in \mathsf{Fin}\ 4,$

$$(\mathsf{rav\ Arr})[((i+1) \bmod 6) \cdot 4 + j] + (\mathsf{rav\ Arr})[((i-1) \bmod 6) \cdot 4 + j].$$

using the ψ-correspondence theorem while assuming a single-core processor as the underlying hardware architecture.

We wish to distribute the computation over two machines. We will achieve this through a combination of dimension lifting and padding. To distribute the computation over two machines, it is natural to perform dimension lifting along the 0^{th} axis of Arr, taking $d_0 = 2$. We thus start out by creating a new array Arr′ such that

$$\mathsf{Arr}' = \mathsf{prelift}_0(2, \mathsf{Arr}).$$

From Definition 11, we have that

$$\rho_{\text{ann+}}(\mathsf{Arr}') = \left\langle 6^{2,0,3}, 4 \right\rangle.$$

Since $\mathsf{prelift}_0$ does not modify the layout of the array it operates on in any way, we have

$$\mathsf{expr} = (1\ \theta_0\ \mathsf{Arr}') + (-1\ \theta_0\ \mathsf{Arr}').$$

Definition 12. Given an array A with shape

$$\left\langle s_0^{d_0,b_0,e_0}, \ldots, s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}} \right\rangle$$

and an integer $i \in \mathsf{Fin}\, n$ we define the right pre-dimension lifting padding operation on axis i as $\mathsf{dpadr}_i(A) = R$ such that

$$\mathsf{liftp}_i(R)_{K_i} = \mathsf{cat}(\triangle(q_i, \triangledown(k_{i-1} \times q_i, A_{K_{i-1}})), A_{\langle k_0, \ldots, ((k_{i-1}+2)\times q_i + b_i - e_i) \bmod s_i \rangle})$$

for $j \in \mathsf{Fin}\, i$ and $k_j \in \mathsf{Fin}\, s_j$.

Note that we consider operations on the axis i to be done in $\mathsf{Fin}\, n$, e.g. for $i = 0$, we have $k_{i-1} = k_{n-1}$.

The shape $\rho_{\mathrm{ann}+}(R)$ is as in:

$$\left\langle s_0^{d_0,b_0,e_0}, \ldots, (s_i + d_i)^{d_i,b_i,e_i}, \ldots, s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}} \right\rangle.$$

In the same way, we define the left pre-dimension lifting padding operation on axis i as $\mathsf{dpadl}_i(A) = L$ such that

$$\mathsf{liftp}_i(L)_{K_i} = \mathsf{cat}(A_{\langle k_0, \ldots, ((k_{i-1}-1)\times q_i + e_i - b_i - 1) \bmod s_i \rangle}, \triangle(q_i, \triangledown(k_{i-1} \times q_i, A_{K_{i-1}})))$$

The shape of L is as in:

$$\left\langle s_0^{d_0,b_0,e_0}, \ldots, (s_i + d_i)^{d_i,b_i+1,e_i+1}, \ldots, s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}} \right\rangle.$$

Finally, we call dpadl^{-1} (respectively dpadr^{-1}) the left inverse function of dpadl (respectively dpadr).

We are now ready to start padding Arr'. In this case, we would like the two workers to only communicate at the start and at the end of the computation. To do that, we need to provide each machine with enough padding to do all of the required computations in one go.

Similarly to the approach we took in the previous section, we create a new array Arr'' such that

$$\mathsf{Arr}'' = \mathsf{dpadl}_0(\mathsf{dpadr}_0(\mathsf{Arr}')).$$

From Definition 12, we have that

$$\rho_{\mathrm{ann}+}(\mathsf{Arr}'') = \left\langle 10^{2,1,4}, 4 \right\rangle$$

and

$$\mathsf{Arr}'' = \left(\begin{array}{cccc} 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \end{array}\right).$$

Finally, we create an array $\mathsf{Arr}^{+} = \mathsf{liftp}_0(\mathsf{Arr}'')$. From Definition 10, we have:

$$\rho_{\mathrm{ann}+}(\mathsf{Arr}^{+}) = \langle 2, 5^{1,4}, 4 \rangle$$

and

$$\mathsf{Arr}^{+}_{\langle 0 \rangle} = \left(\begin{array}{cccc} 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \end{array}\right),$$

$$\mathsf{Arr}^{+}_{\langle 1 \rangle} = \left(\begin{array}{cccc} 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \end{array}\right).$$

By definition of dpadr and dpadl, we have the following:

$$\begin{aligned} \mathsf{expr} &= \mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{dpadl}_0(\mathsf{dpadr}_0(\mathsf{expr})))) \\ &= \mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}(\mathsf{liftp}_0(\mathsf{dpadl}_0(\mathsf{dpadr}_0(\mathsf{expr})))))). \end{aligned}$$

Proposition 13. For a given axis i, the functions dpadl_i and dpadr_i commute, i.e.

$$\mathsf{dpadl}_i \circ \mathsf{dpadr}_i = \mathsf{dpadr}_i \circ \mathsf{dpadl}_i .$$

Proposition 13 can be proven using the associativity of cat.

Proposition 14. Let A be an array and $i \in \mathsf{Fin}\,(\mathsf{dim}(A))$. Let

$$\begin{aligned} R &= \mathsf{dpadr}_i(A) \\ L &= \mathsf{dpadl}_i(A) \end{aligned}$$

then

$$\mathsf{padr}_i^{-1}(\mathsf{liftp}_i(R)_{K_i}) = \triangle(q_i, \triangledown(k_{i-1} \times q_i, A_{K_{i-1}})) \tag{3.1}$$

$$\mathsf{padl}_i^{-1}(\mathsf{liftp}_i(L)_{K_i}) = \triangle(q_i, \triangledown(k_{i-1} \times q_i, A_{K_{i-1}})) \tag{3.2}$$

hold.

Proof. We give a proof for Equation 3.1:

$$\begin{aligned} \mathsf{padr}_i^{-1}(\mathsf{liftp}_i(R)_{K_i}) &= \mathsf{padr}_i^{-1}(\mathsf{cat}(\triangle(q_i, \triangledown(k_{i-1} \times q_i, A_{K_{i-1}})), A_{\langle k_0\,,\,\ldots\,,\,((k_{i-1}+2)\times q_i + b_i - e_i) \bmod s_i \rangle})) \\ &= \triangle(q_i, \triangledown(k_{i-1} \times q_i, A_{K_{i-1}})). \end{aligned}$$

The proof for Equation 3.2 follows the same pattern as the above. □

Informally, Proposition 14 tells us that for a given array A' resulting from padding and dimension lifting an array A on an axis i, unpadding and concatenating all the subarrays resulting from the dimension lifting operation is the same as concatenating them and *unpadding* the result.

Proposition 15. Let A be an array without right padding on its i^{th} axis, i.e. an array such that

$$\rho_{\text{ann}+}(A) = \left\langle s_0^{d_0,b_0,e_0}, \ldots, s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}} \right\rangle$$

with $e_i = s_i$, $i \in \mathsf{Fin}\, n$. Given an integer $m \in \mathsf{Fin}\,(s_i + 1)$, let $A' = \mathsf{dpadr}_i^m(A)$. Then, the following holds:

$$\begin{aligned}\mathsf{liftp}_i(A')_{K_i} = \mathsf{cat}(&\triangle(q_i, \triangledown(k_{i-1} \times q_i, A_{K_{i-1}})),\\ &\triangle(m, \triangledown(((k_{i-1}+1) \times q_i + b_i) \bmod s_i, A_{K_{i-1}})))\end{aligned}$$

In the same way, for A an array without left padding and $A'' = \mathsf{dpadl}_i^m(A)$, then

$$\begin{aligned}\mathsf{liftp}_i(A'')_{K_i} = \mathsf{cat}(&\triangledown(e_i - b_i - m, \triangle(e_i - b_i, \triangledown((k_{i-1} - 1) \times q_i \bmod s_i, A_{K_{i-1}}))),\\ &\triangle(q_i, \triangledown(k_{i-1} \times q_i, A_{K_{i-1}})))\end{aligned}$$

holds.

A proof for Proposition 15 may be written using induction on m, like the proof for Proposition 5.

Proposition 16. Let B, C be n-dimensional MoA expressions with identical shapes and $\oplus$ a binary map operation. Then, liftp_i distributes over $\oplus$, i.e.

$$\mathsf{liftp}_i(B \oplus C) = \mathsf{liftp}_i(B) \oplus \mathsf{liftp}(C) \tag{3.3}$$

for any axis i of B and C. This idea is trivially extensible to n-ary map operations.

Proposition 16 can be proven using the definition of liftp, and the shape-conserving property of n-ary map operations.

Proposition 17. Let B, C be n-dimensional MoA expressions with identical shapes and $\oplus$ a binary map operation. Then, dpadr_i distributes over $\oplus$, i.e.

$$\mathsf{dpadr}_i(B \oplus C) = \mathsf{dpadr}_i(B) \oplus \mathsf{dpadr}_i(C). \tag{3.4}$$

Similarly, we have that

$$\mathsf{dpadl}_i(B \oplus C) = \mathsf{dpadl}_i(B) \oplus \mathsf{dpadl}_i(C). \tag{3.5}$$

This idea is trivially extensible to n-ary map operations.

Proof. To improve readability, we write

$$T_i = \left\langle k_0, \ldots, ((k_{i-1} + 2) \times q_i + b_i - e_i) \bmod s_i \right\rangle.$$

Since $\oplus$ is a binary map operation, we have:

$$\begin{aligned}&(\mathsf{liftp}_i(\mathsf{dpadr}_i(B)) \oplus \mathsf{liftp}_i(\mathsf{dpadr}_i(C)))_{K_i} =\\ &\mathsf{cat}(\triangle(q_i, \triangledown(k_{i-1} \times q_i, B_{K_{i-1}})), B_{T_i}) \oplus \mathsf{cat}(\triangle(q_i, \triangledown(k_{i-1} \times q_i, C_{K_{i-1}})), C_{T_i})\\ \Leftrightarrow\ &(\mathsf{liftp}_i(\mathsf{dpadr}_i(B)) \oplus \mathsf{liftp}_i(\mathsf{dpadr}_i(C)))_{K_i} =\\ &\mathsf{cat}(\triangle(q_i, \triangledown(k_{i-1} \times q_i, B_{K_{i-1}})) \oplus \triangle(q_i, \triangledown(k_{i-1} \times q_i, C_{K_{i-1}})), B_{T_i} \oplus C_{T_i})\\ \Leftrightarrow\ &(\mathsf{liftp}_i(\mathsf{dpadr}_i(B)) \oplus \mathsf{liftp}_i(\mathsf{dpadr}_i(C)))_{K_i} = (\mathsf{liftp}_i(\mathsf{dpadr}_i(B \oplus C)))_{K_i}\\ \Leftrightarrow\ &\mathsf{liftp}_i(\mathsf{dpadr}_i(B)) \oplus \mathsf{liftp}_i(\mathsf{dpadr}_i(C)) = \mathsf{liftp}_i(\mathsf{dpadr}_i(B \oplus C))\\ \Leftrightarrow\ &\mathsf{dpadr}_i(B) \oplus \mathsf{dpadr}_i(C) = \mathsf{dpadr}_i(B \oplus C).\end{aligned}$$

The proof for Equation 3.5 follows the same pattern as above. Since it does not provide any additional insight, we do not develop it here. □

By applying Propositions 16 and 17 in our example, we get:

$$\begin{aligned}\mathsf{expr} = \mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}(\mathsf{liftp}_0(\mathsf{dpadl}_0(\mathsf{dpadr}_0(1\ \theta_0\ \mathsf{Arr}))) + \\ \mathsf{liftp}_0(\mathsf{dpadl}_0(\mathsf{dpadr}_0(-1\ \theta_0\ \mathsf{Arr})))))).\end{aligned}$$

Proposition 18. Let A be a n-dimensional unpadded MoA expression, j an axis of A and r an integer. Then, on any axis i of A, we have that

$$\begin{aligned}r\ \theta_j\ A &= \mathsf{dpadr}_i^{-m_2}(\mathsf{dpadl}_i^{-m_1}(\mathsf{liftp}_i^{-1}(\mathsf{liftp}_i(\mathsf{dpadl}_i^{m_1}(\mathsf{dpadr}_i^{m_2}(r\ \theta_j\ A)))))) \\ &= \mathsf{dpadr}_i^{-m_2}(\mathsf{dpadl}_i^{-m_1}(\mathsf{liftp}_i^{-1}(r\ \theta_j\ \mathsf{liftp}_i(\mathsf{dpadl}_i^{m_1}(\mathsf{dpadr}_i^{m_2}(A)))))).\end{aligned}$$

holds if either one of the following cases holds:

(i) $j \neq i$;

(ii) $r = 0$;

(iii) $r < 0$ and $m_2 \geq |r|$;

(iv) $r > 0$ and $m_1 \geq r$.

The proof for Proposition 18 follows the same pattern as the proof for Proposition 7.

We can now apply Proposition 18 in our example, and get:

$$\begin{aligned}\mathsf{expr} &= \mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}((1\ \theta_0\ \mathsf{liftp}_0(\mathsf{dpadl}_0(\mathsf{dpadr}_0(\mathsf{Arr})))) + \\ &\qquad (-1\ \theta_0\ \mathsf{liftp}_0(\mathsf{dpadl}_0(\mathsf{dpadr}_0(\mathsf{Arr}))))))) \\ &= \mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}((1\ \theta_0\ \mathsf{Arr}^+) + (-1\ \theta_0\ \mathsf{Arr}^+)))).\end{aligned}$$

We can now transform the resulting expression expr to ONF for each machine. The bounds of i and j are once again given by Proposition 8. Thus, for $c \in \{0, 1\}$, we have the following:

$$\begin{aligned}&\forall i \in \{i \in \mathsf{Fin}\,5\ :\ 1 \leq i < 4\}, \\ &\quad \langle i \rangle\ \psi\ ((1\ \theta_0\ \mathsf{Arr}^+_{\langle c \rangle}) + (-1\ \theta_0\ \mathsf{Arr}^+_{\langle c \rangle})) \\ &\equiv (\mathsf{rav}\,\mathsf{Arr}^+_{\langle c \rangle})[\gamma(\langle i+1 \rangle ; \langle 5 \rangle) \times 4 + \iota 4] + (\mathsf{rav}\,\mathsf{Arr}^+_{\langle c \rangle})[\gamma(\langle i-1 \rangle ; \langle 5 \rangle) \times 4 + \iota 4].\end{aligned}$$

We then apply γ, rav and turn ι into a loop, and we get the following generic program:

$$\begin{aligned}&\forall i \in \{i \in \mathsf{Fin}\,5\ :\ 1 \leq i < 4\}, j \in \mathsf{Fin}\,4, \\ &\quad (\mathsf{rav}\,\mathsf{Arr}^+_{\langle c \rangle})[(i+1) \times 4 + j] + (\mathsf{rav}\,\mathsf{Arr}^+_{\langle c \rangle})[(i-1) \times 4 + j].\end{aligned}$$

Finally, we join the results using liftp_0^{-1} and apply dpadl_0^{-1} and dpadr_0^{-1} to obtain the same result as we would have gotten evaluating expr directly. Moreover, in this case, both expressions had the same number of loop iterations, and *exactly* all the padding was consumed in the computation.

In practice however, what we studied above corresponds to a single step of the PDE solver. Assume the same scenario as above, except that the solver actually runs this step two times. For simplicity, we generalize expr to a function step such that, for any array A, $\mathsf{step}(A) = \mathsf{expr}[\mathsf{Arr} := A]$. Two sequential executions of step would then be written as $\mathsf{step}^2(A)$.

According to Proposition 8, we have $\mathsf{speed}_0(\mathsf{expr}) = (1, 1)$. Thus, for the padding to last two steps and thus avoid padding exhaustion before the end of the full computation, we need to pad the 0^{th} axis of A m_l times on the left and m_r times on the right, where m_l and m_r are given by:

$$(m_l, m_r) = 2 \times \mathsf{speed}_0(\mathsf{expr}) = 2 \times (1, 1) = (2, 2).$$

We start again by creating an array Arr'' such that

$$\mathsf{Arr}'' = \mathsf{dpadl}_0^2(\mathsf{dpadr}_0^2(\mathsf{Arr}')).$$

From Definition 12, we have that

$$\rho_{\text{ann+}}(\mathsf{Arr}'') = \left\langle 14^{2,2,5}\ 4 \right\rangle$$

and

$$\mathsf{Arr}'' = \left(\begin{array}{cccc}
17 & 18 & 19 & 20 \\
21 & 22 & 23 & 24 \\
\hline
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
\hline
13 & 14 & 15 & 16 \\
17 & 18 & 19 & 20 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
\hline
13 & 14 & 15 & 16 \\
17 & 18 & 19 & 20 \\
21 & 22 & 23 & 24 \\
\hline
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8
\end{array}\right).$$

We create an array $\mathsf{Arr}^{++} = \mathsf{liftp}(\mathsf{Arr}'')$. From Definition 10:

$$\rho_{\text{ann+}}(\mathsf{Arr}^{++}) = \left\langle 2, 7^{2,5}, 4 \right\rangle$$

and

$$\mathsf{Arr}^{++}_{\langle 0 \rangle} = \left(\begin{array}{cccc}
17 & 18 & 19 & 20 \\
21 & 22 & 23 & 24 \\
\hline
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
\hline
13 & 14 & 15 & 16 \\
17 & 18 & 19 & 20
\end{array}\right),$$

$$\mathsf{Arr}^{++}_{\langle 1 \rangle} = \left(\begin{array}{cccc}
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
\hline
13 & 14 & 15 & 16 \\
17 & 18 & 19 & 20 \\
21 & 22 & 23 & 24 \\
\hline
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8
\end{array}\right).$$

Once again, using Proposition 17 and Proposition 18, we get

$$\mathsf{step}^2(\mathsf{Arr}) = \mathsf{dpadr}_0^{-2}(\mathsf{dpadl}_0^{-2}(\mathsf{liftp}_0^{-1}(\mathsf{step}^2(\mathsf{Arr}^{++})))).$$

We can now transform the resulting expression expr to ONF for each machine. The bounds of i and j are once again given by Proposition 8. Thus, for $c \in \{0, 1\}$, we have the following:

$$\begin{aligned}
&\forall i \in \{i \in \mathsf{Fin}\,7 \;:\; 1 \leq i < 6\},\\
&\quad \langle i \rangle \;\psi\; ((1\;\theta_0\;\mathsf{Arr}^{++}_{\langle c \rangle}) + (-1\;\theta_0\;\mathsf{Arr}^{++}_{\langle c \rangle}))\\
&\equiv (\mathsf{rav}\,\mathsf{Arr}^{++}_{\langle c \rangle})[\gamma(\langle i+1 \rangle ; \langle 7 \rangle) \times 4 + \iota 4] + (\mathsf{rav}\,\mathsf{Arr}^{++}_{\langle c \rangle})[\gamma(\langle i-1 \rangle ; \langle 7 \rangle) \times 4 + \iota 4].
\end{aligned}$$

We once again apply γ, rav and turn ι into a loop, and we get the following generic program:

$$\begin{aligned}
&\forall i \in \{i \in \mathsf{Fin}\,7 \;:\; 1 \leq i < 6\}, j \in \mathsf{Fin}\,4,\\
&\quad (\mathsf{rav}\,\mathsf{Arr}^{++}_{\langle c \rangle})[(i+1) \times 4 + j] + (\mathsf{rav}\,\mathsf{Arr}^{++}_{\langle c \rangle})[(i-1) \times 4 + j].
\end{aligned}$$

At that point, we can rewrite our expression as such:

$$\begin{aligned}
\mathsf{step}^2(\mathsf{Arr}) &= \mathsf{dpadr}_0^{-2}(\mathsf{dpadl}_0^{-2}(\mathsf{liftp}_0^{-1}(\mathsf{step}^2(\mathsf{Arr}^{++}))))\\
&= \mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}(\mathsf{step}(\mathsf{liftp}_0(\mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}(\mathsf{step}(\mathsf{Arr}^{++}))))))))).
\end{aligned}$$

But here, as given by Proposition 14, applying

$$\mathsf{liftp}_0 \circ \mathsf{dpadr}_0^{-1} \circ \mathsf{dpadl}_0^{-1} \circ \mathsf{liftp}_0^{-1}$$

to $\mathsf{step}(\mathsf{Arr}^{++})$ is equivalent to applying padr_0^{-1} and padl_0^{-1} once to both $\mathsf{Arr}^{++}_{\langle 0 \rangle}$ and $\mathsf{Arr}^{++}_{\langle 1 \rangle}$. As a result, for $c \in \{0, 1\}$,

$$\rho_{\mathrm{ann}+}(\mathsf{padr}_0^{-1}(\mathsf{padl}_0^{-1}(\mathsf{Arr}^{++}_{\langle c \rangle}))) = \left\langle 5^{1,4}, 4 \right\rangle.$$

The rest of the computation follows the single step distributed case presented above. Note that in this case four intermediate rows of the result were computed twice (once on each machine), resulting in four additional outer loop iterations compared to the equivalent single machine unpadded two-step case. Thus, getting rid of inter-process communication involved both data redundancy and duplicated calculations. Whether performing calculations several times instead of exchanging states between different computation loci is beneficial, must be determined based on hardware-dependent cost functions.

