Optimal program variant generation for hybrid manycore systems

Cristian Urlea

Submitted in fulfilment of the requirements for the
Degree of Doctor of Philosophy

School of Engineering
College of Science and Engineering
University of Glasgow

January 2021

Abstract

Field Programmable Gate Arrays (FPGAs) promise to deliver superior energy efficiency in heterogeneous high performance computing (HPC), as compared to multicore CPUs and GPUs. The rate of adoption is however hampered by the relative difficulty of programming FPGAs. High-level synthesis tools such as Xilinx Vivado, Altera OpenCL or Intel's HLS address a large part of the programmability issue by synthesizing a *Hardware Description Languages* (HDL) representation from a high-level specification of the application, given in programming languages such as OpenCL C, typically used to program CPUs and GPUs. Although HLS solutions make programming easier, they fail to also lighten the burden of optimization. Application developers must rely on expert knowledge to manually optimize their applications for each target device, meaning that traditional HLS solutions do not offer a solution to the issue of *performance portability*. This state of fact prompted the development of compiler frameworks such as *TyTra* that operate at an even higher level of abstraction, amenable to the use of *Design Space Exploration* (DSE). With DSE the initial program specification can be seen as the starting location in a *search-space* of *correct-by-construction* program transformations. In TyTra the search-space is generated from the transitive-closure of *term-level transformations* derived from *type-level* transformations. Compiler frameworks such as TyTra theoretically solve the issue of performance portability by providing a way to automatically generate alternative correct program variants. They however suffer from the very practical issue that the generated space is often too large to fully explore. As a consequence, the globally optimal solution may be overlooked.

In this work we provide a novel solution to issue *performance portability* by deriving an efficient yet effective DSE strategy for the TyTra compiler framework. We make use of *categorical data types* to derive *categorical semantics* for the formal languages that describe the terms, types, cost-performance estimates and their transformations. From these we define a category of interpretations for TyTra applications, from which we derive a DSE strategy that finds the globally optimal transformation sequence in polynomial time. This is achieved by reducing the size of the generated search space. We formally state and prove a theorem for this claim and then show that the polynomial run-time for our DSE strategy has practically negligible coefficients leading to sub-second exploration times for realistic applications.

Contents

Abstract **i**

Acknowledgements **viii**

Declaration **ix**

1 Introduction **1**
- 1.1 Design Space Exploration . . . 3
- 1.2 Thesis statement . . . 5
- 1.3 Contributions and Publications . . . 6

2 Background **7**
- 2.1 Performance portability . . . 8
- 2.2 Taxonomy of parallel computation . . . 10
  - 2.2.1 Narrow and wide: data parallelism . . . 11
  - 2.2.2 High and low: abstraction level . . . 15
  - 2.2.3 Near and far: locality . . . 17
  - 2.2.4 The middle way . . . 22
- 2.3 Formal methods . . . 23
  - 2.3.1 Models of computation . . . 24
  - 2.3.2 Imperative languages . . . 26
  - 2.3.3 Functional languages . . . 28
  - 2.3.4 Bridging models . . . 32
- 2.4 The TyTra Compiler Framework . . . 33
  - 2.4.1 TyTra Compiler workflow . . . 33
  - 2.4.2 TyTra Coordination Language . . . 38
  - 2.4.3 TyTra Intermediate Representation . . . 45
  - 2.4.4 TyTra Semantics . . . 46
- 2.5 Introduction to Category Theory . . . 50

3 Related Work **58**
3.1 Practical . . . 58
3.1.1 Imperative Languages . . . 59
3.1.2 Functional Languages . . . 67
3.1.3 Program and Behavioural Synthesis . . . 71
3.1.4 Compiler Optimization Techniques . . . 72
3.2 Theoretical . . . 73
3.2.1 Structured Parallelism . . . 73
3.2.2 Categorical Data Types and Techniques . . . 74
3.3 Graphical summary . . . 78