3.6 Experiments

We extended the scenario depicted in Section 3.5.1 to our implementation of a PDE solver using a $(-1, 0, 1)$ stencil along every axis; that is, at every derivation step the left and right padding operation are applied once along the specified axis. The memory overhead of such padding is roughly 0.4% in our example. The execution times of 50 derivation steps given different padding parameters are gathered in Table 3.2.

We see a performance improvement on CPU 1, 2, and 4 between the original code in which no padding was applied and the cases with padding on either axis. The difference is particularly striking on CPU 2 and 4, where the program runs roughly twice as fast when padding is applied.

Table 3.2: Execution time (in seconds) of a 3-dimensional PDE solver C implementation compiled with GCC 8.2.0 with different padding parameters on a single core. CPU 1: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz; CPU 2: AMD EPYC 7601 32-Core; CPU 3: Intel(R) Xeon(R) Silver 4112 CPU @ 2.60GHz; and CPU 4: ThunderX2 CN9980. The code of the experiments is at https://github.com/mathematics-of-arrays/padding-in-the-mathematics-of-arrays.

	No padding	Padding axis 2	Padding axis 3
CPU 1	225.74	168.59	167.84
CPU 2	299.42	119.61	120.12
CPU 3	172.71	160.51	192.70
CPU 4	660.53	347.47	357.32

This large difference seems to indicate that padding allows the compiler to perform better optimizations on these architectures. This analysis is corroborated by the output of *perf stat*: on CPU 2 and 4, the runs without padding execute 2 to 2.5 times as many instructions as their padded counterparts.

On CPU 3, padding the last axis makes execution slower. Looking at the output of *perf stat* tells us that both padded programs execute roughly 87% as many instructions as the unpadded one. When the last axis is padded, the number of executed instructions per cycle (IPC) drops to 81%. That run should last roughly $\frac{87}{81} = 1.07$ times as long as the unpadded one. This is in line with our measurements. One possible explanation is that GCC does not properly take into account the costs of the instructions involved in the computation.

The number of instructions run on CPU 1 and CPU 3 are close. However, the drop in IPC is much smaller on CPU 1, resulting in a slight performance improvement.

It is hard to quantify the impact of the padding on data locality and cache line usage. The percentage of measured cache misses in all the programs is roughly the same for all three runs for a given architecture.

These experiments further confirm the need for a vehicle for easy exploration of codes that implement different memory layouts.

Further work is needed to explore tiling in this setting. This is because tiling requires reorganizing data within arrays using *transpose*, which we did not study here.

3.7 Conclusion

We showed that MoA provides the required building blocks to discuss padding as well as data distribution given an arbitrary architecture. It is thus well-suited to explore the space of optimal computations for array expressions at a high level of abstraction. Along the way, we built two examples demonstrating exactly how to use these notions to optimize stencil computations. Our approach could be implemented as a compiler optimization to automatically rewrite array expressions based on hardware and known operational costs. We expect future work to focus both on better quantifying the benefit of using this approach instead of existing solutions and on implementing

MoA and its properties using proof assistants. For the latter, effort is already underway at https://github.com/mathematics-of-arrays/moa-formalization.

Acknowledgements We thank Jonathan Prieto-Cubides and Wrya Kadir for their helpful feedback on early drafts of the paper. We also give our thanks to Jeremy Gibbons and our anonymous reviewers for their constructive and insightful comments of our paper. The research presented in this paper has benefited from the Experiment Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

Paper 4

P³ Problem and Magnolia Language: Specializing Array Computations for Emerging Architectures[I]

Benjamin Chetioui[a], Marius Kleppe Larnøy[a], Jaakko Järvi[b], Magne Haveraaen[a], Lenore Mullin[c]

[a]Department of Informatics, University of Bergen, Norway, [b]Department of Computing, University of Turku, Finland, [c]College of Engineering and Applied Sciences, University at Albany, SUNY, USA, [c]

Abstract

The problem of producing portable high-performance computing (HPC) software that is cheap to develop and maintain is called the P³ (performance, portability, productivity) problem. Good solutions to the P³ problem have been achieved when the performance profiles of the target machines have been similar. The variety of HPC architectures is, however, large and can be expected to grow larger. Software for HPC therefore needs to be highly adaptable, and there is a pressing need to provide developers with tools to produce software that can target machines with vastly different profiles.

Multi-dimensional array manipulation constitutes a core component of numerous numerical methods, such as finite difference solvers of Partial Differential Equations (PDEs). The efficiency of these computations is tightly connected to traversing and distributing array data in a hardware-friendly way. The Mathematics of Arrays (MoA) allows for formally reasoning about array computations and enables systematic transformations of array-based programs, e.g. to use data layouts that fit to a specific architecture.

This paper shows a general methodology for solving the P³ problem in domains that are well-explored using Magnolia, a language designed to embody generic programming.

[I]Benjamin Chetioui, Marius Larnøy, Jaakko Järvi, Magne Haveraaen, and Lenore Mullin. P³ problem and Magnolia language: Specializing array computations for emerging architectures. *Frontiers in Computer Science*, 4, 2022. doi:10.3389/fcomp.2022.931312

The Magnolia programmer can restrict the semantic properties of abstract generic types and operations by defining so-called axioms. Axioms can be used to produce tests for concrete implementations of specifications, for formal verification, or to perform semantics-preserving program transformations.

We leverage Magnolia's semantic specification facilities to extend the Magnolia compiler with a term rewriting system. We implement MoA's transformation rules in Magnolia, and demonstrate through a case study on a finite difference solver of PDEs how our rewriting system allows exploring the space of possible optimizations.

4.1 Introduction

The quest for higher performance fuels innovation on hardware architectures; we have seen a wide variety of high-performance computing (HPC) architectures in the past and can expect new ones to keep appearing. Long-lived and successful HPC software must thus be highly adaptable, adjustable to different memory hierarchies and changing intra- and interprocess communication hardware.

The problem of producing portable HPC software that is easy, or at least not unreasonably difficult, to develop and maintain is called the P³ (performance, portability, productivity) problem. Good solutions to the P³ problem have been achieved when the performance profiles of the target machines have been similar [177]. As more new hardware architectures emerge, there is a pressing need to provide developers with tools to produce such software for targets with vastly different profiles. This includes architectures within Wolfe's P³ machine performance model (CPUs, GPUs, or other accelerators, possibly distributed) [177] but also those that do not (e.g., Groq's Tensor Streaming Processor [5]).

Multidimensional array manipulation is at the core of numerous numerical methods. The topic of optimizing the performance of array computations is therefore extremely relevant to the P³ problem. We have previously explored the Mathematics of Arrays (MoA) formalism [127] as a tool to optimize array computations for different hardware architectures, first through their *Denotational Normal Form* (DNF) [36] and then through their *Operational Forms* (OFs) [37]. A thorough mathematical understanding of a given domain is key to enabling domain-specific semantic-preserving rewrites—and therefore optimizations.

The portability and productivity pillars of P³ are both strongly related to the notion of code reuse. Portability as meant here is the ability to run the same code with high performance on different machines. Productivity means that applications can be developed and maintained with a reasonable and predicable effort. Research unequivocally shows that productivity increases through reuse [23; 56; 138]. Generic programming has proven to be an effective method of constructing libraries of reusable software components. The Magnolia programming language [14] is designed as an embodiment of generic programming [38]. It allows the flexible intermixing of specifications and implementations. Specifications can additionally be restricted by semantic requirements (called *axioms*) in the form of assertions. These axioms can be used for testing [20], but also for optimization when used as directed rewrite rules, in the case of equational or conditional equational axioms [17].

4.1.1 Schedules as Hardware Abstractions

In their 2012 paper on Halide, Ragan-Kelley et al. introduce the term *schedule* to refer to decisions about storage and about the order of computations in a program [149]. The insight is that the essence of an algorithm is distinct from its schedule—allowing the advent of a programming model where both kinds of computations are not anymore intertwined but instead expressed independently from each other.

Stepanov-style generic programming abstracts algorithms and data structures by specifying minimum syntactic and semantic requirements on instantiations. Said differently, the types and operations underlying a generic implementation are only characterized by the part of their observable behavior that is relevant to the generic algorithm.

When observed through the lens of generic programming, a schedule is an abstraction for the kind of hardware architecture underlying the computations. We consider only the information about the hardware that is relevant for executing our algorithm efficiently: how computations should be ordered, and how data should be stored. Similar hardware architectures are then valid instantiations for the same schedule.

Scheduling, in the case of array computations, relates particularly to the access patterns of the arrays. As a motivating example, consider an array program running on a single CPU with memory, the classical model of a computer. We may have three standard traversal patterns for computations over our arrays:

1. a row-major traversal;
2. a column-major traversal;
3. a tiled traversal.

While the original algorithm can be expressed without making any assumption about the underlying hardware, the choice of a particular hardware will dictate which traversal pattern is most efficient. In other cases, the choice of a particular schedule may be desirable. E.g., on hardware consisting of several distributed CPUs connected through some communication network, we may want the schedule to handle inter-CPU communication using MPI. If each one of these CPUs is connected to several GPUs, we may also want the schedule to load data on and off the GPUs as needed. Such choices will affect the desired data layout, and consequently the data access patterns so as to match the distribution of the data. These changes will have to be reflected in the presentation of the algorithm.

The execution time for an algorithm adapted to its schedule may be dramatically shorter than for an algorithm exhibiting inadapted data access patterns. While an algorithm and its schedule can be expressed independently, choices in the latter may affect what is an appropriate expression of the former, and vice versa. Our approach uses rewriting technology to adapt a unique algorithm to adequately exploit the data traversal pattern of a schedule, and underlying hardware characteristics.

Throughout the rest of the paper, we view schedules as hardware abstractions. This view is fully compatible with Ragan-Kelley et al.'s definition of schedules, but conveys our intent more accurately.

4.1.2 Contribution and Structure of the Paper

The contribution of this paper is a general methodology for solving the P³ problem in well-explored domains that keeps the essence of the algorithm separate from its schedule. We define well-explored domains as those for which significant domain-specific knowledge and a mathematical formalization exist. We perform a case study on a Partial Differential Equation (PDE) solver based on Finite Difference Methods (FDM). We extend the Magnolia compiler with code generation and term rewriting facilities based on axioms. We implement our solver in Magnolia, using MoA as an underlying basis for the code, giving us both generic and hardware-specific formally verified optimization rules—also directly implemented in Magnolia.

The paper is structured as follows. Section 4.2 provides necessary background on Magnolia. Section 4.3 describes our methodology in detail, and illustrates it with a PDE solver based on FDM. Section 4.4 reflects on our work and ties it together with relevant related work. All the code is available as an example in the repository for magnoliac [35] (see the *examples/pde* folder at `https://github.com/magnolia-lang/magnolia-lang/tree/base-program`).

4.2 Background

4.2.1 Magnolia

The phrase *generic programming* has over decades of programming language development come to have a variety of intepretations, depending on the main type of genericity considered. Gibbons gives a taxonomy of interpretations [62]. Stepanov-style generic programming [49] corresponds to what Gibbons calls *genericity by property*, where one describes data structures and algorithms in terms of syntactic and semantic requirements. This is the essence of Stepanov's and Musser's *concepts* [137]. They are the direct inspiration behind C++0x concepts [76]; the C++20 concepts are a scaled back realization of those that only allow syntactic requirements on instantiations. (In this latter case, we talk of *genericity by structure*.)

Magnolia is a programming language designed as an embodiment of Stepanov-style generic programming [14]. Magnolia code is structured into modules that mix abstract specifications of operations and their concrete implementations flexibly, following the work of Goguen and Burstall on the theory of institutions [68]. The language does not offer any primitive types aside from predicates: every data structure is implemented in a configurable host programming language. As of today, Magnolia can target C++ and Python [35]. Our prior work coins the term *genericity by host language* to refer to this axis of parameterization, in the style of Gibbons' taxonomy [38]. Composite operations can be implemented in Magnolia, while the base types and operations, including loop structures, are implemented in the host language. The programmer can freely decide where to set the boundary between the operations implemented in Magnolia, and those implemented in the base library written in the host language—depending on what is more convenient. An appropriate choice of underlying data structures results in code that is as performant as if implemented directly in the host language [38]. Because the axiom formalism is semantically compatible with the program code, Magnolia avoids the semantic gap common in approaches to formal software verification [153].

A Magnolia **signature** declares types and operations. A **signature** can be augmented with **axiom**s

that restrict the properties of its types and operations: the resulting module is a **concept**. An **implementation** allows the same declarations as a **signature**, but also (generic) implementations for the declared operations. The last kind of module in Magnolia is a **program**, a specific kind of **implementation** in which all the specified operations and types are matched with implementations. Crucially, types and operations in a **program** are no longer generic but instead fully concrete. An **implementation** can be a model of a **concept**; a **concept** can also be a model of another **concept**. Such modeling relations can be specified directly in Magnolia using the **satisfaction** language construct.

Magnolia operations can be **function**s, **procedure**s, and **predicate**s. The arguments to functions and predicates are immutable, while arguments to procedures are given explicit modes: **obs** (read-only), **upd** (read/write), and **out** (write-only, and the procedure promises to initialize the argument). Procedures do not return a value. Calls to procedures are prefixed with the **call** keyword.

Listing 4.1 gives a general overview of the different kinds of Magnolia modules. We first specify the signature of a magma (a set `T` with a closed binary operation `bop`). By asserting the associativity property on a magma, we get a semigroup. The `ConcretePartialSemigroup` implementation describes an external C++ API providing a guarded multiplication operator over integer matrices, where the guard is intended to ensure the argument matrices have compatible dimensions. `ExampleProgram` builds `multiplyThreeMatrices` off of the primitive building blocks provided by `ConcretePartialSemigroup`. The `ExampleProgramHasMulPartialSemigroup` satisfaction relation indicates that `ExampleProgram` satisfies the semigroup axioms, with the set of integer matrices and guarded multiplication on it. The guard provided on the multiplication operation in the left-hand side of the satisfaction is propagated to the right-hand side. The resulting satisfaction relation thus asserts that the `ExampleProgram` has a partial semigroup structure. A block of renamings (`[ T => IntMatrix, bop => _*_ ]`) is applied to `Semigroup`. Magnolia's renamings allow changing the names of types and operations in places where a module is "opened". This is a powerful feature which allows normalizing the names exposed by modules when we open them in a given scope, independently of how their types and operations were initially named.

Listing 4.1: Multiplying three matrices in Magnolia.

```
signature Magma = {
  type T;
  function bop(a: T, b: T): T;
}

concept Semigroup = {
  use Magma;
  axiom associativity(a: T, b: T, c: T) {
    assert bop(bop(a, b), c) == bop(a, bop(b, c));
  }
}

implementation ConcretePartialSemigroup =
  external C++ lib.int_matrices {
    type Nat;
    type IntMatrix;

    predicate lhsNrowsIsRhsNcols(
```

```
      m1: IntMatrix, m2: IntMatrix);

    function _*_(m1: IntMatrix, m2: IntMatrix): IntMatrix
      guard lhsNrowsIsRhsNcols(m1, m2);
  }

program ExampleProgram = {
  use ConcretePartialSemigroup;

  function multiplyThreeMatrices(
    A: IntMatrix, B: IntMatrix, C: IntMatrix): IntMatrix =
      A * B * C;
}

// The guard on _*_ in ExampleProgram is lifted to the
// specification on Semigroup in the left-hand side---this
// satisfaction relation thus states that ExampleProgram has
// a partial semigroup structure.
satisfaction ExampleProgramHasMulPartialSemigroup =
  ExampleProgram models
    Semigroup[ T => IntMatrix, bop => _*_ ];
```

4.2.1.1 Exploiting Magnolia axioms

Concept axioms have previously found use as test oracles [20] and as generic optimization rules [17; 171]. We implement two module transformations called *rewrite* and *implement* in *magnoliac*, the Magnolia compiler under active development [35].

The *rewrite* transformation extracts all assertions of equations from a given concept, and uses them as directed rewrite rules within a target module expression. The maximum allowed number of applications of these directed rewrite rules is provided as an argument to the transformation. The rewrite rules can only be applied from left to right in the current implementation, and there is thus no need to specify how to orient them.

The *implement* transformation highlights a third possible use case for Magnolia axioms, i.e. code generation. The transformation extracts all the assertions of equations from a given concept where the left-hand side is a call to a declared function (or predicate) with pairwise distinct universally quantified arguments, and generates an implementation for the function where the body is the right-hand side of the assertion. Intuitively, an assertion with the properties we outlined describes the behavior of the function on the left-hand side at every point. Therefore, such assertions are not only a way to specify the intended behavior of a function, but also a way to derive an actual implementation for it in case one was not already provided.

The intuition behind *implement* is that it transforms a specification into an implementation. The *implement* transformation produces changes visible at the module level, while *rewrite* replaces expressions within already implemented operations. Figure 4.1 describes the grammar for the *rewrite* and *implement* transformations.

Figure 4.1: The grammar for the *rewrite* and *implement* module transformations in Magnolia.

```
transformation  ::= rewrite module-expr with module-expr int
                  | implement module-expr in module-expr
```


Consider the `multiplyThreeMatrices` function in Listing 4.1. The function is intended to multiply three matrices together—and its body `A * B * C` desugars to the expression `_*_(_*_(A, B), C)`. Due to the associativity property, the order in which the multiplications are executed does not matter when it comes to the correctness of the result. However, it matters a lot when it comes to performance: suppose `A` is of dimensions 100×2, `B` of dimensions 2×20, and `C` of dimensions 20×90. Executing `A * B` requires $100 \times 2 \times 20$ scalar multiplications, and executing `(A * B) * C` thus requires $100 \times 2 \times 20 + 100 \times 20 \times 90 = 184000$ scalar multiplications. On the other side, executing `B * C` requires $2 \times 20 \times 90$ scalar multiplications, and executing `A * (B * C)` requires executing $2 \times 20 \times 90 + 100 \times 2 \times 90 = 21600$ scalar multiplications, nearly ten times fewer.

Suppose that a developer wants to use the `multiplyThreeMatrices` function in their program. They care about efficiency, and know that the input matrices `A`, `B`, and `C` have the same dimensions as specified above. They can use the assertion provided in the associativity property of the `Semigroup` concept defined in Listing 4.1 as a rewrite rule in `multiplyThreeMatrices` to optimize the expression from `(A * B) * C` to `A * (B * C)`. Listing 4.2 shows how.

Listing 4.2: Demonstration of the Magnolia *rewrite* transformation.

```
program DevProgram = rewrite ExampleProgram
  with Semigroup[ bop => _*_, T => IntMatrix ] 1;
```

The Magnolia *rewrite* module transformation takes three arguments: the module on which to perform the rewrite (`ExampleProgram` in the example), the module from which to extract rewriting rules (`Semigroup` with some renamings applied in the example), and a maximum allowed number of rule applications (`1` in the example).

Here, `multiplyThreeMatrices` is a toy example, and defined directly in the **program** being transpiled—it would therefore be very easy to reimplement it manually. However, this is not always the case: the function one wants to transform could be very complicated, and hidden deep inside an external dependency. Without the ability to perform rewritings on functions that have been previously defined, the developer would have to write their own version of this function.

4.3 Methodology and Case Study

We describe here our proposed methodology for writing performant and portable code productively using the Magnolia programming language. Each step of this methodology is first described from a high-level perspective, and then concretely demonstrated for our PDE solver example. Figure 4.2 gives a graphical overview of the concrete steps we take to optimize the PDE solver example in the following.

4

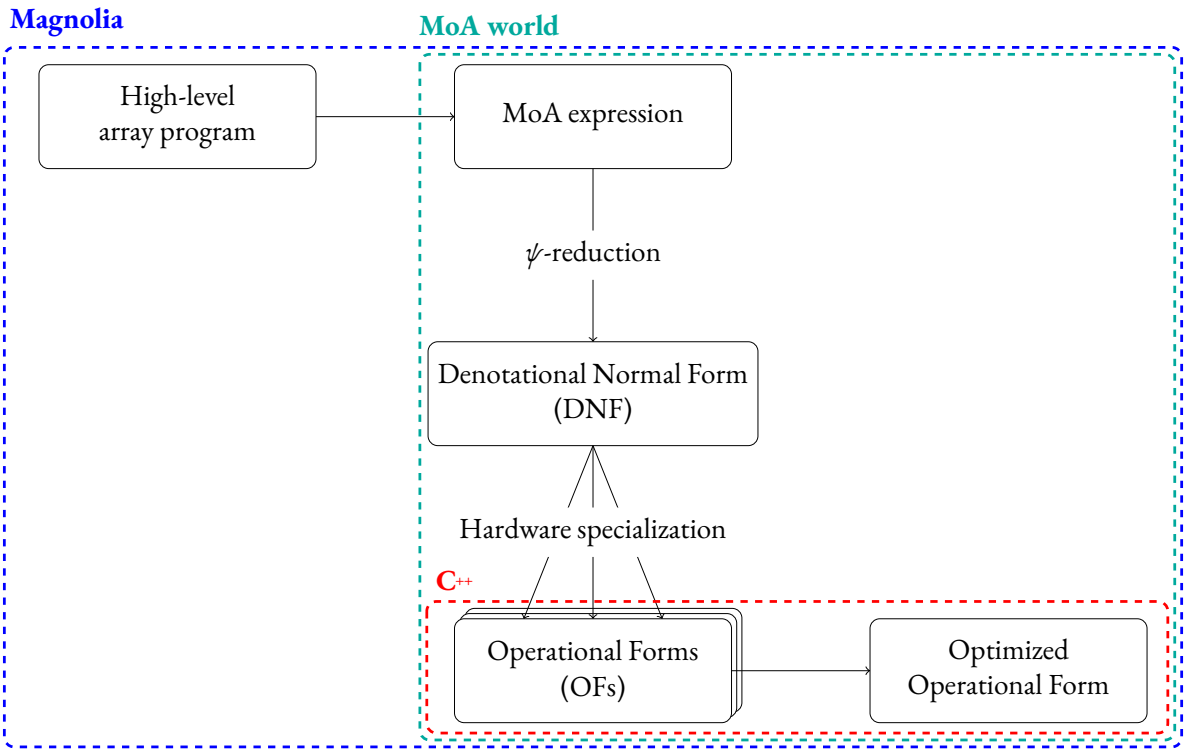

Figure 4.2: A graphical overview of the methodology presented in the paper. A high-level array program is passed as input, and translated to a corresponding MoA expression. This MoA expression is then normalized using a process known as ψ-reduction to produce the DNF. ψ-reduction gives hardware-independent rewriting rules on MoA expressions. By adding in knowledge about the specific hardware architecture underlying the computation, the DNF can be transformed into one of its OFs. This enables also hardware-specific optimization rules, which we can apply to the OF so as to produce an optimized OF. The program is initially written in Magnolia, and all the manipulation steps in the MoA world are done in Magnolia. The hardware specialization is implemented in the host language underlying the implementation (here C++), and the code contributing to the production of an optimized OF is thereby split between Magnolia and C++.

4.3.1 Identifying and Formalizing the Domain

The first step of our methodology is to build a thorough understanding of the targeted problem. We do that by identifying and formalizing the domain underlying the problem. Formalizing the domain gives us a mathematical understanding of the properties expected of the types and operations involved in the problem. These in turn allow specifying semantics-preserving optimization rules on them, whose correctness can be proven.

PDE solvers using FDM are based on multi-dimensional array computations. In 2018, Burrows et al. identified an array API for FDM solvers. In 2019, Chetioui et al. followed up with a formalization of the identified array API using MoA. We will first give an overview of PDE solvers as described by Burrows et al., and an introduction to the corresponding MoA theory. With this background in place, we will reimplement the PDE solver based on FDM from the work of Chetioui et al. [36], and implement hardware-agnostic and hardware-dependent rewriting rules. We show how they can be applied to our Magnolia program, and measure the resulting performance improvements.

4.3.1.1 PDEs

PDE solvers have many application areas. One example is numerical simulations of wind flow—e.g. for optimizing windmill positioning in large-scale wind farms.

Computing solutions to PDEs numerically requires discretizing continuous equations to a discrete domain. This approach to PDE solvers is often illustrated in the literature with Burgers' equation [27]. Equation 4.1 presents the equation in its coordinate-free form.

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = \nu \nabla^2 \vec{u}, \tag{4.1}$$

where $\vec{u}$ is velocity, t time, and ν the viscosity coefficient.

Assuming a 3D space, we can use a Cartesian coordinate system to rewrite Equation 4.1 as the following system of equations

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = \nu\frac{\partial^2 u}{\partial x^2} + \nu\frac{\partial^2 u}{\partial y^2} + \nu\frac{\partial^2 u}{\partial z^2} \tag{4.2}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = \nu\frac{\partial^2 v}{\partial x^2} + \nu\frac{\partial^2 v}{\partial y^2} + \nu\frac{\partial^2 v}{\partial z^2} \tag{4.3}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = \nu\frac{\partial^2 w}{\partial x^2} + \nu\frac{\partial^2 w}{\partial y^2} + \nu\frac{\partial^2 w}{\partial z^2}, \tag{4.4}$$

where $\vec{u} = (u, v, w)$.

To discretize the domain, we describe a $N_x \times N_y \times N_z$ grid of velocity values bounded by L_x (respectively L_y and L_z) on axis x (respectively y and z) such that the u component of the velocity at index (i, j, k)

and timestep n is given by

$$u^n_{i,j,k} = u(i\Delta x, j\Delta y, k\Delta z, n\Delta t), \tag{4.5}$$

with $\Delta x = \frac{L_x}{N_x}$, $\Delta y = \frac{L_y}{N_y}$, and $\Delta z = \frac{L_z}{N_z}$.

Similarly, the partial derivative of u in the x direction at index (i, j, k) and timestep n is

$$\frac{\partial u}{\partial x}(i\Delta x, j\Delta y, k\Delta z, n\Delta t). \tag{4.6}$$

In the FDM, we compute a partial derivative as a weighted sum of neighbouring grid points—where the weights are given by a list of factors called a *stencil*. The stencil is chosen by a numerical expert. This paper, following the work of Burrows et al. uses the numerical stencils $(-\frac{1}{2}, 0, \frac{1}{2})$ and $(1, -2, 1)$ for the first and second order partial derivatives respectively.

Given these stencils, the partial derivative of u in the x direction at index (i, j, k) and timestep n is approximated by

$$\frac{\partial u}{\partial x}(i\Delta x, j\Delta y, k\Delta z, n\Delta t) \approx \frac{1}{2\Delta x}(u^n_{i+1,j,k} - u^n_{i-1,j,k}), \tag{4.7}$$

which is accurate to $O((\Delta x)^2, \Delta t)$. Computing the partial derivative along the y (respectively z) axis follows a similar pattern, where j (respectively k) varies instead of i.

The standard 3D explicit finite difference approximation of Equation 4.2 is then given by

$$\begin{aligned} u^{n+1}_{i,j,k} = u^n_{i,j,k} &- \frac{\Delta t}{2\Delta x} u^n_{i,j,k}(u^n_{i+1,j,k} - u^n_{i-1,j,k}) + \frac{\nu\Delta t}{(\Delta x)^2}(u^n_{i+1,j,k} + u^n_{i-1,j,k} - 2u^n_{i,j,k}) \\ &- \frac{\Delta t}{2\Delta y} v^n_{i,j,k}(u^n_{i,j+1,k} - u^n_{i,j-1,k}) + \frac{\nu\Delta t}{(\Delta y)^2}(u^n_{i,j+1,k} + u^n_{i,j-1,k} - 2u^n_{i,j,k}) \\ &- \frac{\Delta t}{2\Delta z} w^n_{i,j,k}(u^n_{i,j,k+1} - u^n_{i,j,k-1}) + \frac{\nu\Delta t}{(\Delta z)^2}(u^n_{i,j,k+1} + u^n_{i,j,k-1} - 2u^n_{i,j,k}). \end{aligned}$$

The discretization of Equations 4.3 and 4.4 follows the same pattern.

The API of Burrows et al. is sufficient to compute numerical solutions to PDEs using FDM. It consists of elementwise arithmetic operations at the array level (+, -, *), a rotation operation on arrays (called "shift"), and arithmetic operations at the scalar level—corresponding to the behavior of the elementwise operations at each index of the array.

4.3.1.2 MoA

MoA [127; 130] is an algebra for describing operations on arrays. MoA distinguishes between two abstraction levels: the *Denotational Normal Form* (DNF), which describes an array by its shape together with a function describing its value at every index, and the *Operational Form* (OF) which describes it on the level of the memory layout. Programs written at the DNF level do not presume knowledge of a hardware architecture. Reasoning at the DNF level is thus completely hardware agnostic. By repeatedly applying a set of terminating rewrite rules, any MoA expression can be

reduced to its DNF [36; 132]—where the resulting array is described at each index by indexing into the input arrays and scalar-level operations.

Given information about the hardware architecture and the memory layout of the arrays, the ψ-correspondence theorem [130] allows transforming a DNF expression into a corresponding hardware-dependent OF—in which the access patterns on the array are described in terms of *start*, *stride*, and *length*.

We give an informal overview of some operations at the DNF and OF levels below. We refer the interested reader to previous work for formal definitions [37; 127].

DNF Operations The *dimension* of an array A is denoted $\mathsf{dim}(A)$. It corresponds to the number of axes of the array. For $\mathsf{dim}(A) = n$, the *shape* of A is an n-element vector $\rho(A) = \langle s_0, \ldots, s_{n-1} \rangle$ where s_i is the length of axis i. The total number of elements (or *size*) of A is given by the product of the shape, $\Pi\rho(A) = \Pi_{i=0}^{n-1} s_i$.

In the definitions below A stands for an arbitrary array with dimension n and shape as defined above. Further, we use the following array in examples:

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}.$$

Thus, $\rho(M) = \langle 3, 2 \rangle$.

The relevant MoA operations on the DNF level are:

- the indexing function ψ, which takes an index i into an array and returns the subarray at the indexed position. When i's length equals the dimension of the array, i is a *total* index. Otherwise, it is *partial*. $\langle\rangle \ \psi \ A = A$ holds. For our example, we have

$$\langle 2 \rangle \ \psi \ M = \langle 5, 6 \rangle,$$
$$\rho(\langle 2 \rangle \ \psi \ M) = \langle 2 \rangle.$$

- the $\mathsf{reshape}$ function that takes an array A and a shape s such that $\Pi s = \Pi\rho(A)$, and creates a new array with shape s containing the elements of A. Thus, $\rho(\mathrm{reshape}(s, A)) = s$ holds. For example,

$$\mathrm{reshape}(M, \langle 2, 3 \rangle) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

- a rotation function rotate that takes an array A, an axis j and an offset o, and shift A by o along its j^{th} axis. The shape is unchanged, i.e. $\rho(\mathsf{rotate}(A, j, o)) = \rho(A)$ holds. We give a few examples of how rotation behaves on axis 0 and 1 of M:

$$\mathsf{rotate}(M, 0, 1) = \begin{pmatrix} 5 & 6 \\ 1 & 2 \\ 3 & 4 \end{pmatrix},$$
$$\mathsf{rotate}(M, 0, -1) = \begin{pmatrix} 3 & 4 \\ 5 & 6 \\ 1 & 2 \end{pmatrix},$$

$$\mathsf{rotate}(M, 1, 1) = \begin{pmatrix} 2 & 1 \\ 4 & 3 \\ 6 & 5 \end{pmatrix}.$$

4

ψ-Reduction Mullin and Thibault described a rewriting system for MoA expressions at the DNF level, referred to as ψ-reduction. They conjectured that ψ-reduction is canonical (i.e. it is terminating and confluent)—and thus takes any MoA expression to its unique DNF. In their work on embedding Burrows et al.'s array API for FDM solvers in MoA, Chetioui et al. outline a rewriting system sufficient to transform a program based on this API to its DNF and show that this rewriting system is indeed canonical [36]. This draws appeal to MoA as a framework for the optimization of PDE solvers based on FDM. ψ-reduction essentially consists of rules that move indexing operations inwards—until eventually, the expression does not contain any collective operation, but consists only of indexing and scalar operations. As a consequence, it is guaranteed that the resulting array expression can be computed without the need to materialize any intermediate array. Because the rewriting system is canonical, another consequence is that the form in which we choose to express our computation is irrelevant: all equivalent expressions in the language of MoA reduce to the same DNF expression.

OF Operations At the OF level, we assume knowledge of the target architecture, and an intended memory layout of the array. The central MoA operations on the OF level are:

- the family of lifting operations lift_j that take two natural numbers d, q such that $d \cdot q = s_j$, and reshape A into the shape $\langle s_0, \ldots, s_{j-1}, d, q, s_{j+1}, \ldots, s_{n-1} \rangle$;
- the flattening function rav that transforms a multidimensional array into its linear representation in memory. Thus, $\rho(\mathsf{rav}(A)) = \langle \Pi\rho(A) \rangle$ holds;
- the mapping function γ, which takes a shape s with $\Pi s = \Pi\rho(A)$ and a total index into A and returns the corresponding index into $\mathsf{rav}(A)$. In this paper, we assume a row-major ordering.

The OF operations presented here are crucial to the theory of MoA. We thus include them for the sake of completion. These operations however do not appear explicitly in the development of our methodology.

4.3.1.3 Initial Magnolia Implementation

We implemented a PDE solver using the MoA array API. The implementation consists of four components:

1. a specification of the necessary MoA types and operations, with axioms asserting that they respect the relevant properties;
2. a foreign API exposing the core types and operations of the MoA specification;
3. an external implementation of the foreign API in a host language (C++);
4. an implementation of the PDE solver built upon the external MoA building blocks.

The ψ-calculus conflates arrays, indices, shapes, and scalars into a single array type. While convenient

in the formalism, we distinguish these types in our Magnolia implementation for ease of reasoning, and to leverage the language's type system to avoid programming errors.

Listing 4.3 shows the API from Burrows et al. in the language of MoA.

Listing 4.3: An array API for FDM solvers in Magnolia.

```
signature ArrayAPI = {
  type Array;
  type E;

  type Axis;
  type Index;
  type Offset;

  /* Scalar-Scalar operations */
  function _+_(lhs: E, rhs: E): E;
  function _-_(lhs: E, rhs: E): E;
  function _*_(lhs: E, rhs: E): E;
  function _/_(lhs: E, rhs: E): E;

  /* Scalar-Array operations */
  function _+_(lhs: E, rhs: Array): Array;
  // ... prototypes as above for _-_, _*_, _/_

  /* Array-Array operations */
  function _+_(lhs: Array, rhs: Array): Array;
  // ... prototypes as above for _-_, _*_, _/_

  /* Rotation */
  function rotate(array: Array, axis: Axis, offset: Offset)
    : Array;

  /* Indexing */
  function psi(ix: Index, array: Array): E;
}
```

The declaration of the types and operations form an algebraic *signature*. We augment that signature with semantic properties in the form of *axioms* to obtain a *concept*. Listing 4.4 relates each array-level arithmetic operation in the API to its corresponding scalar-level operation [28; 36]. The axioms for all binary operations follow the same pattern, we hence only show axiom bodies for the + operation for the sake of brevity.

Listing 4.4: Axioms for the arithmetic operations of our array API.

```
concept ArrayAPI_ArithmeticAxioms = {
  require ArrayAPI;

  /* Scalar-Array Axioms */
  axiom scalarBinaryPlusAxiom(lhs: E, rhs: Array,
                              ix: Index) {
    assert psi(ix, lhs + rhs) == lhs + psi(ix, rhs);
```

```
  }
  // axiom scalarBinarySubAxiom(lhs: E, rhs: Array,
  //                            ix: Index)
  // axiom scalarMulAxiom(lhs: E, rhs: Array, ix: Index)
  // axiom scalarDivAxiom(lhs: E, rhs: Array, ix: Index)

  /* Array-Array Axioms */
  axiom arrayBinaryPlusAxiom(lhs: Array, rhs: Array,
                             ix: Index) {
    assert psi(ix, lhs + rhs) ==
           psi(ix, lhs) + psi(ix, rhs);
  }
  // axiom arrayBinarySubAxiom(lhs: Array, rhs: Array,
  //                           ix: Index)
  // axiom arrayMulAxiom(lhs: Array, rhs: Array, ix: Index)
  // axiom arrayDivAxiom(lhs: Array, rhs: Array, ix: Index)
}
```

The specifications in Listing 4.3 are (straightforwardly) implemented as external C++ functions and types, not shown here. Lastly, Listing 4.5 shows our implementation of one full step of the PDE.