4 Categorical Semantics **80**
4.1 Categorical Data Types . . . 81
4.1.1 Optimisation from recursion schemes . . . 85
4.1.2 Terms and Transformations . . . 87
4.1.3 Types and Type Constructors . . . 90
4.1.4 Cost-Performance Estimates . . . 95
4.2 DSE in Categorical Terms . . . 99
4.2.1 Specification . . . 99
4.2.2 Analysis . . . 106
4.2.3 Selection . . . 107
4.3 Optimal DSE in TyTra . . . 108
4.3.1 Naïve tactic . . . 111
4.3.2 Expert tactic . . . 116
4.4 TyTra Categorical Semantics . . . 123
4.4.1 Type Inference and checking . . . 124
4.4.2 Correct term transformation . . . 125
4.4.3 Cost-performance aware transformation . . . 126
4.4.4 Borrowing structure . . . 127
4.4.5 Fused DSE . . . 137
4.5 Efficient DSE Theorem . . . 140
4.5.1 Proof . . . 140

5 Experimental Evaluation, Conclusion & Future Work **143**
5.1 Experimental validation . . . 144
5.2 Conclusion . . . 151
5.3 Future work . . . 152

List of Figures

1.1 Conceptual DSE stages: specification, analysis, selection. 3

2.1 Abstract three-dimensional space of programming languages and compilers. . . 10
2.2 CPU (narrow) and GPU (wide) architectures, representing the number of parallel compute units 11
2.3 Pragma directives have a local effect, typically function, variable scope or basic-block level. 13
2.4 SIMD (left) vs scalar (right) addition. 14
2.5 FPGAs: A switched sea of lookup tables. 15
2.6 Harvard Architectures and logical distance. 18
2.7 Relations between parallel architectures and tools. 22
2.8 Graphical depiction of a finite state machine. 26
2.9 Composition of finite state machines. 27
2.10 Lambda abstraction and application. Execution Trace. 29
2.11 **I** and **K** combinators. 30
2.12 The Haskell Monad interface. 31
2.13 High-level view of the TyTra Compiler Workflow. 33
2.14 Detailed view of the TyTra Compiler Workflow. 34
2.15 Performance measured in clock-cycles. 36
2.16 Simplified presentation of the performance model. 37
2.17 TyTra CL term-level expression grammar. 39
2.18 TyTra CL expressions and actions 39
2.19 Alternative graphical representation for TyTra CL expressions and actions. . . . 39
2.20 TyTra CL Size Constructors. 41
2.21 The TyTra CL Atomic Type Constructor 41
2.22 TyTra CL Structural Type Constructors. 42
2.23 Graphical depiction of a TyTra CL expression and types. 42
2.24 An abstract type transformation. 43
2.25 Nested type constructors can be simply seen as a single constructor. 43
2.26 The effect of Vector type Split and Merge operations on terms. 44

2.27 Type-level merge/split rules. Their composition yields the identity transformation. 44
2.28 TyTra IR, its sub-languages and the relationship to TyTra CL 45
2.29 Abstract and full interpretation (sequential). 49
2.30 Abstract interpretation (average time). 49
2.31 The category **C** with three objects. 50
2.32 Composition or morphisms f and g in **C**. 51
2.33 The composition of morphisms must be associative. 51
2.34 The objects of a category are identified by their identity morphisms. 52
2.35 Functors: structure preserving maps between categories. 52
2.36 Triangle identity. 54
2.37 Pentagon identity. 54
2.38 Morphisms from Initial F-Algebra to all F-Algebras in that category. 56

3.1 OpenCL compilation and runtime. 59
3.2 OpenCL optimization workflow. 60
3.3 Fusion rules in Lift. 69
3.4 Cancellation rules in Lift. 69
3.5 The OpenCL Specific *map*, *reduce* and *reorderStride* rules in Lift. 69
3.6 Denotational semantics relating the various *map* implementations to an abstract *map* operation in the Lift compiler. 69
3.7 The build/fold rule corresponds to the Hylomorphism recursion scheme. 77
3.8 Overview of the relationship between practical and theoretical related work. . . 78
3.9 Hylomorphisms as a unifying construct for the object-language of *Structured Arrows*. 79
3.10 Hylomorphisms as a unifying construct for the meta-language of the TyTra Compiler Framework. 79