Listing 4.5: Implementation of one full step of the PDE solver in Magnolia.

```
/* Solver */
procedure step(upd u0: Array, upd u1: Array, upd u2: Array) {
  var v0 = u0;
  var v1 = u1;
  var v2 = u2;

  v0 = substep(v0, u0, u0, u1, u2);
  v1 = substep(v1, u1, u0, u1, u2);
  v2 = substep(v2, u2, u0, u1, u2);
  u0 = substep(u0, v0, u0, u1, u2);
  u1 = substep(u1, v1, u0, u1, u2);
  u2 = substep(u2, v2, u0, u1, u2);
}

function substep(u: Array, v: Array, u0: Array,
                 u1: Array, u2: Array) : Array =
  u + dt()/(two(): Float) * (nu() *
  ((one(): Float)/dx()/dx() *
    (rotate(v, zero(), -one(): Offset) +
     rotate(v, zero(), one(): Offset) +
     rotate(v, one(): Axis, -one(): Offset) +
     rotate(v, one(): Axis, one(): Offset) +
     rotate(v, two(): Axis, -one(): Offset) +
     rotate(v, two(): Axis, one(): Offset)) -
    three() * (two(): Float)/dx()/dx() * u0) -
  (one(): Float)/(two(): Float)/dx() *
    ((rotate(v, zero(), one(): Offset) -
```

```
      rotate(v, zero(), -one(): Offset)) * u0 +
     (rotate(v, one(): Axis, one(): Offset) -
      rotate(v, one(): Axis, -one(): Offset)) * u1 +
     (rotate(v, two(): Axis, one(): Offset) -
      rotate(v, two(): Axis, -one(): Offset)) * u2));

/* Float ops */
require function -_(f: Float): Float;
// magnoliac does not offer support for literals as of yet,
// and we must thus define constant functions in the host
// language for each number of a given type we want to use.
require function one(): Float;
require function two(): Float;
require function three(): Float;

/* Axis utils */
require function zero(): Axis;
require function one(): Axis;
require function two(): Axis;

/* Offset utils */
require function one(): Offset;
require function -_(o: Offset): Offset;

/* Problem-specific parameters */
require function nu(): Float;
require function dt(): Float;
require function dx(): Float;
```

4.3.2 Deriving Optimization Rules

Armed with a thorough understanding of the problem, we can now derive semantics-preserving optimization rules—hardware-specific or otherwise.

Before we can apply rewriting rules defined using MoA to our program, we need to change its level of abstraction, i.e. go from an implementation that describes the resulting array using whole-array operations to one that describes its value at every index. This transformation corresponds to the step from the high-level array program to a corresponding MoA expression in Figure 4.2.

We define a Magnolia program called `BasePDEProgram` that contains the functions in Listing 4.5, giving a concrete implementation to the basic underlying operations and data structures in a host language. Listing 4.6 shows how we achieve the transformation from the high-level array program to a corresponding MoA expression in Magnolia. We break down the components of the listing in the following.

Listing 4.6: Lowering step from a high-level array program to a MoA expression.

```
program PDEProgramMoA = {
  use (rewrite
```

```
        (implement ToIxwiseGenerator in BasePDEProgram)
      with ToIxwise 1);
  use ExtBasicSchedule;
};
```


The `ToIxwiseGenerator` concept is shown in Listing 4.7. The `toIxwiseGenerator` axiom consists of a single assertion, which describes the behavior of the `substepIx` function when all of its arguments are universally quantified distinct variables. The right-hand side of the equation is thus a valid implementation for `substepIx`. Because this function is not implemented in the original program, we can use the *implement* transformation with `ToIxwiseGenerator` to generate an implementation of `substepIx` in the implementation given in Listing 4.5. So as to enable further optimizations, *implement* unfolds function calls in the right-hand side of the equation. The resulting index-level code is shown in Listing 10 (in Appendix 3.3).

Listing 4.7: A generator for an index-level implementation of substep.

```
concept ToIxwiseGenerator = {
  type Array;
  type Float;
  type Index;

  function substepIx(u: Array, v: Array, u0: Array,
                     u1: Array, u2: Array, ix: Index)
    : Float;
  function substep(u: Array, v: Array, u0: Array,
                   u1: Array, u2: Array): Array;

  function psi(ix: Index, array: Array): Float;

  axiom toIxwiseGenerator(u: Array, v: Array, u0: Array,
                          u1: Array, u2: Array, ix: Index) {
    assert substepIx(u, v, u0, u1, u2, ix) ==
           psi(ix, substep(u, v, u0, u1, u2));
  }
}
```

To make use of `substepIx` within the program, we need to replace calls to `substep` with calls to a scheduling function `schedule` that uses `substepIx` to describe the value of the result array at every index. This is achieved through the outermost program transformation in Listing 4.10, that uses the `ToIxwise` concept of Listing 4.8. Throughout the rest of the paper, we use the term *schedule* as in Halide [149].

Listing 4.8: A concept with a rewrite rule from substep to a new scheduling function.

```
concept ToIxwise = {
  type Array;

  function substep(u: Array, v: Array,
                   u0: Array, u1: Array, u2: Array): Array;
  function schedule(u: Array, v: Array,
                    u0: Array, u1: Array, u2: Array): Array;
```

```
  axiom toIxwiseRule(u: Array, v: Array,
                     u0: Array, u1: Array, u2: Array) {
    assert substep(u, v, u0, u1, u2) ==
           schedule(u, v, u0, u1, u2);
  }
}
```

Magnolia does not expose native looping constructs. For that reason, the implementation of `schedule` is done in the host language, and imported through `ExtBasicSchedule`. From that point onwards, we can use MoA's transformation rules on our program.

4.3.2.1 Reusability of Modules

Of the modules presented in Listings 4.6, 4.7, and 4.8, `PDEProgramMoA` is the only one that is completely problem-specific. The `ToIxwise` and `ToIxwiseGenerator` concepts's types and operations are given names that relate to our domain of application. Due to the renaming feature however, specific names within a module are largely irrelevant: two signatures that expose the same set of types and operations (and overloads) up to renaming (of types and operations) can be made to match.

For example, the `ToIxwise` concept states that there exists two functions with the same prototype (they take in five arguments of the same type that is also the return type), such that calling one of them is equivalent to calling the other. Listing 4.9 shows a more generic presentation of the `ToIxwise` concept.

Listing 4.9: A domain-generic formulation of the *ToIxwise* concept.

```
concept FunctionEquality5 = {
    type T;
    function f(t1: T, t2: T, t3: T, t4: T, t5: T): T;
    function g(t1: T, t2: T, t3: T, t4: T, t5: T): T;

    axiom functionEqualityRule(
        t1: T, t2: T, t3: T, t4: T, t5: T) {
      assert f(t1, t2, t3, t4, t5) == g(t1, t2, t3, t4, t5);
    }
}

// ToIxwise can be defined from FunctionEquality5, and
// vice-versa.
concept ToIxwise =
  FunctionEquality5[ f => substep
                   , g => schedule
                   , T => Array
                   , functionEquality5Rule => toIxwiseRule
                   ];
```

The `FunctionEquality5` concept can be further generalized by taking in arguments of five different

types, and returning an element of a sixth different type—all of which would be mapped to `Array` for defining `ToIxwise`. The concept could also be stated more concisely for any number of arguments using *variadics*. Magnolia does not support variadics today, but the feature is a desired future language extension [38].

The listings in the rest of the paper follow the same pattern. We present Magnolia program definitions that apply specific rewrite rules in a specific way and are completely problem-specific, and Magnolia concept definitions that specify reusable rewriting systems and are completely problem-independent.

4.3.2.2 Hardware-Agnostic Transformation Rules

The next step outlined in Figure 4.2 is to reduce the MoA expression we just constructed to its DNF. Rewriting rules at the DNF level do not require hardware knowledge, and therefore constitute hardware-agnostic transformation rules. Listing 4.10 shows how we achieve this transformation in Magnolia. The `DNFRules` concept is spelled out in Listing 9, which can be found in Appendix 3.3. The choice of applying the rewrite rules defined in `DNFRules` twenty times is carefully made by the developer, and leads to the full reduction of the MoA expression produced previously to its DNF.

Listing 4.10: Transforming the MoA expression to its DNF.

```
program PDEProgramDNF =
  rewrite PDEProgramMoA with DNFRules 20;
```

The DNF reduction rules discussed here mirror the specification presented in Listing 4.4 for the arithmetic operations of the API, and are completely reusable for any program based on this API. These axioms describe for each operation the content of the resulting array at each index, turning the array from a large opaque block to a function from its index space to its content. Applying the DNF rules pushes computations down from the array-level to the index-level, i.e. the resulting computations are devoid of whole-array operations and contain only indexing and scalar arithmetic operations. This can be thought of as *loop fusion*, or also as some kind of *function composition*.

The external schedule implemented in Listing 4.6 describes how the indexwise computation is executed on the underlying hardware. The `psi` function is also given an external implementation that thus describes how the arrays are actually laid out in memory. This hardware "specialization" gives us an initial executable OF—as outlined in Figure 4.2. For the sake of completeness, our (non-specific) C++ implementation of `schedule` is shown in Listing 11 (available in Appendix 3.3).

The reduction of our MoA expression to its DNF (and resulting "default" OF) already leads to significant performance improvements. Table 4.1 shows runtime results for our PDE solver implementation in Magnolia, before and after full DNF reduction using the DNF rewriting rules. The baseline implementation shown here is a direct lowering of the program written in Magnolia to C++, and we do not perform any transformation beyond what is offered by g++'s optimization level O3 (for the CPUs) and what is offered by nvcc by default (for the GPU). Every array is allocated on the heap, and every intermediate array in the computation is materialized—resulting in a baseline that is inefficient. DNF reduction speeds up the code by a factor between roughly 5.78× and 14.11× depending on the targeted device, and significantly reduces memory usage. At the DNF level, the expression is written in terms of scalar and indexing operations, eliminating the costly need to compute temporary arrays, and increasing computational density.

This experiment shows that such a rewriting system gives the ability to write programs using whole-array operations without losing out on the benefits of writing index-level code. The ability to write algorithms in different ways without inducing a loss of performance is key to the productive development of performant code.

	CPU 1	CPU 2	GPU
Before DNF reduction	2622.09	3494.35	8.75
After DNF reduction	312.10	604.25	0.62

Table 4.1: Execution time (in seconds) of the 3-dimensional PDE solver Magnolia implementation compiled to C++, with and without reduction to DNF. The code is compiled with g++ 10.2.0 with optimization level O3 for the CPU runs; it is compiled with nvcc 11.6 for the GPU runs. The space dimensions are $512 \times 512 \times 512$ and the solver is run for 50 timesteps. The code is run 5 times on each device, and the time measurements are averaged. CPU 1: Intel Xeon Silver 4112, CPU 2: ThunderX2 CN9980, GPU: NVIDIA A100.

4.3.2.3 Hardware-Specific Transformation Rules

Which hardware-specific transformation rules are relevant to implement is by nature dependent on the underlying hardware architecture we are interested in. For example, Chetioui et al.'s previous work on formalizing PDE computations in MoA gave rise to rules for introducing padding into array expressions [37]. Their work also discusses rewrites rules that use the *dimension lifting* operation, which is a *reshape* operation with the explicit purpose of matching the shape of arrays with characteristics of the underlying hardware—more commonly called *array partitioning*. E.g. lifting by d_1 across the first axis allows one to *scatter* the resulting subarrays across d_1 processes; or, lifting by 4 across the last axis of an array of 32-bit floats allows one to vectorize computations on an architecture with 128-bit vector registers. The hardware architecture combined with the data dependencies of the algorithm determine the shape and layout of the arrays.

We discuss an example of a hardware-dependent rewriting system for padding below.

Example: Padding computations Our example assumes a toroidal space—i.e. the first element is a neighbor of the last element for each dimension. Figure 4.3 shows the dependency patterns for one third of a half-step of the PDE across the last axis of the array. The element at index i at time $t+1$ depends on the elements at index i, $(i-1) \bmod N$, and $(i+1) \bmod N$ at time t. The modulo operation serves to index the right dependencies for the first (respectively last) element of the array, where decrementing (respectively incrementing) the index would create an out-of-bounds index. Modulo operations are still expensive, even on modern hardware [117]. Additionally, if N is large, the computations at the boundary need to access elements that are far apart in memory—therefore benefitting less from data locality than the computations in the middle of the array.

How boundaries are handled in our computation is not relevant for our previous rewrite rules. However, Chetioui et al. showed that padding is a way to eliminate these modulo computations and to increase data locality for such dependency patterns, at the cost of duplicating data in memory [37].

Figure 4.4 shows the dependency patterns for one third of a half-step of the PDE across the last axis of the array when the array is padded. In that case, the computation at the boundaries of the array

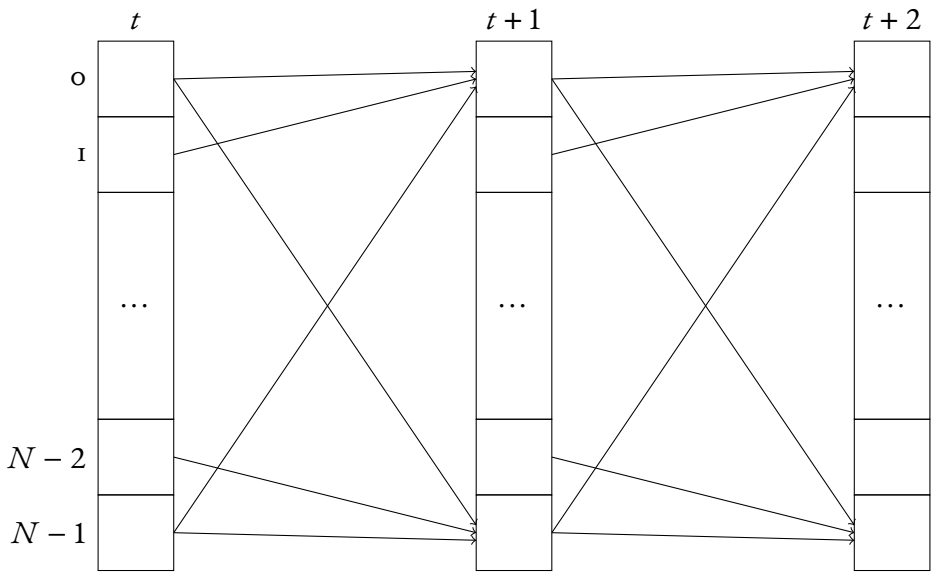

Figure 4.3: The dependency pattern for one third of a half-step of the PDE across the last axis of the array. Each column represents an array of length N indexed from 0 to $N-1$ for a given timestep. The element at index i of the array at time $t+1$ depends on the elements at indices i, $(i-1) \bmod N$ and $(i+1) \bmod N$ of the array at time t.

can be rewritten to depend on three adjacent elements in the array. The modulo computation can also be eliminated. We pay for these improvements by using more space, and by refilling the padding before every timestep.

Listing 4.11 shows one way of introducing padding in the `PDEProgramDNF` program introduced in Listing 4.10. We truncate some of the modules in the following listings so as not to clutter the presentation, and add corresponding listings containing the full modules to Appendix 3.3 for the sake of completeness.

Listing 4.11: Introducing padding into *PDEProgramDNF*. The full specification of *OFPad* can be found in Listing 12.

```
program PDEProgramPadded = {
  use (rewrite PDEProgramDNF with OFPad 1);
  // imports a new schedule, a new function for index
  // rotation, and a procedure for refilling padding
  use ExtExtendPadding;
}

concept OFPad = {
  ...
  procedure refillPadding(upd a: Array);
  function schedulePadded(u: Array, v: Array,
                          u0: Array, u1: Array, u2: Array)
    : Array;
  function schedule(u: Array, v: Array,
                    u0: Array, u1: Array, u2: Array): Array;
  axiom padRule(u: Array, v: Array,
                u0: Array, u1: Array, u2: Array) {
```

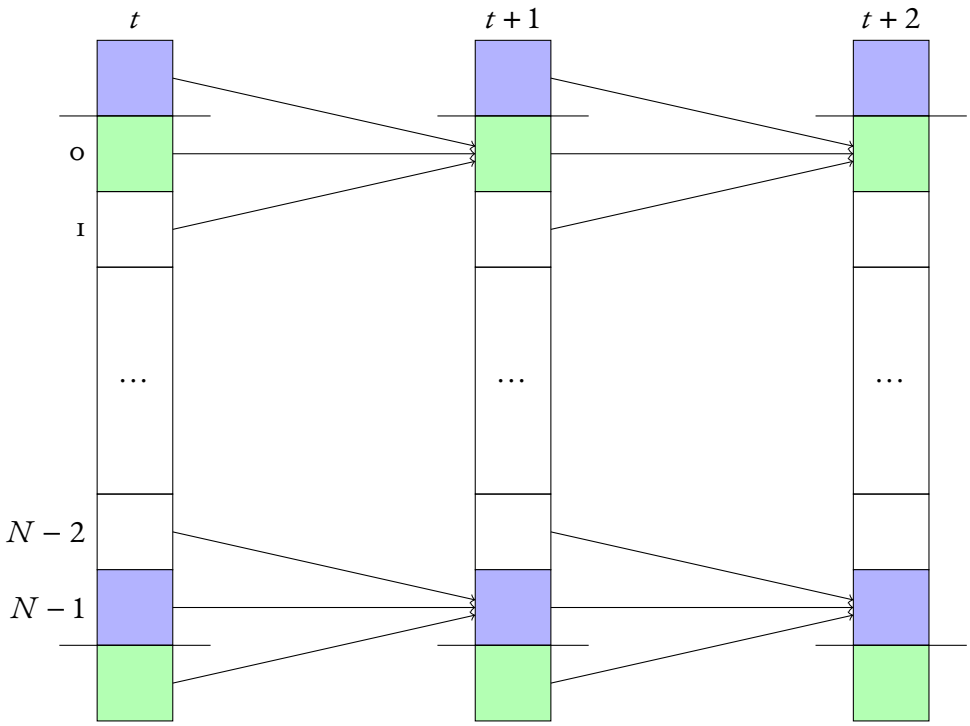

Figure 4.4: The dependency pattern for one third of a half-step of the PDE across the last axis of the array when the array is padded once on each side on the last axis. Each column represents an array of length N indexed from 0 to $N-1$ for a given timestep. The elements colored in the same color have the same value. The element at index i of the array at time $t+1$ depends on the elements at indices i, $i-1$ and $i+1$ of the array at time t.

```
    assert schedule(u, v, u0, u1, u2) ==
      { var result = schedulePadded(u, v, u0, u1, u2);
        call refillPadding(result);
        value result;
      };
  }

  function rotateIx(ix: Index, axis: Axis, offset: Offset)
    : Index;
  function rotateIxPadded(ix: Index, axis: Axis,
                          offset: Offset): Index;
  axiom rotateIxPadRule(ix: Index, axis: Axis,
                        offset: Offset) {
    assert rotateIx(ix, axis, offset) ==
           rotateIxPadded(ix, axis, offset);
  }
}
```

The transformations given by `OFPad` act on the OF of the program. The transformation rules replace calls to `rotateIx` with calls to `rotateIxPadded` within the implementation of `substepIx`, and calls to `schedule` with calls to `schedulePadded` succeeded by an operation refilling the padding within the implementation of `step`. The resulting `step` procedure is shown in Listing 14 in Appendix 3.3. Likewise, the schedule is now replaced by one that is mindful of padding—whose C++ implementation is shown in Listing 13 in Appendix 3.3. The result is a program with a different—a priori more optimized—OF. The rewrites correspond to the last transformation step in the methodology presented in Figure 4.2.

Our implementation in Listing 4.11 assumes that the input arrays are padded arbitrarily across each axis in the host language, in a way that is compatible with the new `rotateIxPadded` function. Details such as the amount of padding across each axis are therefore not visible in Magnolia. This is however purely a design choice, insofar as we have chosen to make the `Index` type completely opaque. This has the benefit of making the program naturally shape polymorphic to a degree—though the program is not as interesting for input arrays with initial number of dimensions different than three.

We can control padding across each axis more explicitly by specializing our code further. This can also be achieved using transformation rules, as is shown in Listing 4.12.

Listing 4.12: Adding padding to and specializing *PDEProgramDNF* to 3 dimensions.

```
program PDEProgram3DPadded = {
  use (rewrite
        (rewrite
          (rewrite
            (rewrite
              (implement OFSpecializeSubstepGenerator in
                         PDEProgramDNF)
              with OFSpecializePsi 10)
            with OFReduceMakeIxRotate 20)
          with OFPad[schedulePadded =>
                     schedule3DPadded] 1)
```

```
        with OFEliminateModuloPadding 10);

  use ExtScalarIndex;     // pulling in ScalarIndex utils
  use ExtAxisLength;      // pulling in AxisLength utils
  use ExtSpecializeBase; // pulling in psi
  use Ext3DSchedule;      // pulling in schedule3DPadded
}
```


The content of `OFSpecializeSubstepGenerator` is shown in Listing 4.13. The concept contains an axiom following the generator pattern to specialize the shape polymorphic `substepIx` to three dimensions. As previously, the call to `substepIx` on the right-hand side of the equation is unfolded to enable additional optimizations.

Listing 4.13: A generator for a 3D implementation of substepIx. The full specification of the concept can be found in Listing 15.

```
concept OFSpecializeSubstepGenerator = {
  ...
  function mkIx(i: ScalarIndex, j: ScalarIndex,
                k: ScalarIndex): Index;
  function substepIx(u: Array, v: Array,
      u0: Array, u1: Array, u2: Array, ix: Index): Float;
  function substepIx3D(u: Array, v: Array,
      u0: Array, u1: Array, u2: Array,
      i: ScalarIndex, j: ScalarIndex, k: ScalarIndex)
    : Float;

  axiom specializeSubstepRule(u: Array, v: Array,
      u0: Array, u1: Array, u2: Array,
      i: ScalarIndex, j: ScalarIndex, k: ScalarIndex) {
    assert substepIx3D(u, v, u0, u1, u2, i, j, k) ==
           substepIx(u, v, u0, u1, u2, mkIx(i, j, k));
  }
};
```

Recall the original implementation of `substepIx` given in Listing 10. Every indexing operation of some array `a` in the resulting implementation of `substepIx3D` is now either of the form `psi(mkIx(i, j, k), a)`, or of the form `psi(rotateIx(mkIx(i, j, k), x, o), a)` for some axis `x` and some offset `o`.

The `OFSpecializePsi` (shown in Listing 4.14) then introduces a specialized `psi` function for 3D arrays. It does that by introducing three projection functions `ix0`, `ix1`, and `ix2` on `Indexes`. General indexing operations of the form `psi(mkIx(i, j, k), a)` are first specialized to expressions of the form `psi(ix0(mkIx(i, j, k)), ix1(mkIx(i, j, k)), ix2(mkIx(i, j, k)), a)` by an application of `specializePsiRule`—which can then be reduced to `psi(i, j, k, a)` via three applications of `reduceMakeIxRule`.

Listing 4.14: Specializing calls to the indexing function ψ. The full specification of the concept can be found in Listing 16.

```
concept OFSpecializePsi = {
```

```
  ...
  type ScalarIndex;

  function ix0(ix: Index): ScalarIndex;
  ...

  function mkIx(i: ScalarIndex, j: ScalarIndex,
                k: ScalarIndex): Index;
  function psi(i: ScalarIndex, j: ScalarIndex,
               k: ScalarIndex, array: Array): E;

  axiom specializePsiRule(ix: Index, array: Array) {
    assert psi(ix, array) ==
           psi(ix0(ix), ix1(ix), ix2(ix), array);
  }

  axiom reduceMakeIxRule(i: ScalarIndex, j: ScalarIndex,
                         k: ScalarIndex) {
    var ix = mkIx(i, j, k);
    assert ix0(ix) == i;
    assert ix1(ix) == j;
    assert ix2(ix) == k;
  }
}[ E => Float ];
```

We also want to call our specialized version of `psi` instead of the general one in expressions now of the form `psi(ix0(rx), ix1(rx), ix2(rx), a)` where `rx = rotateIx(mkIx(i, j, k), x, o)`. For that purpose, we apply the rewriting rules defined in `OFReduceMakeIxRotate`—shown in Listing 4.15. These rewriting rules essentially unfold `rotateIx`. All the indexing operations in `substepIx3D` now use the specialized form of `psi`, and the scalar indices are either constants or of the form `(i + o) % s`, with `i` a scalar index, `o` an offset, and `s` the length of the relevant axis of the array.

Listing 4.15: A rewriting system to specialize the index rotation operation. The full specification of the concept can be found in Listing 17.

```
concept OFReduceMakeIxRotate = {
  ...

  function rotateIx(ix: Index, axis: Axis, offset: Offset)
    : Index;

  type AxisLength;
  function shape0(): AxisLength;
  ...

  function _+_(six: ScalarIndex, o: Offset): ScalarIndex;
  function _%_(six: ScalarIndex, sc: AxisLength)
    : ScalarIndex;
```

```
  axiom reduceMakeIxRotateRule(i: ScalarIndex,
    j: ScalarIndex, k: ScalarIndex, o: Offset) {
    var ix = mkIx(i, j, k);

    assert ix0(rotateIx(ix, zero(), o)) ==
           (i + o) % shape0();
    assert ix0(rotateIx(ix, one(), o)) == i;
    assert ix0(rotateIx(ix, two(), o)) == i;
    ...
    assert ix1(rotateIx(ix, one(), o)) ==
           (j + o) % shape1();
    ...
    assert ix2(rotateIx(ix, two(), o)) ==
           (k + o) % shape2();
  }
}
```

At this point, we can reintroduce padding using the rules previously defined in Listing 4.11, and renaming `schedulePadded` to `schedule3DPadded`. As we are in the case when an implementation for `schedulePadded` is not in scope before the rules defined in `OFPad` are applied, we can replace the rewrite by a simple renaming—as shown in Listing 4.12.

We decide to implement this function externally such that the array is always circularly padded at least once on each side of each axis — a decision made based on the width of the stencil. With that knowledge, we can completely eliminate the modulo operations in `substepIx3D`. The `OFEliminateModuloPadding` concept (shown in Listing 4.16) defines the relevant rewriting rules.

Listing 4.16: Elimination of the modulo operations in the program. The full specification of the concept can be found in Listing 18.

```
// We suppose here that the amount of padding is sufficient
// across each axis for every indexing operation.
concept OFEliminateModuloPadding = {
  ...
  function psi(i: ScalarIndex, j: ScalarIndex,
               k: ScalarIndex, a: Array): Float;

  axiom eliminateModuloPaddingRule(i: ScalarIndex,
    j: ScalarIndex, k: ScalarIndex, a: Array, o: Offset) {
    assert psi((i + o) % shape0(), j, k, a) ==
           psi(i + o, j, k, a);
    assert psi(i, (j + o) % shape1(), k, a) ==
           psi(i, j + o, k, a);
    assert psi(i, j, (k + o) % shape2(), a) ==
           psi(i, j, k + o, a);
  }
}
```

Table 4.2 gives an overview of the performance variations for four different implementations, all

produced by the application of rewriting rules on our original solver implementation presented in Listing 4.5. We briefly describe the resulting implementations below.

DNF reduction This implementation is the same as `PDEProgramDNF` as described in Listing 4.10.

DNF reduction + Padding This implementation adds padding to `PDEProgramDNF`, i.e. it is the same as `PDEProgramPadded` as described in Listing 4.11.

DNF reduction + Index specialization This implementation is a version of `PDEProgramDNF` in which each element of type `Index` is transformed into three elements of type `ScalarIndex`, e.g. a call of the form `psi(ix, a)` is transformed into a call of the form `psi(i, j, k, a)`, and rotation along a specific axis is implemented as a modular addition over that axis.

DNF reduction + Index specialization + Padding This implementation is a version of `PDEProgramPadded` in which elements of type `Index` are also decomposed into three elements of type `ScalarIndex`. It is assumed that each axis is padded sufficiently such that rotation can be implemented as a non-modular addition.

On both CPUs, the performance variation follows the same pattern. The fastest runs are achieved by the padded versions of the baseline implementation with DNF reduction applied, and its counterpart with also specialized indexing. In the unpadded case, the version of the code that incorporates specialized indexing runs faster—1.22× faster on CPU 1, and 1.08× faster on CPU 2. As outlined above, we expect padded implementations to perform better due to increased data locality at the boundaries of the computations. On the GPU considered, the variations are different: the unpadded programs (with or without specialized indexing) perform best. We conclude that this is due to additional calls to the expensive *cudaMemcpy* in the implementation of `replenishPadding`—which add a cost to the computation that is not offset by the expected benefits of padding.

	CPU 1	CPU 2	GPU
DNF reduction	312.10	604.25	0.62
DNF reduction + Padding	190.46	311.52	0.91
DNF reduction + Index specialization	256.64	561.95	0.63
DNF reduction + Index specialization + Padding	190.95	313.22	0.91

Table 4.2: Execution time (in seconds) of the 3-dimensional PDE solver Magnolia implementation compiled to C++ with specialized indexing and with or without padding. The code is compiled with g++ 10.2.0 with optimization level O3 for the CPU runs; it is compiled with nvcc 11.6 for the GPU runs. The space dimensions are 512 × 512 × 512 and the solver is run for 50 timesteps. In the padded case, each axis is padded circularly exactly once on both ends. The code is run 5 times on each device. CPU 1: Intel Xeon Silver 4112, CPU 2: ThunderX2 CN9980, GPU: NVIDIA A100.

Crucially, the performance improvements and variations we observe here did not require any reimplementation of the core algorithm. Building our core algorithm generically allows us to introduce specialized underlying types and operations, once more information is known about our input data or the underlying hardware architecture. The Magnolia term rewriting engine then allows us to introduce new operations and to replace calls to existing concrete implementations with calls to other functions with possibly different argument lists.

This is another twist of generic programming: *rewrite* and *implement* allow to replace operations (or combinations of operations) in a generic module with others that have potentially different argument lists — so long as we can describe the behavior of the former at all points in terms of calls to the new operation(s).

4.4 Discussion and Related Work

We presented a methodology for solving the P³ problem on existing and emerging architectures and applied it to the domain of array computations. Instead of developing one program to target n hardware architectures, we implement a single program, along with hardware-specific rewriting rules. By relating the high-level problem to a mathematical basis, we ensure that the set of optimization rules we implement is correct, and reusable for problems that can be embedded within the same formalism. For example, the exact set of optimization rules we defined may be reused with other explicit finite difference solvers—and likely for stencil computations in general—as these are problems for which Burrows et al.'s API is suitable [28].

Magnolia gives developers the tools to write high-level, domain-specific compilers with custom optimization rules, and a custom target language. The ability to choose flexibly to which opaque building blocks a Magnolia program reduces allows the application of optimization rules at various abstraction levels, until the boundary between Magnolia and the external primitives implemented in the host language is reached. Our approach is centered around the idea of expressing generic algorithms independently from any particular schedule, i.e. independently from any hardware abstraction.

As we mentioned in Section 4.1.1, the term *schedule* as used throughout the paper originates in the work of Ragan-Kelley et al. on Halide [149]. SPIRAL [146] and Sequoia [54] predate Halide, but make a similar distinction between an algorithm and its mapping to the underlying hardware architecture. Halide exposes a set of scheduling primitives from which developers can build their own schedules. TVM [34] follows this idea and extends Halide's set of scheduling primitives. The set of schedules that can be expressed in such systems is necessarily limited by the set of available scheduling primitives. Extending this set requires modifications to the language and its compiler, and is thus costly. Recent work by Liu et al. shows that carefully choosing high-level rewriting rules on schedules allows optimizing tensor programs beyond what is currently possible in these languages [120]. Exo allows for expressing schedules for different hardware targets through composable rewrites and user-defined hardware abstractions [99]. Ikarashi et al. note that adding support for new hardware using a library approach (as in Exo) appears to require one order of magnitude less development time than in systems like Halide or TVM. In our system, schedules are fully specified by the developer—similarly to the work of Ikarashi et al.. Compared to the approach taken by Halide or TVM, the developer has full control over how their computations are executed, but incur a higher implementation cost when no scheduling algorithm exists for their particular flavor of target hardware architecture. Adding "default" scheduling primitives to Magnolia as a convenience could improve the developer experience, and is therefore a consideration for future work.

MLIR [115] makes heavy use of rewrite rules through the MLIR *PatternRewrite* infrastructure [173]. Their design is influenced by LIFT [166; 167], another programming language exploiting rewrite rules for high-performance array computations. In LIFT, the application of rewrite rules is automated by a stochastic search method. Hagedorn et al. extend LIFT specifically for optimizing stencil programs [81]. Such rewrite approaches are so far limited in that they do not always deliver high

enough performance for real-world use [82]. This is in contrast to autoscheduling in Halide, which outperforms human experts on average [9]. Automatic scheduling techniques are key to improving solutions to the P³ problem, and are thus an important topic to further explore also for rewrite rules-based optimizers.

4

Approaches to optimization based on rewrite rules, such as the one presented here, can benefit from rewriting strategies, e.g. for localizing rewrites to only a particular chunk of the input program or for traversing the AST in a specific order. Kirchner gives a recent survey of strategic rewriting [105]. Example of tools implementing such strategies include Maude [44; 122; 123] and Stratego [175]. Hagedorn et al. introduce a functional approach to high-performance code generation based on rewriting strategies [82]: computations are expressed in the RISE programming language, and rewrite rules and strategies in the ELEVATE strategy language. Fu et al. [58] later added a type system to ELEVATE to ensure statically that rewrites are composed correctly. As shown throughout the paper, our rewriting system today only provides the ability to apply sets of rewrite rules a certain number of times, in sequence. Given a rule $e_1 = e_2$, the sequence $e_1;\ e_1$ can be rewritten to $e_2;\ e_1$, but not directly to $e_1;\ e_2$. Such a transformation can be expressed today by applying the rule $e_1 = e_2$ twice, and then applying the opposite rule $e_2 = e_1$ once, but this is both embarassingly verbose and inefficient. Adding rewriting strategies to Magnolia will unlock those rewrites that are not easily accessible today, and thus further improve the system's code reuse capabilities. The implementation of Magnolia strategies is of particular interest, and fits into our larger project of exploring module transformations through the lens of *Syntactic Theory Functors* [87].

For future work, we also envision the implementation of an extension to the Magnolia rewriting system that supports conditional rewrite rules. Conditional equations can already be expressed in Magnolia, but the rewriting system is not yet able to exploit them.

Whether axioms constitute valid rewriting rules is verifiable by extending Magnolia with formal verification tools—insofar as the relevant properties that a program must satisfy can be derived from the stated axioms about its external building blocks. The properties asserted about externally implemented code can however only be assumed to hold, and constitute the trusted computing base of the whole program. Work on connecting verification tools with Magnolia's specification facilities is already underway, with encouraging results [83].

Acknowledgements

The research presented in this paper has benefited from the Experiment Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

Part III

Reflections

Chapter 3

Concluding remarks

The work presented in this dissertation shows that the marriage of "classic" generic programming with property-based specifications yields programming languages that are well positioned for driving down the cost of constructing high quality software. Key to this positioning are the additional opportunities for reuse that property-based specifications unlock.

The empirical evidence gathered through the several experiments presented in this thesis shows that generic programming and abstraction are not antithetical to performance. Even further, Paper 4 shows a methodology aimed for tackling the P^3 (performance, portability, and productivity) problem using Magnolia. Despite the success of our investigations, many more research directions remain unexplored. The below suggests directions that the line of research pursued in this dissertation should take going forward.

The learnings extracted from Magnolia have already found their way to influence the design of generics in a mainstream programming language: Fortran [91]. An overarching question throughout the course of my research project was how we could go beyond building an experimental research language, and put Magnolia itself in the hands of *real* users—those that Mary Shaw dubs *vernacular programmers* in her inspiring HOPL IV keynote address *"Myths and mythconceptions: what does it mean to be a programming language, anyhow?"* [158]. If we want to concretize on our goal of making the construction of high quality software less costly with Magnolia, then the language needs to find a user base—it can't raise the quality of software across the board if it isn't used. This question guided many of the design choices throughout the development of magnoliac. It encouraged me to strive to produce a compiler not merely functional, but instead a properly engineered and documented software artifact. I have not succeeded yet—as of today, I am not aware of any non-research project using or building atop Magnolia or magnoliac.

From a purely pragmatic perspective, the ease of connecting a tool with a user base seems to mainly depend on the answers to two questions. First, how much immediate value does the tool provide to its target audience? Second, how much additional pain does using the tool cause to its target audience? The astute reader may notice that a tool that puts generic programming as its core value proposition is ill-positioned on these two questions by design: making use of the tool immediately brings new challenges (even if only that of learning to use it), but does not yield tangible value until it has enabled a critical amount of code reuse.

For Magnolia to achieve the ambitious goal of finding a user base of vernacular programmers, it needs

a clear value proposition on which users could capitalize very early. Our work in Paper 4 goes in the way of producing such a value proposition: it exploits Magnolia's generic programming facilities to construct programs that can be retargeted to and optimized for different hardware platforms by designing and applying arbitrary transformation rules defined directly in Magnolia. That value needs to be contrasted with the difficulties that are simultaneously introduced by the use of Magnolia.

3

3.1 Some Difficulties of Magnolia Development

Magnolia programming can be challenging. We identify here some sharp bits of Magnolia development, and offer some ideas to smooth them out and improve the development experience.

3.1.1 What is in a Module?

A challenge we often encounter when programming with Magnolia is keeping track of what is in scope for a given module. Paper 1 mentions this issue.

We usually intend to build sophisticated modules in Magnolia by combining many smaller modules—in order to reuse components as much as possible. Sophisticated **implementation**s end up containing many declarations—including some that do not yet have an associated definition. This does not cause any visible issue until we try to produce a **program** from a sophisticated **implementation**.

Because a **program** has the additional constraint that every single declaration it contains must have an associated definition, we frequently run into compiler errors indicating that declarations in the **program**'s scope that we don't particularly care about or need are missing a definition. In that case, the right fix is to provide such a definition, but where to do so is often unclear. E.g., we may have forgotten to rename the declaration at some point to unify it with another definition—but where exactly should we have done that? This depends on where the declaration came from, and which renamings were applied to it, etc.

The problem we highlight here calls for a good Integrated Development Environment (IDE) that would help users cope with the singular challenges of the language. This is not a groundbreaking discovery, nor a controversial argument: Bagge's work on providing a Magnolia IDE in Eclipse was motivated by similar concerns [14; 16], and magnoliac specifically captures some metadata during the checking phase in order to make it easier to eventually integrate it with an IDE.