4.1 $ListF : 1 + a \times List\ a$ describes an Endofunctor. 81
4.2 Category of Integers and List of Integers. 82
4.3 F-Algebra on a list of Integers. 82
4.4 Objects and morphisms in the category of F-Algebras over a functor F. 84
4.5 A term-level structure of nested list functors. 85
4.6 The *TermVar* data constructor is a functor from the category of Strings to the category of TyTra CL Terms 87
4.7 The Type of TyTra CL AST Transformations. 88
4.8 Functorial Type TyTra CL AST Transformations 88
4.9 Identity morphisms on TyTra CL types 89
4.10 TyTra CL Type System derivation rules. 90
4.11 Atomic Types as objects of a category (top) and the category of names (bottom). 91

4.12 Tuple Type projections are morphisms in the category of TyTra CL Types. . . . 92
4.13 A *Vector Type* of size 2 corresponds to a binary tuple $Tup(a,a)$. 92
4.14 Vec is a polynomial functor. 93
4.15 Distributivity of products over sums. 93
4.16 Function types. 94
4.17 Fin_3: the category of finite natural numbers up to the value 3. 96
4.18 Relationship between Fin_3 and a vector type $Vec_3\ a$. 96
4.19 Sub-terms (blue/purple circles) as projections (grey arrows) in the category of terms, and their transformations (blue/purple morphisms). 99
4.20 Design Space of cuboids. 100
4.21 The category of *Point* objects. 101
4.22 The category of *Design'* objects where *Line* is the initial object and its relation to *Design*. 102
4.23 A parametrized design space of cuboids. 103
4.24 The category of *Point* objects. 103
4.25 The unit cube: the initial object in the category of cuboids. 104
4.26 Analysis: maps the design space into an (ordered) space of properties. 106
4.27 Selection via Functors. 107
4.28 Selective object construction. 107
4.29 Map fusion rule. 109
4.30 Family of *split* transformations. 109
4.31 Power set of transformations. 111
4.32 Brute force FSA accepting states. 111
4.33 *Data Variable Terms* generate DS with expressed parallelism $\leq$ size of vector. 112
4.34 *Function Variable Terms* generate DS with expressed parallelism $\leq$ EFI. 113
4.35 Design Space merging 113
4.36 Design space structure traversal producing program transformations. 114
4.37 Cost-performance model output. 115
4.38 Final selection of program transformation sets. 115
4.39 A computational pipeline. 116
4.40 Effect of *Split Transform* on computational pipeline. 116
4.41 Balanced computational pipeline. 117
4.42 *Map Fusion* on balanced computational pipeline. 117
4.43 *Fold Term* fission transformation. 118
4.44 *Tuple Term* distributes bounds. 119
4.45 *Application Term* merges bounds. 119
4.46 *ZipT Term* merges bounds. 120
4.47 *Elt Term*. 121

4.48 *Stencil Term*. 121
4.49 DSE on an isolated sub-term. 122
4.50 DS merge improvement. 122
4.51 Functor from the category of term F-Algebras to the category of type F-Algebras. 124
4.52 Functor from the category of type F-Algebras to the category of term F-Algebras. 125
4.53 Endofunctor on the category of term F-Algebras. 125
4.54 Adjoint functors relating term transformations to cost-performance expressions. 126
4.55 Category of performance measures. 126
4.56 Algebra pairs. 127
4.57 Defining an interpreter on TyTra CL terms as transformation on TermBB. . . . 130
4.58 Type-level application of *viewTerm* to *termApp (termLit "foo") (termLit "bar")*. 130
4.59 Term-level application of *viewTerm* to *termApp (termLit "foo") (termLit "bar")*. 130
4.60 Term-level: Application of *viewTerm* to *termApp (termLit "foo") (termLit "bar")*. 132
4.61 The three DSE stages are dependent. 134
4.62 Homomorphism from *TermBB* to *CostBB*. 135
4.63 Strict CostBB merge operation. 136
4.64 Case-splitting on the *TermBB* encoding of the application. 140
4.65 Graphical summary for the proof of Theorem 1. 141
4.66 Additive sub-space mixing. 142

5.1 Hardware resource limit bounded Design Space (left) vs Filtered Design Space (right) for Synth Kernel on XC6SLX150T. 147
5.2 Hardware simulation showing program variants generated through DSE present the expected cost-performance characteristics. 149

Acknowledgements

I would like to express the most heartfelt thanks to my supervisor, Wim Vanderbauwhede, for introducing me to functional programming, shaping my professional development and supporting me with nothing but kindness and understanding. The same is true of my second supervisor, Jeremy Singer whom not only introduced me to the area of optimizing compilers but also encouraged me to start on this journey. I find it impossible to imagine a more suitable choice of supervisors.