Another issue worth highlighting here is that Magnolia programs end up exposing an interface that is often wider than the one that was intended. To handle this, we may want to augment Magnolia with support for some kind of encapsulation. OCaml's *module sealing* [41, Chapter 5, Section 4] spring to mind as a possible source of inspiration.

3.1.2 No Ad-Hoc Programming in Magnolia

Producing generic code in Magnolia is no more complicated than producing ad-hoc, specialized code—but producing ad-hoc, specialized code in Magnolia is typically harder than in other programming

Listing 3.1: A functional implementation of `fizzbuzz`. The `doFizzbuzz` function takes an integer as input, and returns `fizz()` if it is divisible by 3, `buzz()` if it is divisible by 5, and `fizzbuzz()` if it is divisible by both 3 and 5. Unlike in the usual implementation, `doFizzbuzz` is not called on a range of integers, and does not directly print to the output. This is because it makes the implementation simpler in Magnolia. The primitive types and operations necessary to the program are implemented in C++.

```
package fizzbuzz;

implementation IntegerType = external C++ base.integer_type {
    type Int;
    function zero(): Int;
    function three(): Int;
    function five(): Int;
    function modulo(a: Int, modulus: Int): Int;
}

implementation FizzbuzzOps =
  external C++ base.fizzbuzz_ops {
    type Fizzbuzz;
    function fizz(): Fizzbuzz;
    function buzz(): Fizzbuzz;
    function fizzbuzz(): Fizzbuzz;
    function nope(): Fizzbuzz;
  }

program Fizzbuzz = {
    use FizzbuzzOps;
    use IntegerType;

    function doFizzbuzz(i: Int): Fizzbuzz =
        if modulo(i, three()) == zero() &&
           modulo(i, five()) == zero()
        then fizzbuzz()
        else if modulo(i, three()) == zero() then fizz()
        else if modulo(i, five()) == zero() then buzz()
        else nope();
}
```

languages. This is an obstacle to using Magnolia in workflows involving iterative experimentation. Listing 3.1 shows a possible functional implementation of `fizzbuzz` in Magnolia.

The implementation consists of over 30 lines of Magnolia code, along with the implementation of the `base.integer_type` and `base.fizzbuzz_ops` data structures in C++. These two definitions appear in a different C++ source file, and mirror the declarations in `IntegerType` and `FizzbuzzOps`. In an ideal world, `IntegerType` would already be provided by Magnolia's standard library, but `FizzbuzzOps` (being a very ad-hoc data structure) wouldn't.

With such a standard library made available, the main remaining challenge would be the connection of Magnolia code to the underlying host language and the code duplication it introduces.

One suggestion to make this effortless for the user is to allow inlining blocks of Magnolia code inside a host-language source file, and to automatically generate **external** modules from the host-language source code. This would allow users to start writing experimental Magnolia code easily from within an existing codebase, while benefitting freely from everything that was already defined in the host language.

3.2 Where Magnolia could Deliver Value

3.2.1 Fuzzing and Property-Based Testing

Fuzzing consists in generating random inputs and feeding them to a test oracle. Fuzzing is a popular technique for hardening complex and critical pieces of software, e.g. compilers [181].

Property-based testing is a form of targeted fuzzing, where the test oracle validates that properties of the software under test hold for the generated inputs. Property-based testing is a widely used testing technique, thanks to popular libraries like e.g. QuickCheck [40] in Haskell, or Hypothesis [121] in Python.

Property-based testing is another use case for Magnolia's property-based specifications. Given a satisfaction relation asserting that an implementation models a concept, we can automatically derive property-based tests for that implementation. Thanks to Magnolia's parametrized modules and its genericity by host language, a relevant satisfaction relation can be used to generate property-based tests for any refinement of the modeling implementation across host languages. This offers one additional opportunity for reuse: that of generated property-based test suites.

Introducing Magnolia into a codebase as a testing tool rather than as a development tool also seems like an easier sell. Testing code is typically well decoupled from implementation code. There is therefore no need to intertwine existing code with Magnolia code. One only needs to express the relevant specifications and to define relevant **external** implementations in Magnolia to start harnessing the outlined benefits.

Bagge and Haveraaen already used axioms for property-based testing in their paper *"Axiom-Based Transformations: Optimization and Testing"* [17], and later also in *"Testing with Axioms in C++ 2011"* [20] in collaboration with David. This line of research seems impactful, and deserves to be explored further.

3.2.2 Dynamically Selecting Concept Implementations

When writing programs, developers often have several valid alternatives to choose from for data structures and algorithms. For instance, there exists many different implementations for the concept of a `map`, and different algorithms to perform a `sort`.

Choosing the right implementation is not always easy, and often depends on characteristics of the data that is being processed. For instance, while a `hashmap` is commonly used to implement fast key-based lookups, there are many cases in which the data structure's performance ends up being subpar. For instance, when there is very little data to search through, a contiguous `list` of key-value pairs may have better runtime characteristics. Similarly for `sort`, it is common to use an insertion sort implementation when sorting very small arrays, even though quicksort would be a better alternative in the more general case.

Magnolia allows us to document the syntactic and semantic requirements of our data types and algorithms, and to make them requirements of implementation parameters. Given a precise enough specification, and leveraging the renaming mechanism, it becomes possible to use constructs modeling the same requirements interchangeably. Given a host of such specifications and implementations, a compiler equipped with profile-guided optimization facilities would be able to—over time—produce a program where all the data structures and algorithms are the most appropriate for the program's purposes.

Promising work in this direction already materialized in Qin, O'Connor, and Steuwer's paper *Primrose: Selecting Container Data Types by Their Properties* [147].

3.3 Onwards

Property-based specifications can be used to make software better. This dissertation leveraged them successfully to exploit additional kinds of reuses inaccessible to generic programming facilities that do not offer property-based specifications. Reuse is a tried-and-true method to raise the quality of software, and property-based specifications thus appear to be a desired feature to make available in programming languages on the path towards practical high quality software. However, constructing a programming language that can use them effectively and practically is far from being a solved problem. It is commendable how mainstream programming languages have managed to construct useful and effective generic programming facilities while completely side-stepping the many difficulties associated with using property-based specifications.

I hope that the research presented here will contribute to making genericity by property a more compelling choice of paradigm for language implementers in the future, and a stepping stone towards practical high quality software.

Bibliography

[1] Haskell 2010 language report, 2010. URL https://www.haskell.org/onlinereport/haskell2010/. [Last accessed 22-October-2021].

[2] Haskell Applicative => Monad Proposal, 2014. URL https://wiki.haskell.org/Functor-Applicative-Monad_Proposal. [Last accessed 22-October-2021].

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[4] Philip Samuel Abrams. *An APL machine*. PhD thesis, Stanford University, Stanford, CA, USA, 1970.

[5] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 145–158, 2020. doi:10.1109/ISCA45697.2020.00023.

[6] Ole Jørgen Abusdal. Transformations for array programming. Master's thesis, The University of Bergen, 2020. URL https://bora.uib.no/bora-xmlui/handle/1956/22284. [Last accessed 29-August-2024].

[7] Evrim Acar, Animashree Anandkumar, Lenore R. Mullin, Sebnem Rusitschka, and Volker Tresp. Tensor Computing for Internet of Things (Dagstuhl Perspectives Workshop 16152). *Dagstuhl Reports*, 6(4):57–79, 2016. ISSN 2192-5283. doi:10.4230/DagRep.6.4.57.

[8] Evrim Acar, Animashree Anandkumar, Lenore R. Mullin, Sebnem Rusitschka, and Volker Tresp. Tensor Computing for Internet of Things (Dagstuhl Perspectives Workshop 16152). *Dagstuhl Manifestos*, 7(1):52–68, 2018. ISSN 2193-2433. doi:10.4230/DagMan.7.1.52.

[9] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley.

Learning to optimize Halide with tree search and random programs. *ACM Trans. Graph.*, 38 (4), jul 2019. ISSN 0730-0301. doi:10.1145/3306346.3322967.

[10] Andreas Arvidsson, Moa Johansson, and Robin Touche. Proving type class laws for haskell. In David Van Horn and John Hughes, editors, *Trends in Functional Programming*, pages 61–74, Cham, 2019. Springer International Publishing. ISBN 978-3-030-14805-8. doi:10.1007/978-3-030-14805-8_4.

[11] Roland Carl Backhouse, Patrik Jansson, Johan Jeuring, and Lambert G. L. T. Meertens. Generic Programming: An Introduction. In *Revised Lectures of the Third International Spring School on Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, 1998. ISBN 3-540-66241-3. doi:10.1007/10704973_2.

[12] Brett W. Bader, Tamara G. Kolda, et al. Matlab tensor toolbox version 2.6. Available online, February 2015. URL http://www.sandia.gov/~tgkolda/TensorToolbox/. [Last accessed 6-June-2019].

[13] Anya Helene Bagge. The Magnolia programming language. In *PLDI Student Research Competition 2008*. ACM, 2008. URL http://www.ii.uib.no/~anya/papers/bagge-pldisrc08-magnolia.html. [Last accessed 29-August-2024].

[14] Anya Helene Bagge. *Constructs & Concepts: Language Design for Flexibility and Reliability*. PhD thesis, Research School in Information and Communication Technology, Department of Informatics, University of Bergen, Norway, PB 7803, 5020 Bergen, Norway, 2009. URL http://www.ii.uib.no/~anya/phd/. [Last accessed 30-September-2022].

[15] Anya Helene Bagge. Facts, resources and the IDE/compiler mind-meld. In *Proceedings of the 4th International Workshop on Academic Software Development Tools and Techniques (WASDeTT'13)*, July 2013. URL http://wasdett.org/2013/submissions/wasdett2013_submission_10.pdf. [Last accessed 30-September-2022].

[16] Anya Helene Bagge. Managing facts and resources with the pica ide infrastructure library. *Science of Computer Programming*, 134:100–111, 2017. ISSN 0167-6423. doi:10.1016/j.scico.2016.09.004. 6th issue of Experimental Software and Toolkits (EST-6).

[17] Anya Helene Bagge and Magne Haveraaen. Axiom-based transformations: Optimisation and testing. In Jurgen J. Vinju and Adrian Johnstone, editors, *Proceedings of the Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, volume 238, pages 17–33. Elsevier, 2009. doi:10.1016/j.entcs.2009.09.038.

[18] Anya Helene Bagge and Magne Haveraaen. Interfacing concepts: Why declaration style shouldn't matter. In Torbjörn Ekman and Jurgen J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, volume 253, pages 37–50. Elsevier, 2010. doi:10.1016/j.entcs.2010.08.030.

[19] Anya Helene Bagge and Magne Haveraaen. Specification of generic APIs, or: Why algebraic may be better than pre/post. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 71–80, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3217-0. doi:10.1145/2663171.2663183.

[20] Anya Helene Bagge, Valentin David, and Magne Haveraaen. Testing with axioms in C++ 2011. *Journal of Object Technology*, 10:10:1–32, 2011. ISSN 1660-1769. doi:10.5381/jot.2011.10.1.a10.

[21] Toby Bailey and Michael B. Gale. Chesskell: A two-player game at the type level. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Haskell 2021, pages 110–121, New York, NY, USA, 2021. ACM. ISBN 9781450386159. doi:10.1145/3471874.3472987.

[22] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2004. Springer-Verlag. ISBN 3540242872. doi:10.1007/978-3-540-30569-9_3.

[23] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, 1996. URL https://dl.acm.org/doi/pdf/10.1145/236156.236184. [Last accessed 29-August-2024].

[24] Jan A. Bergstra and John V. Tucker. Algebraic specifications of computable and semicomputable data types. *Theoretical Computer Science*, 50(2):137–181, 1987. ISSN 0304-3975. doi:10.1016/0304-3975(87)90123-X.

[25] Klaus Berkling. Arrays and the Lambda Calculus. Technical Report 93, Electrical Engineering and Computer Science Technical Reports, 1990. URL https://surface.syr.edu/cgi/viewcontent.cgi?article=1080&context=eecs_techreports. [Last accessed 29-August-2024].

[26] Michel Bidoit and Peter D. Mosses. *CASL User Manual - Introduction to Using the Common Algebraic Specification Language*, volume 2900 of *Lecture Notes in Computer Science*. Springer, 2004. ISBN 3-540-20766-X. doi:10.1007/b11968.

[27] Johannes Martinus Burgers. A mathematical model illustrating the theory of turbulence. In *Advances in applied mechanics*, volume 1, pages 171–199. Elsevier, 1948. doi:10.1016/S0065-2156(08)70100-5.

[28] Eva Burrows, Helmer André Friis, and Magne Haveraaen. An array API for finite difference methods. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2018, pages 59–66, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5852-1. doi:10.1145/3219753.3219761.

[29] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, dec 1985. ISSN 0360-0300. doi:10.1145/6041.6042.

[30] Jacques Carette, Russell O'Connor, and Yasmine Sharoda. Building on the diamonds between theories: Theory presentation combinators, 2019. URL https://arxiv.org/abs/1812.08079. [Last accessed 29-August-2024].

[31] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 241–253, New York, NY, USA, 2005. ACM. ISBN 1595930647. doi:10.1145/1086365.1086397.

[32] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 1–13, New York, NY, USA, 2005. ACM. ISBN 158113830X. doi:10.1145/1040305.1040306.

[33] Craig Chambers and the Cecil Group. *The Cecil language: specification and rationale, Version 3.2*, 2004. URL https://projectsweb.cs.washington.edu/research/projects/cecil/www/Release/doc-cecil-lang/cecil-spec.pdf. [Last accessed 30-September-2022].

[34] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018. USENIX Association. ISBN 9781931971478.

[35] Benjamin Chetioui. magnoliac: A Magnolia compiler, Dec. 2020. doi:10.5281/zenodo.6572953.

[36] Benjamin Chetioui, Lenore Mullin, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Sandra Macià. Finite difference methods fengshui: Alignment through a Mathematics of Arrays. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, page 2–13, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367172. doi:10.1145/3315454.3329954.

[37] Benjamin Chetioui, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Lenore Mullin. Padding in the Mathematics of Arrays. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2021, page 15–26, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384667. doi:10.1145/3460944.3464311.

[38] Benjamin Chetioui, Jaakko Järvi, and Magne Haveraaen. Revisiting language support for generic programming: When genericity is a core design goal. *The Art, Science, and Engineering of Programming*, 7(2), oct 2022. doi:10.22152/programming-journal.org/2023/7/4.

[39] Benjamin Chetioui, Marius Larnøy, Jaakko Järvi, Magne Haveraaen, and Lenore Mullin. P^3 problem and Magnolia language: Specializing array computations for emerging architectures. *Frontiers in Computer Science*, 4, 2022. doi:10.3389/fcomp.2022.931312.

[40] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1581132026. doi:10.1145/351240.351266.

[41] Michael R. Clarkson, Robert L. Constable, Nate Foster, Michael D. George, Dan Grossman, Justin Hsu, Daniel P. Huttenlocher, Dexter Kozen, Greg Morrisett, Andrew C. Myers, Radu Rugina, and Ramin Zabih. *OCaml Programming: Correct + Efficient + Beautiful*. 2021. URL https://cs3110.github.io/textbook. [Last accessed 29-August-2024].

[42] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of maude. *Electronic Notes in Theoretical Computer Science*, 4:65–89, 1996. ISSN 1571-0661. doi:10.1016/S1571-0661(04)00034-9. RWLW96, First International Workshop on Rewriting Logic and its Applications.

[43] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002. ISSN 0304-3975. doi:10.1016/S0304-3975(01)00359-0. Rewriting Logic and its Applications.

[44] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about Maude—a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3540719407.

[45] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. *Maude Manual (Version 3.2.1)*, February 2022. URL `https://maude.lcc.uma.es/maude321-manual-html/maude-manual.html`. [Last accessed 29-August-2024].

[46] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

[47] James A. Crotinger, Julian Cummings, Scott Haney, William Humphrey, Steve Karmesin, John Reynders, Stephen Smith, and Timothy J. Williams. Generic programming in POOMA and PETE. In Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser, editors, *Generic Programming*, pages 218–231, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-39953-7.

[48] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, may 1979. ISSN 0001-0782. doi:10.1145/359104.359106.

[49] James C. Dehnert and Alexander A. Stepanov. *Fundamentals of Generic Programming*, pages 1–11. In Jazayeri et al. [102], 01 1998. ISBN 3540410902. doi:10.1007/3-540-39953-4_1.

[50] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ report: The language, proof techniques, and methodologies for object-oriented algebraic specification*, volume 6. World Scientific, 1998. ISBN 978-9810235130.

[51] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2020. doi:10.1109/TETC.2017.2785299.

[52] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *ACM Symposium on Operating Systems Principles*, pages 133–150, Farmington, PA, USA, Nov. 2013. doi:10.1145/2517349.2522720.

[53] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110, New York, NY, USA, 2010. ACM. ISBN 9781605586397. doi:10.1145/1774088.1774531.

[54] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, page 83–es, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 0769527000. doi:10.1145/1188455.1188543.

[55] Robert W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993. ISBN 978-94-011-1793-7. doi:10.1007/978-94-011-1793-7_4.

[56] William B. Frakes and Giancarlo Succi. An industrial study of reuse, quality, and productivity. *Journal of Systems and Software*, 57(2):99–106, June 2001. doi:10.1016/S0164-1212(00)00121-7.

[57] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM. ISBN 0897914287. doi:10.1145/113445.113468.

[58] Rongxiao Fu, Xueying Qin, Ornela Dardha, and Michel Steuwer. Row-polymorphic types for strategic rewriting. *CoRR*, abs/2103.13390, 2021. doi:10.48550/arXiv.2103.13390.

[59] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of obj2. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 52–66, New York, NY, USA, 1985. ACM. ISBN 0897911474. doi:10.1145/318593.318610.

[60] Kokichi Futatsugi, Joseph A. Goguen, José Meseguer, and Koji Okada. Parameterized programming in obj2. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 51–60, Washington, DC, USA, 1987. IEEE Computer Society Press. ISBN 0897912160.

[61] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17(2):145–205, 2007. doi:10.1017/S0956796806006198.

[62] Jeremy Gibbons. Datatype-generic programming. In *Proceedings of the 2006 International Conference on Datatype-Generic Programming*, SSDGP'06, pages 1–71, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540767851. doi:10.1007/978-3-540-76786-2_1.

[63] Joseph A. Goguen. Parameterized programming. *IEEE Transactions on Software engineering*, (5):528–543, 1984. URL `https://www-public.imtbs-tsp.eu/~gibson/Teaching/Teaching-ReadingMaterial/Goguen84.pdf`. [Last accessed 29-August-2024].

[64] Joseph A. Goguen. *Higher-Order Functions Considered Unnecessary for Higher-Order Programming*, pages 309–351. Addison-Wesley Longman Publishing Co., Inc., USA, 1990. ISBN 0201172364.

[65] Joseph A. Goguen. *Types as Theories*, pages 357–385. Oxford University Press, Inc., USA, 1991. ISBN 0198537603.

[66] Joseph A. Goguen. *Tossing algebraic flowers down the great divide*, pages 93–129. Springer, New York, 1999. ISBN 9789814021135.

[67] Joseph A. Goguen and Rod M. Burstall. Introducing institutions. In Edmund Clarke and Dexter Kozen, editors, *Logics of Programs*, pages 221–256, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg. ISBN 978-3-540-38775-6. doi:10.1007/3-540-12896-4_366.

[68] Joseph A. Goguen and Rod M. Burstall. Introducing institutions. In Edmund Clarke and Dexter Kozen, editors, *Logics of Programs*, pages 221–256, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg. ISBN 978-3-540-38775-6.

[69] Joseph A. Goguen and William Tracz. *An implementation-oriented semantics for module composition*, pages 231–263. 2000. ISBN 0-521-77164-1.

[70] Joseph A. Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy Winkler. An introduction to OBJ 3. In *1st International Workshop on Conditional Term Rewriting Systems*, pages 258–263, Berlin, Heidelberg, 1988. Springer-Verlag. ISBN 3540192425. doi:10.1007/3-540-19242-5_22.

[71] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. *Introducing OBJ*, pages 3–167. Springer US, Boston, MA, 2000. ISBN 978-1-4757-6541-0. doi:10.1007/978-1-4757-6541-0_1.

[72] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. Mostly automated proof repair for verified libraries. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi:10.1145/3591221.

[73] Andy Gordon and Cédric Fournet. Principles and applications of refinement types. Technical Report MSR-TR-2009-147, October 2009. URL https://www.microsoft.com/en-us/research/publication/principles-and-applications-of-refinement-types/. [Last accessed 30-September-2022].

[74] Philip W. Grant, Magne Haveraaen, and Michael F. Webster. Coordinate free programming of computational fluid dynamics problems. *Scientific Programming*, 8(4):211–230, 2000. doi:10.1155/2000/419840.

[75] Douglas Gregor and Jaakko Järvi. Variadic templates for C++0x. *Journal of Object Technology*, 7 (2):31–51, Feb. 2008. ISSN 1660-1769. doi:10.5381/jot.2008.7.2.a2. OOPS Track at the 22nd ACM Symposium on Applied Computing, SAC 2007.

[76] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-348-4. doi:10.1145/1167473.1167499.

[77] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. Featherweight go. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), nov 2020. doi:10.1145/3428217.

[78] Ian Grout and Lenore R. Mullin. Hardware considerations for tensor implementation and analysis using the field programmable gate array. *Electronics*, 7(11), 2018. ISSN 2079-9292. doi:10.3390/electronics7110320.

[79] John L. Gustafson and Lenore R. Mullin. Tensors come of age: Why the AI revolution will help HPC. *CoRR*, abs/1709.09108, 2017. doi:10.48550/arXiv.1709.09108.

[80] John L. Gustafson and Isaac T. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2):71–86, Apr. 2017. doi:10.14529/jsfi170206.

[81] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 100–112, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5617-6. doi:10.1145/3168824.

[82] Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. Achieving high-performance the functional way: A functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.*, 4 (ICFP), aug 2020. doi:10.1145/3408974.

[83] Hans-Christian Hamre. Automated verifications for Magnolia satisfactions. Master's thesis, The University of Bergen, 2022. [Available on request].

[84] Magne Haveraaen. Domain engineering the Magnolia way. In Alexander K. Petrenko and Andrei Voronkov, editors, *Perspectives of System Informatics - 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers*, volume 10742 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2017. ISBN 978-3-319-74313-4. doi:10.1007/978-3-319-74313-4_15.

[85] Magne Haveraaen. Axiom based testing for fun and pedagogy. In Antonio Cerone and Markus Roggenbach, editors, *Formal Methods – Fun for Everybody*, pages 27–57, Cham, 2021. Springer International Publishing. ISBN 978-3-030-71374-4. doi:10.1007/978-3-030-71374-4_2.

[86] Magne Haveraaen and Karl Trygve Kalleberg. JAxT and JDI: The simplicity of JUnit applied to axioms and data invariants. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 731–732, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-220-7. doi:10.1145/1449814.1449834.

[87] Magne Haveraaen and Markus Roggenbach. Specifying with syntactic theory functors. *Journal of Logical and Algebraic Methods in Programming*, 113:100543, 2020. ISSN 2352-2208. doi:10.1016/j.jlamp.2020.100543.

[88] Magne Haveraaen and Eric G. Wagner. Guarded algebras: Disguising partiality so you won't know whether its there. In Didier Bert, Christine Choppy, and Peter D. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, pages 182–200, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44616-3. doi:10.1007/978-3-540-44616-3_11.

[89] Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. Formal software engineering for computational modelling. *Nord. J. Comput.*, 6(3):241–270, 1999. URL https://www.ii.uib.no/saga/papers/HaveraaenFriisJohansen1999njc-sophus_coordinate_free_numerics_formal_software_engineering_computational_modelling-haveraaenfj1999.pdf. [Last accessed 29-August-2024].

[90] Magne Haveraaen, Helmer André Friis, and Hans Munthe-Kaas. Computable scalar fields: a basis for PDE software. *Journal of Logic and Algebraic Programming*, 65(1):36–49, September-October 2005. doi:10.1016/j.jlap.2004.12.001.

[91] Magne Haveraaen, Jaakko Järvi, and Damian Rouson. Reflecting on generics for Fortran. Technical report, 2019. URL https://j3-fortran.org/doc/year/19/19-188.pdf. [Last accessed 30-September-2022].

[92] Ralf Hinze and Johan Jeuring. Weaving a web. *Journal of Functional Programming*, 11(6): 681–689, 2001. doi:10.1017/S0956796801004129.

[93] Ralf Hinze and Johan Jeuring. *Generic Haskell: Practice and Theory*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2003. ISBN 978-3-540-45191-4. doi:10.1007/978-3-540-45191-4_1.

[94] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. URL http://sunnyday.mit.edu/16.355/Hoare-CACM-69.pdf. [Last accessed 29-August-2024].

[95] Paul Holser. junit-quickcheck: Property-based testing, junit-style, 2014. URL https://pholser.github.io/junit-quickcheck/site/1.0/. [Last accessed 30-May-2022].

[96] Harry B. Hunt III, L. Mullin, and Daniel J. Rosenkrantz. Experimental design and development of a polyalgorithm for the FFT. Technical Report 98–5, University at Albany, Department of Computer Science, 1998.

[97] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23(3): 396–450, may 2001. ISSN 0164-0925. doi:10.1145/503502.503505.

[98] Harry B. Hunt III, Lenore R. Mullin, Daniel J. Rosenkrantz, and James E. Raynolds. A transformation–based approach for the design of parallel/distributed scientific software: the FFT. *CoRR*, abs/0811.2535, 2008. doi:10.48550/arXiv.0811.2535.

[99] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 703–718, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi:10.1145/3519939.3523446.

[100] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, Inc. New York, 1962. URL https://www.softwarepreservation.org/projects/apl/Books/APROGRAMMING%20LANGUAGE. [Last accessed 29-August-2024].

[101] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 1–19, New York, NY, USA, 2005. ACM. ISBN 1595930310. doi:10.1145/1094811.1094813.

[102] Mehdi Jazayeri, Ruediger Loos, and David Musser. *Generic Programming: International Seminar on Generic Programming Dagstuhl Castle, Germany, April 27–May 1, 1998 Selected Papers*. 01 2000. ISBN 978-3-540-41090-4. doi:10.1007/3-540-39953-4.

[103] Wolfram Kahl and Jan Scheffczyk. Named instances for Haskell type classes. In Ralf Hinze, editor, *Proceedings of the 2001 Haskell Workshop*, volume 59 of *ENTCS*, 2001. URL https://www.cas.mcmaster.ca/~kahl/Publications/Conf/Kahl-Scheffczyk-2001.html. [Last accessed 29-August-2024].

[104] Deepak Kapur, David R. Musser, and Alexander A. Stepanov. Operators and algebraic structures. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 59–64, New York, NY, USA, 1981. ACM. ISBN 0897910605. doi:10.1145/800223.806763.

[105] Hélene Kirchner. *Rewriting Strategies and Strategic Rewrite Programs*, pages 380–403. Springer International Publishing, Cham, 2015. ISBN 978-3-319-23165-5. doi:10.1007/978-3-319-23165-5_18.

[106] Oleg Kiselyov. Functions with the variable number of (variously typed) arguments, June 2004. URL https://okmij.org/ftp/Haskell/polyvariadic.html#polyvar-fn. [Last accessed 24-May-2022].

[107] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107, New York, NY, USA, 2004. ACM. ISBN 1581138504. doi:10.1145/1017472.1017488.

[108] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. HList: Heterogeneous lists, February 2022. URL https://hackage.haskell.org/package/HList-0.5.2.0/. [Last accessed 24-May-2022].

[109] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1:77:1–77:29, Oct. 2017. ISSN 2475-1421. doi:10.1145/3133901.

[110] Fredrik Berg Kjolstad and Marc Snir. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450301275. doi:10.1145/1953611.1953615.

[111] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1), feb 2014. ISSN 0734-2071. doi:10.1145/2560537.

[112] Jan W. Klop, Marc Bezem, and Roel C. De Vrijer, editors. *Term Rewriting Systems*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0521391156.

[113] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, September 2009. doi:10.1137/07070111X.

[114] Herb Krasner. Cost of poor software quality in the US: a 2022 report, 2022. URL https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/. [Last accessed 29-August-2024].

[115] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi:10.1109/CGO51591.2021.9370308.

[116] Gary T. Leavens and Yoonsik Cheon. Design by contract with jml, 2006. URL https://www.cs.ucf.edu/~leavens/JML/jmldbc.pdf. [Last accessed 20-September-2022].

[117] Daniel Lemire, Owen Kaser, and Nathan Kurz. Faster remainder by direct computation: Applications to compilers and software libraries. *Softw., Pract. Exper.*, 49(6):953–970, 2019. doi:10.1002/spe.2689.

[118] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 42–54, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595930272. doi:10.1145/1111037.1111042.

[119] John M. Li and Andrew W. Appel. Deriving efficient program transformations from rewrite rules. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021. doi:10.1145/3473579.

[120] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.*, 6 (POPL), jan 2022. doi:10.1145/3498717.

[121] David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 11 2019. ISSN 2475-9066. doi:10.21105/joss.01891.

[122] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Towards a strategy language for Maude. *Electron. Notes Theor. Comput. Sci.*, 117(C):417–441, jan 2005. ISSN 1571-0661.

[123] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. A rewriting semantics for Maude strategies. *Electron. Notes Theor. Comput. Sci.*, 238(3):227–247, jun 2009. ISSN 1571-0661. doi:10.1016/j.entcs.2009.05.022.

[124] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, NY, 1991. ISBN 0-13-247925-7.

[125] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992. doi:10.1109/2.161279.

[126] Andrey Mokhov. United monoids. *The Art, Science, and Engineering of Programming*, 6(3), feb 2022. doi:10.22152/programming-journal.org/2022/6/12.

[127] Lenore R. Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, 12 1988. URL https://www.researchgate.net/publication/308893116_A_Mathematics_of_Arrays. [Last accessed 29-August-2024].

[128] Lenore R. Mullin. Psi, the indexing function: A basis for FFP with arrays. In *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers, 1991. doi:10.1007/978-1-4615-4002-1_12.

[129] Lenore R. Mullin. A uniform way of reasoning about array-based computation in radar: Algebraically connecting the hardware/software boundary. *Digital Signal Processing*, 15(5): 466–520, 2005. doi:10.1016/j.dsp.2005.02.003.

[130] Lenore R. Mullin and Michael A. Jenkins. Effective data parallel computation using the Psi calculus. *Concurrency—Practice and Experience*, 8(7):499–515, 1996. doi:10.1002/(SICI)1096-9128(199609)8:7<499::AID-CPE230>3.0.CO;2-1.

[131] Lenore R. Mullin and James E. Raynolds. *Scalable, Portable, Verifiable Kronecker Products on Multi-scale Computers*, volume 539 of *Constraint Programming and Decision Making. Studies in Computational Intelligence*. Springer, Cham, 2014. doi:10.1007/978-3-319-04280-0_14.

[132] Lenore R. Mullin and Scott Thibault. Reduction semantics for array expressions: The Psi compiler. Technical Report CSC 94-05, Dept. of CS, University of Missouri-Rolla, 1994. URL https://www.researchgate.net/publication/2705090_A_Reduction_Semantics_for_Array_Expressions_The_PSI_Compiler. [Last accessed 29-August-2024].

[133] Lenore R. Mullin, Ashok Krishnamurthi, and Deepa Iyengar. The design and development of a basis, alpha$_1$, for formal functional programming languages with arrays based on a Mathematics of Arrays. In *Proceedings of the International Conference on Parallel Processing, ICPP '88, The Pennsylvania State University, University Park, PA, USA, August 1988. Volume 2: Software.*, 1988.

[134] Lenore R. Mullin, Daria Dooling, Erik Sandberg, and Scott Thibault. Formal methods for scheduling, routing and communication protocol. In *Proceedings of the Second International Symposium on High Performance Distributed Computing (HPDC-2)*. IEEE Computer Society, July 1993.

[135] Lenore R. Mullin, Edward Rutledge, and Robert Bond. Monolithic compiler experiments using C++ expression templates. In *Proceedings of the High Performance Embedded Computing Workshop (HPEC 2002)*, MIT Lincoln Lab, Lexington, MA, Sept. 2002. URL https://people.csail.mit.edu/bushl2/rpi/portfolio/hpec/cites/4-mullin.pdf. [Last accessed 29-August-2024].

[136] Lenore R. Mullin, Daniel J. Rosenkrantz, Harry B. Hunt III, and X. Luo. Efficient radar processing via array and index algebras. In *Proceedings First Workshop on Optimizations for DSP and Embedded Systems (ODES)*, pages 1–12, San Francisco, CA, Mar. 2003.

[137] David R. Musser and Alexander A. Stepanov. Generic programming. In Patrizia M. Gianni, editor, *Symbolic and Algebraic Computation, International Symposium ISSAC'88, Rome, Italy, July 4-8, 1988, Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1988. doi:10.1007/3-540-51084-2_2.

[138] Derek L. Nazareth and Marcus A. Rothenberger. Assessing the cost-effectiveness of software reuse: A model for planned reuse. *Journal of Systems and Software*, 73(2):245–255, Oct. 2004. doi:10.1016/S0164-1212(03)00248-6.

[139] Matt Noonan. Ghosts of departed proofs (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, pages 119–131, New York, NY, USA, 2018. ACM. ISBN 9781450358354. doi:10.1145/3242744.3242755.

[140] Linda M. Northrop and Paul C. Clements. A framework for software product line practice version 5.0. Whitepaper, Carnegie Mellon University, Software Engineering Institute, 2012. URL http://www.sei.cmu.edu/productlines/framework.html. [Last accessed 29-August-2024].

[141] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM. ISBN 9781450302036. doi:10.1145/1869459.1869489.

[142] Peter Csaba Ölveczky. *Designing Reliable Distributed Systems - A Formal Methods Approach Based on Executable Modeling in Maude*. Undergraduate Topics in Computer Science. Springer, 2017. ISBN 978-1-4471-6686-3. doi:10.1007/978-1-4471-6687-0.

[143] OpenMP Architecture Review Board. OpenMP application program interface version 5.1, November 2020. URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf. [Last accessed 31-January-2022].

[144] Paul Chang and Lenore R. Mullin. An Optimized QR Factorization Algorithm based on a Calculus of Indexing. Technical report, MIT Lincoln Laboratory, 2002. URL https://www.researchgate.net/publication/271530488_An_optimized_QR_Factorization_Algorithm_based_on_a_Calculus_of_Indexing. [Last accessed 29-August-2024].

[145] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In Ralf Hinze, editor, *Proceedings of the 2001 Haskell Workshop*, volume 59 of *ENTCS*, 2001. URL https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/. [Last accessed 29-August-2024].

[146] Markus Puschel, José M.F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005. doi:10.1109/JPROC.2004.840306.

[147] Xueying Qin, Liam O'Connor, and Michel Steuwer. Primrose: Selecting container data types by their properties. *The Art, Science, and Engineering of Programming*, 7(3), Feb. 2023. ISSN 2473-7321. doi:10.22152/programming-journal.org/2023/7/11.

[148] Gabriel Radanne. Typing tricks: Diff lists, August 2016. URL https://drup.github.io/2016/08/02/difflists/. [Last accessed 24-May-2022].

[149] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4), jul 2012. ISSN 0730-0301. doi:10.1145/2185520.2185528.

[150] Talia Ringer. *Proof Repair*. PhD thesis, University of Washington, 2021. URL https://www.proquest.com/dissertations-theses/proof-repair/docview/2568297410/se-2. [Last accessed 29-August-2024].

[151] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. ISBN 9781595938602. doi:10.1145/1375581.1375602.

[152] Donald Sannella and Andrzej Tarlecki. *Extended ML: An Institution-Independent Framework for Formal Program Development*, pages 364–389. Springer Berlin Heidelberg, 1986. ISBN 978-3-540-47213-1. doi:10.1007/3-540-17162-2_133.

[153] Donald Sannella and Andrzej Tarlecki. Mind the gap! Abstract versus concrete models of specifications. In Wojciech Penczek and Andrzej Szalas, editors, *Mathematical Foundations of Computer Science 1996, 21st International Symposium, MFCS'96, Cracow, Poland, September 2-6, 1996, Proceedings*, volume 1113 of *Lecture Notes in Computer Science*, pages 114–134. Springer, 1996. doi:10.1007/3-540-61550-4_143.

[154] Gabriel Scherer. The 6 parameters of ('a, 'b, 'c, 'd, 'e, 'f) format6, April 2014. URL http://gallium.inria.fr/blog/format6/. [Last accessed 24-May-2022].

[155] Gabriel Scherer and Didier Rémy. Gadts meet subtyping. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 554–573, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37036-6. doi:10.1007/978-3-642-37036-6_30.

[156] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 51–62, New York, NY, USA, 2008. ACM. ISBN 9781595939197. doi:10.1145/1411204.1411215.

[157] seL4 Foundation. Frequently Asked Questions on seL4, August 2023. URL https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html. [Last accessed 14-January-2024].

[158] Mary Shaw. Myths and mythconceptions: what does it mean to be a programming language, anyhow? *Proceedings of the ACM on Programming Languages*, 4(HOPL), apr 2022. doi:10.1145/3480947.

[159] Jeremy G. Siek. *A Language for Generic Programming*. PhD thesis, USA, 2005. URL https://www.researchgate.net/publication/44117985_A_Language_for_Generic_Programming. [Last accessed 29-August-2024].

[160] Jeremy G. Siek. *The C++ox "Concepts" Effort*, pages 175–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-32202-0. doi:10.1007/978-3-642-32202-0_4.

[161] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, London, UK, UK, 1998. Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=646894.709706. [Last accessed 29-August-2024].

[162] Jeremy G. Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, page 12, Erfurt, Germany, Oct. 2000. URL https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.427. [Last accessed 01-October-2022].

[163] Jeremy G. Siek and Andrew Lumsdaine. A language for generic programming in the large. *Science of Computer Programming*, 76(5):423–465, 2011. ISSN 0167-6423. doi:10.1016/j.scico.2008.09.009. Special Issue on Generative Programming and Component Engineering (Selected Papers from GPCE 2004/2005).

[164] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-72914-8.

[165] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009. ISBN 032163537X.

[166] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 205–217, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336697. doi:10.1145/2784731.2784754.

[167] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 74–85. IEEE Press, 2017. ISBN 9781509049318. doi:10.1109/CGO.2017.7863730.

[168] Andrew Sutton. Overloading with concepts. *Overload*, 24, December 2016. URL https://accu.org/journals/overload/24/136/sutton_2316/. [Last accessed 30-September-2022].