Many thanks go out to the numerous people who have listened to my ideas, contributed their own thoughts and made building this tiny sand castle an enjoyable experience. Amongst these individuals are: Syed Waqar Nabi who has poured countless hours into the TyTra project, many of which for the sole purpose of validating my own work; Sorin Suciu, Andrey Mokhov, Dejice Jacob, Marco Monti who have painstakingly read through many unfinished copies and engaged in discussing some of the most frustrating of half-baked ideas that have come to mind; the many examiners that have sat through my yearly progression vivas; the students that have taught me more than I could ever teach them. A special round of thanks go to my internal and external examiners: Anna Lito Michala and Vijay Nagarajan. Their attention to detail and inquisitive nature have left a very deep and positive mark both this work. Without their effort, this work would have been truly incomplete.

I am grateful to my family and friends for their unbounded love and support. None have unconditionally given more of themselves than my mother, Denisa Urlea, and my dear friend Laura Voinea, who have been there for me through thick and thin. To those looking to cash in their bets: I must admit defeat. I have lost the race to complete this PhD to my mother, who completed hers with a full week's worth of lead time.

Support from the UK EPSRC under grant EP/L00058X/1 is gratefully acknowledged.

Declaration

I hereby declare that, except where specific references are made to the work of others, the contents of this document are original and have not been submitted, in whole or in part, for consideration for any other degree or qualification, in this or any other university. This doctoral thesis is the result of my own work, under the supervision of Prof. Dr. Wim Vanderbauwhede and Dr. Jeremy Singer. Nothing included is the outcome of work done in collaboration, except where otherwise indicated within the text.

Chapters 1, 2 and 3 contain introductory material, background and a discussion of related work. The use of the royal *"we"* as it appears *in relation to the TyTra project* throughout this work indicates that what follows is a personal contribution/statement made, or interpretation/-conclusion drawn, by the author of this work. Work or results derived from work involving other TyTra project members are explicitly indicated.

Chapter 1

Introduction

Heterogeneous High-Performance Computing (HPC) describes the use of multiple parallel hardware architectures in tackling large and complex applications [SEP+09], primarily for reasons having to do with energy efficiency. Specialized hardware architectures are more energy efficient at solving particular problems [SEP+09]. A system incorporating *multiple architectures* can be used to solve larger and more diverse problem sets [BBL+16] [ESFC14]. The most common types of compute hardware used in HPC can be roughly described as follows.

- Multi-core central processing units (CPUs) greatly benefit from the economies of scale in manufacturing. They can execute general purpose applications which exhibit *locally divergent* behaviour [ESFC14]. Geoscience workloads, for example, are typically memory bound [LCP+11]. This means that the abundance of system memory that can be connected to CPUs can benefit them greatly. With more memory, larger problems can be solved by saving intermediate results in main memory.

- Graphical Processing Units (GPUs) implement massively parallel architectures, typically single instruction, multiple data (SIMD) [DKK09]. Although SIMD architectures feature many more compute units than CPUs, meaning that they can process more data items in parallel, divergent behaviour is problematic as groups of compute units operate synchronously sharing a single instruction decoder [DAF11] among them.

- Field Programmable Gate Arrays (FPGAs) implement highly configurable architectures that allow for *massive parallelism*. They also have a higher degree of energy efficiency than CPUs and GPUs [BTL10] [SCP02] [CCA+11]. FPGAs can be seen as configurable circuits that can solve domain-specific problems. The down-side to FPGAs is that they are relatively difficult to program and optimize [BTL10] [CCA+11].