[169] Andrew Sutton and Jonathan I. Maletic. Emulating C++0x concepts. *Science of Computer Programming*, 78(9):1449–1469, 2013. ISSN 0167-6423. doi:10.1016/j.scico.2012.10.009.

[170] Andrew Sutton and Bjarne Stroustrup. Design of concept libraries for C++. In Anthony Sloane and Uwe Aßmann, editors, *Software Language Engineering*, pages 97–118, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28830-2. doi:10.1007/978-3-642-28830-2_6.

[171] Xiaolong Tang and Jaakko Järvi. Axioms as generic rewrite rules in C++ with concepts. *Science of Computer Programming*, 97:320–330, 2015. ISSN 0167-6423. doi:10.1016/j.scico.2014.05.006. Object-Oriented Programming and Systems (OOPS 2010).

[172] Hai-Chen Tu and Alan J. Perlis. FAC: A functional APL language. *IEEE Software*, 3(1):36–45, Jan. 1986. doi:10.1109/MS.1986.232431.

[173] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction. 2022. doi:10.48550/ARXIV.2202.03293.

[174] Benoît Vaugon. A new format implementation based on GADTs, May 2013. URL `https://github.com/ocaml/ocaml/issues/6017`. [Last accessed 24-May-2022].

[175] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. ISSN 0747-7171. doi:10.1016/j.jsc.2004.12.011. Reduction Strategies in Rewriting and Programming special issue.

[176] Artjoms Šinkarovs, Robert Bernecky, and Sven-Bodo Scholz. Convolutional neural networks in apl. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, page 69–79, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367172. doi:10.1145/3315454.3329960.

[177] Michael Wolfe. Performant, portable, and productive parallel programming with standard languages. *Computing in Science Engineering*, 23(5):39–45, 2021. doi:10.1109/MCSE.2021.3097167.

[178] Dana N. Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 107–116, New York, NY, USA, 2012. ACM. ISBN 9781450311182. doi:10.1145/2103746.2103767.

[179] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 41–52, New York, NY, USA, 2009. ACM. ISBN 9781605583792. doi:10.1145/1480881.1480889.

[180] Edward Z. Yang. Type classes: confluence, coherence and global uniqueness, 2014. URL `http://blog.ezyang.com/2014/07/type-classes-confluence-coherence-global-uniqueness/`. [Last accessed 01-February-2022].

[181] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi:10.1145/1993498.1993532.

[182] Artjoms Šinkarovs. *Data Layout Types: a type-based approach to automatic data layout transformations for improved SIMD vectorisation*. PhD thesis, School of Mathematical and Computer Sciences, Heriot-Watt University, Heriot-Watt University, Edinburgh Campus, Edinburgh, Scotland, EH14 4AS, may 2015. URL https://www.ros.hw.ac.uk/handle/10399/2880. [Last accessed 29-August-2024].

Appendix: Paper I

Listing 2: Example of tension between overloading and subsorting in Maude specifications (adapted from an example by Ölveczky [142, Chapter 2.5]). When calling f on an argument of type *s12*, Maude can not determine which overload of f should be called.

```
fth OVERLOADING is
    sorts s1 s2 s12 u1 u2 .
    op f : s1 -> u1 .
    op f : s2 -> u2 .
endfth

fth SUBSORTING is
    including OVERLOADING .
    subsorts s12 < s1 s2 . --- error!
endfth
```

Listing 3: Implementation of generic BFS utils in Magnolia.

```
implementation GenericBFSUtils = {
  /* snip types and helper operation declarations */
  procedure breadthFirstVisit(obs g: Graph,
      obs s: VertexDescriptor, upd a: A, upd q: Queue,
      upd c: ColorPropertyMap) {
    call discoverVertex(s, g, q, a);
    call push(s, q);
    call put(c, s, gray());
    call bfsOuterLoopRepeat(a, q, c, g);
  }

  predicate bfsOuterLoopCond(a: A, q: Queue,
      c: ColorPropertyMap, g: Graph) {
    value !isEmptyQueue(q);
  }

  procedure bfsOuterLoopStep(upd x: A, upd q: Queue,
      upd c: ColorPropertyMap, obs g: Graph) {
    var u = front(q);
    call pop(q);
    call examineVertex(u, g, q, x);
    var edgeItr: EdgeIterator;
    call outEdges(u, g, edgeItr);
    call bfsInnerLoopRepeat(edgeItr, x, q, c, g, u);
    call put(c, u, black());
    call finishVertex(u, g, q, x);
  }

  procedure bfsInnerLoopStep(obs edgeItr: EdgeIterator,
      upd x: A, upd q: Queue, upd c: ColorPropertyMap,
      obs g: Graph, obs u: VertexDescriptor) {
    var e = edgeIterUnpack(edgeItr);
    var v = tgt(e, g);
    call examineEdge(e, g, q, x);
    var vc = get(c, v);
    if vc == white() then {
      call treeEdge(e, g, q, x);
      call put(c, v, gray());
      call discoverVertex(v, g, q, x);
      call push(v, q);
    } else if vc == gray() then {
      call grayTarget(e, g, q, x);
    } else { // vc == black();
      call blackTarget(e, g, q, x);
    };
  }
}
```

Listing 4: Implementation of a DFS in Magnolia.

```
implementation DFS = {
  // LIFOQueue
  use GraphSearch[ search => depthFirstSearch, front => top,
    isEmptyQueue => isEmptyStack, Queue => Stack ];
  use Stack[ A => VertexDescriptor,
             isEmpty => isEmptyStack ];
}
```

Listing 5: User-provided (hand-coded) implementation of a stack in C++.

```
template <typename _A>
struct stack {
  typedef _A A;
  typedef std::stack<A> Stack;

  Stack empty() { return Stack(); }
  bool isEmpty(const Stack &s) { return s.empty(); }
  void push(const A &a, Stack &s) { s.push(a); }
  void pop(Stack &s) { s.pop(); }
  const A &top(const Stack &s) { return s.top(); }
};
```

Listing 6: User-provided (hand-coded) implementation of a stack in Python.

```
def stack(A):
  class Stack:
    def __init__(self): self.stack = []
    def isEmpty(self): return not self.stack
    def push(self, a: A): self.stack.insert(0, a)
    def pop(self): self.stack = self.stack[1:]
    def top(self): return deepcopy(self.stack[0])
    def mutate(self, other): self.stack = other.stack[:]
  def empty(): return Stack()
  def isEmpty(s: Stack): return s.isEmpty()
  def push(a: A, s: Stack): s.push(a)
  def pop(s: Stack): s.pop()
  def top(s: Stack): return s.top()

  stack_tuple = namedtuple('stack',
    ['A', 'Stack', 'empty', 'isEmpty', 'push', 'pop', 'top'])

  return stack_tuple(A, Stack, empty, isEmpty, push, pop,
                     top)
```

Listing 7: User-provided (hand-coded) implementation of a while loop in C++. The *repeat* procedure is always implemented in the host language, which makes the connection between the three functions *repeat*, *cond* and *body* potentially difficult to identify in a Magnolia program.

```
template <typename _Context, typename _State,
          class _body, class _cond>
struct while_ops {
```

```
  typedef _State State;
  typedef _Context Context;

  _body body;
  _cond cond;

  inline void repeat(State &state, const Context &context) {
    while (while_ops::cond(state, context)) {
      while_ops::body(state, context);
    }
  }
};
```

Listing 8: A problem with constraining concepts.

```
concept Graph = {
  type Graph;
  type Vertex;
  type VertexCollection;

  function adjacentVertices(g: Graph, v: Vertex)
    : VertexCollection;
  function vertices(g: Graph): VertexCollection;
  // predicate member(v: Vertex, vc: VertexCollection);
  // axiom adjacentVerticesAreVertices(
  //     v1: Vertex, v2: Vertex, g: Graph) {
  //   assert member(v2, adjacentVertices(g, v1)) =>
  //          member(v2, vertices(g))
  // }
}
```

Appendix: Paper 4

Listing 9: The DNF rewriting rules in Magnolia.

```
concept GenericBinopRules = {
  type E;
  type Array;
  type Index;

  function binop(lhs: E, rhs: E): E;
  function binop(lhs: E, rhs: Array): Array;
  function binop(lhs: Array, rhs: Array): Array;
  function psi(ix: Index, array: Array): E;

  // Rule 1
  axiom binopArrayRule(ix: Index, lhs: Array, rhs: Array) {
    assert psi(ix, binop(lhs, rhs)) ==
           binop(psi(ix, lhs), psi(ix, rhs));
  }

  // Rule 2
  axiom binopScalarRule(ix: Index, lhs: E, rhs: Array) {
    assert psi(ix, binop(lhs, rhs)) ==
           binop(lhs, psi(ix, rhs));
  }
}

concept DNFRules = {
  use GenericBinopRules[ binop => _+_
                       , binopScalarRule => addScalarRule
                       , binopArrayRule => addArrayRule
                       ];
  use GenericBinopRules[ binop => _-_
                       , binopScalarRule => subScalarRule
                       , binopArrayRule => subArrayRule
                       ];
  use GenericBinopRules[ binop => _*_
                       , binopScalarRule => mulScalarRule
                       , binopArrayRule => mulArrayRule
                       ];
```

```
  use GenericBinopRules[ binop => _/_
                       , binopScalarRule => divScalarRule
                       , binopArrayRule => divArrayRule
                       ];

  type Axis;
  type Offset;

  function rotate(array: Array, axis: Axis, offset: Offset)
    : Array;
  function rotateIx(ix: Index, axis: Axis, offset: Offset)
    : Index;

  // Rule 3
  axiom rotateRule(ix: Index, array: Array, axis: Axis,
                   offset: Offset) {
    assert psi(ix, rotate(array, axis, offset)) ==
           psi(rotateIx(ix, axis, offset), array);
  }
}[ E => Float ];
```

Listing 10: Generated index-level implementation of substep in Magnolia.

```
function substepIx(u: Array, v: Array, u0: Array,
                   u1: Array, u2: Array, ix: Index)
    : Array =
  psi(ix,
      u + dt()/(two(): Float) * (nu() *
      ((one(): Float)/dx()/dx() *
        (rotate(v, zero(), -one(): Offset) +
         rotate(v, zero(), one(): Offset) +
         rotate(v, one(): Axis, -one(): Offset) +
         rotate(v, one(): Axis, one(): Offset) +
         rotate(v, two(): Axis, -one(): Offset) +
         rotate(v, two(): Axis, one(): Offset)) -
        three() * (two(): Float)/dx()/dx() * u0) -
      (one(): Float)/(two(): Float)/dx() *
        ((rotate(v, zero(), one(): Offset) -
          rotate(v, zero(), -one(): Offset)) * u0 +
         (rotate(v, one(): Axis, one(): Offset) -
          rotate(v, one(): Axis, -one(): Offset)) * u1 +
         (rotate(v, two(): Axis, one(): Offset) -
          rotate(v, two(): Axis, -one(): Offset)) * u2)));
```

Listing 11: A naive and non-specific C++ implementation of a scheduling function. *TOTAL_ARRAY_SIZE* corresponds to the number of elements within one array. Every array has the same number of elements.

```
Array schedule(const Array &u, const Array &v,
               const Array &u0, const Array &u1,
               const Array &u2) {
  Array result;
  for (size_t i = 0; i < TOTAL_ARRAY_SIZE; ++i) {
    result[i] = substepIx(u, v, u0, u1, u2, i);
  }
  return result;
}
```

Listing 12: Introducing padding into *PDEProgramDNF*.

```
program PDEProgramPadded = {
  use (rewrite PDEProgramDNF with OFPad 1);
  // imports a new schedule, a new function for index
  // rotation, and a procedure for refilling padding
  use ExtExtendPadding;
}

concept OFPad = {
  type Array;
  type Float;
  procedure refillPadding(upd a: Array);
  function schedulePadded(u: Array, v: Array,
                          u0: Array, u1: Array, u2: Array)
    : Array;
  function schedule(u: Array, v: Array,
                    u0: Array, u1: Array, u2: Array): Array;

  axiom padRule(u: Array, v: Array,
                u0: Array, u1: Array, u2: Array) {
    assert schedule(u, v, u0, u1, u2) ==
      { var result = schedulePadded(u, v, u0, u1, u2);
        call refillPadding(result);
        value result;
      };
  }

  type Index;
  type Axis;
  type Offset;
  function rotateIx(ix: Index, axis: Axis, offset: Offset)
    : Index;
  function rotateIxPadded(ix: Index, axis: Axis,
                          offset: Offset): Index;

  axiom rotateIxPadRule(ix: Index, axis: Axis,
                        offset: Offset) {
    assert rotateIx(ix, axis, offset) ==
           rotateIxPadded(ix, axis, offset);
  }
}
```

Listing 13: A C++ implementation of a scheduling function for padded arrays. All the arrays have the same (three dimensional) shape and each axis is padded by the same amount on each ends. S_0, S_1, and S_2 respectively represent the length of the first, second, and third axis of the arrays. PAD_0, PAD_1, and PAD_2 respectively represent the amount of padding for one end of the first, second, and third axis of the arrays.

```
Array schedulePadded(const Array &u, const Array &v,
                     const Array &u0, const Array &u1,
                     const Array &u2) {
  Array result;
  size_t paddedS1 = S1 + 2 * PAD1;
  size_t paddedS2 = S2 + 2 * PAD2;

  for (size_t i = PAD0; i < S0 + PAD0; ++i) {
    for (size_t j = PAD1; j < S1 + PAD1; ++j) {
      for (size_t k = PAD2; k < S2 + PAD2; ++k) {
        size_t ix =
          i * paddedS1 * paddedS2 + j * paddedS2 + k;
        result[ix] = substepIx(u, v, u0, u1, u2, ix);
      }
    }
  }
  return result;
}
```

Listing 14: The implementation of step produced by an application of *rewrite* with *OFPad*.

```
procedure step(upd u0: Array, upd u1: Array,
               upd u2: Array) = {
  var v0: Array = u0;
  var v1: Array = u1;
  var v2: Array = u2;
  v0 = {
      var result: Array =
        schedulePadded(v0, u0, u0, u1, u2);
      refillPadding(result);
      value result;
  };
  v1 = {
      var result: Array =
        schedulePadded(v1, u1, u0, u1, u2);
      refillPadding(result);
      value result;
  };
  v2 = {
      var result: Array =
        schedulePadded(v2, u2, u0, u1, u2);
      refillPadding(result);
      value result;
  };
  u0 = {
      var result: Array =
        schedulePadded(u0, v0, u0, u1, u2);
      refillPadding(result);
      value result;
  };
  u1 = {
      var result: Array =
        schedulePadded(u1, v1, u0, u1, u2);
      refillPadding(result);
      value result;
  };
  u2 = {
      var result: Array =
        schedulePadded(u2, v2, u0, u1, u2);
      refillPadding(result);
      value result;
  };
};
```

Listing 15: A generator for a 3D implementation of substepIx.

```
concept OFSpecializeSubstepGenerator = {
  type Index;
  type Array;
  type Float;
  type ScalarIndex;

  function mkIx(i: ScalarIndex, j: ScalarIndex,
                k: ScalarIndex): Index;

  function substepIx(u: Array, v: Array,
      u0: Array, u1: Array, u2: Array, ix: Index): Float;

  function substepIx3D(u: Array, v: Array,
      u0: Array, u1: Array, u2: Array, i: ScalarIndex,
      j: ScalarIndex, k: ScalarIndex): Float;

  axiom specializeSubstepRule(u: Array, v: Array,
      u0: Array, u1: Array, u2: Array, i: ScalarIndex,
      j: ScalarIndex, k: ScalarIndex) {
    assert substepIx3D(u, v, u0, u1, u2, i, j, k) ==
           substepIx(u, v, u0, u1, u2, mkIx(i, j, k));
  }
};
```

Listing 16: Specializing calls to the indexing function ψ.

```
concept OFSpecializePsi = {
  type Index;
  type Array;
  type E;
  type ScalarIndex;

  /* 3D index projection functions */
  function ix0(ix: Index): ScalarIndex;
  function ix1(ix: Index): ScalarIndex;
  function ix2(ix: Index): ScalarIndex;

  /* 3D index constructor */
  function mkIx(i: ScalarIndex, j: ScalarIndex,
                k: ScalarIndex): Index;

  function psi(ix: Index, array: Array): E;
  function psi(i: ScalarIndex, j: ScalarIndex,
               k: ScalarIndex, array: Array): E;

  axiom specializePsiRule(ix: Index, array: Array) {
    assert psi(ix, array) ==
           psi(ix0(ix), ix1(ix), ix2(ix), array);
  }

  axiom reduceMakeIxRule(i: ScalarIndex, j: ScalarIndex,
                         k: ScalarIndex) {
    var ix = mkIx(i, j, k);
    assert ix0(ix) == i;
    assert ix1(ix) == j;
    assert ix2(ix) == k;
  }
}[ E => Float ];
```

Listing 17: A rewriting system to specialize the index rotation operation.

```
concept OFReduceMakeIxRotate = {
  use signature(OFSpecializePsi);

  type Axis;
  type Offset;

  function zero(): Axis;
  function one(): Axis;
  function two(): Axis;

  function rotateIx(ix: Index, axis: Axis, offset: Offset)
    : Index;

  type AxisLength;

  function shape0(): AxisLength;
  function shape1(): AxisLength;
  function shape2(): AxisLength;

  function _+_(six: ScalarIndex, o: Offset): ScalarIndex;
  function _%_(six: ScalarIndex, sc: AxisLength)
    : ScalarIndex;

  axiom reduceMakeIxRotateRule(i: ScalarIndex,
      j: ScalarIndex, k: ScalarIndex, array: Array,
      o: Offset) {
    var ix = mkIx(i, j, k);
    var s0 = shape0();
    var s1 = shape1();
    var s2 = shape2();

    assert ix0(rotateIx(ix, zero(), o)) == (i + o) % s0;
    assert ix0(rotateIx(ix, one(), o)) == i;
    assert ix0(rotateIx(ix, two(), o)) == i;

    assert ix1(rotateIx(ix, zero(), o)) == j;
    assert ix1(rotateIx(ix, one(), o)) == (j + o) % s1;
    assert ix1(rotateIx(ix, two(), o)) == j;

    assert ix2(rotateIx(ix, zero(), o)) == k;
    assert ix2(rotateIx(ix, one(), o)) == k;
    assert ix2(rotateIx(ix, two(), o)) == (k + o) % s2;
  }
}
```

Listing 18: Elimination of the modulo operations in the program.

```
// We suppose here that the amount of padding is sufficient
// across each axis for every indexing operation.
concept OFEliminateModuloPadding = {
  use signature(OFReduceMakeIxRotate);

  type Array;
  type Float;

  function psi(i: ScalarIndex, j: ScalarIndex,
               k: ScalarIndex, a: Array): Float;

  axiom eliminateModuloPaddingRule(
      i: ScalarIndex, j: ScalarIndex,
      k: ScalarIndex, a: Array, o: Offset) {
    var s0 = shape0();
    var s1 = shape1();
    var s2 = shape2();

    assert psi((i + o) % s0, j, k, a) ==
           psi(i + o, j, k, a);
    assert psi(i, (j + o) % s1, k, a) ==
           psi(i, j + o, k, a);
    assert psi(i, j, (k + o) % s2, a) ==
           psi(i, j, k + o, a);
  }
}
```