The relative difficulty of programming and optimizing FPGAs is a symptom of the much wider issue of performance portability. Many programming languages (PLs) and compilers have been created to tackle this issue in. particularly in the field of parallel and distributed computation that encompasses HPC. Programming languages are often tightly coupled to particular hardware architectures or features [KS97] [RVDDB10]. as well as the abstractions used to describe distributed computation [BST89]. Applications written in one programming language, that targets a certain system, may be difficult to optimize for execution on new platforms. With every new hardware architecture, programmers must analyse, reimplement or refactor complex applications in order to obtain the much desired speed-up promised by the new hardware. This, in a nutshell, is the issue of *performance portability* [RVDDB10] [FLP+18].

There are many ways in which a programming language can be influenced by architectural decisions. A CPUs memory model, for example, may spell out the semantics of memory operations [BDW16]. Implementation choices for memory caches and prefetching mechanisms may be driven by an assumed type of workload which can vary between workstations and servers [Dow06]. Consumer-grade GPUs may favour single-precision floating point operations, whilst professional-grade hardware, intended for use in computer-aided design may deliver better double-precision performance [PKB14]. FPGAs were initially used as reconfigurable circuits, able to perform the same tasks as application specific circuits (ASIC). Such devices came with small amounts of memory, enough to store the lookup tables which simulate the intended circuit configuration. More recently, FPGAs with large amounts of memory [Leo08] have become available. These can now be used to solve larger and more complicated problems, however, the circuit description of computation is no longer appropriate. In Heterogeneous HPC performance is critical, and it depends on the ability of a programmer to fully account for the architectural differences in the hardware used [GFG+16] [BDPV99]. Unfortunately the individuals that would stand to benefit the most from HPC, scientists and artists are already highly specialized in their respective fields and consequently may not have the breadth of parallel programming expertise required to leverage all of these platforms effectively [SSJ19].

The TyTra compiler framework solves the issue of performance portability by providing an automated optimization route from the programming languages currently used in scientific and high performance computing [VNU19], but also introduces another more specific issue. In TyTra the optimization schedule is recovered through *Design Space Exploration* (DSE), a process that involves *generating* a set of possible optimized applications, *analysing* them to determine their effectiveness, and finally *selecting* the best performing candidate solution. The issue to be solved is that DSE can be very computationally expensive. The search-space generated from even the most trivial of applications can appear to be intractable. Through the present work we solve this issue by providing an efficient DSE implementation for TyTra.

1.1 Design Space Exploration

Design Space Exploration is a process of finding one or more solutions, called *design points*, that satisfy a set of requirements by searching through a design space, and is applicable in many problem domains [KSS+09] [KJS10]. Conceptually. DSE can be seen as a process having three stages, as depicted in Figure 1.1 below. The first stage, *specification*, can be understood as generating the design space of candidate solutions. The second stage, *analysis*, determines properties of interest related to the candidate solutions. The third and final stage, *selection*, leverages these properties to select one or more viable solutions, according to the set of specified requirements.

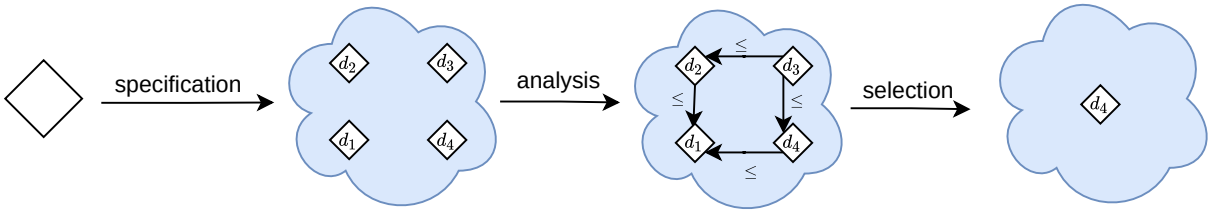


Figure 1.1: Conceptual DSE stages: specification, analysis, selection.

We are primarily interested in the interpretation of DSE within the context of the TyTra compiler framework. TyTra was designed to tackle the issue of optimizing parallel streaming dataflow applications and so it provides the infrastructure required for the *specification* [NV15b] [VN19] *analysis* [NV15b] and selection of optimizing transformations for parallel applications.

Specification involves generating a *design space* of optimizing program transformations through from *type-level* equivalences. The application to be compiled, defines an *initial program variant* made up of term-level expressions that can be described through a powerful dependent type system. Equivalent type expressions denote semantically equivalent implementations that may nonetheless exhibit different performance characteristics. From such equivalences at the type-level, the TyTra compiler defines a set of optimizing transformations for the term-level representation. A sequence of transformations defines a candidate solution, a program variant. The exhaustive set of all transformation sequences defines the total design-space generated in this specification phase.

Analysis involves working out the expected performance an resource use of each program variant. In TyTra, the analysis phase involves applying the transformation sequence defined by a candidate solution to the initial program variant, and then running the cost-performance model implemented by the TyTra back-end compiler.

Selection means that the best performing but least resource-intensive program variant found is returned as the winning solution from the entire search-space generated by the specification stage and characterized by the analysis phase.

Through this work we contribute an efficient DSE strategy implementation for the TyTra compiler. This requires a slightly different interpretation of the three stages we have just outlined.

Specification

Program variants are generated in the TyTra compiler by applying semantics-preserving transformations to an initial program variant. The initial specification is described by a TyTra Coordination Language (TyTra CL) application [section 2.4.2]. Each program variant thus corresponds to *a unique sequence* of program transformations [NV15b] [VN19] [VNU19]. Rather than considering a *design space* to be a set of program variants, we will take on a different interpretation, that a design-space is equivalent to the function that generates it. This alternative view of the specification stage highlights the difference between *intentional* and *extensional* definitions for semantics [section 2.4.4].

Analysis

The TyTra compiler specifies a straight-forward approach to DSE which must execute the cost-performance model on *every* program variant produced by the *specification stage* [NV15b]. This approach implies that every generated *transformation sequence* must be applied to the *initial program variant*, thus generating a new TyTra CL representation for the application. The resulting TyTra CL representation must then be passed through the *code-generator*, giving a TyTra IR encoding. Each program variant encoded in the TyTra IR must then be interpreted using the *cost-performance model* to yield the *total* performance estimate and resource cost. The approach we take in this work is to compute *cost-performance* estimates at the level of each sub-term expression, allowing us to compose the overall cost-performance estimate of each term expression, from the estimates of its sub-terms. In this way, we may compute estimates alongside a representation of the reduced search-space, without having to invoke the code-generator during DSE.

Selection

The *selection* phase in the TyTra compiler must select the best performing yet least resource-intensive program variant from an exhaustive search-space. In our approach the selection phase is fused into the specification phase, restricting the number of generated solutions. We generate only the most efficient and performant solutions. We restrict the generation of solutions that are dominated by already generated program variants, as well as those requiring more hardware resources than are available on the compilation target.

1.2 Thesis statement

As we will see in section 2.1, the issue of *performance portability* in the context of optimizing compilers of HPC, can be solved through DSE. The challenge that remains is to define an efficient exploration strategy for very large *design spaces*. In practical terms, searching through a design space can prove to be intractable, even for small systems and applications. Automated DSE, meaning that which lacks the help of a *human expert* to guide the exploration process, is even more susceptible to this issue. The set of decisions to be made during exploration naturally increases [HHV15] when expert help is not available.

Statement: *Through this work we show that it is possible to implement* automatic design space exploration *strategies for the purpose of identifying* effective optimizing transformations *for* streaming data-flow applications *within a* parallel compiler *targeting* heterogeneous HPC clusters *that incorporate FPGAs, that are both* effective *and* efficient.

We have said that in the case of the TyTra compiler for streaming data-flow applications, a design space consists of alternative program optimizing transformation sequences. The TyTra back-end compiler ultimately generates code for FPGAs. In contrast to CPUs and GPUs, generating code for FPGAs also implies a spatial scheduling of hardware resources, and not just a temporal one. TyTra, being concerned with the optimization of long-running *scientific computations*, implies that we primarily seek to improve an application's *throughput*, rather than its latency. Having established the hardware target, as well as the primary optimization criteria, the attributes *effective* and *efficient* concerning DSE can be given a more specific interpretation.

- An *effective DSE strategy* is one that does not overlook the particular sequence of optimizing transformations that would yield the *maximum possible throughput* for a given application - hardware device pair.
- An *efficient* DSE strategy is one that can be focused on the smallest *sub-region* of the design space which contains the best performing *design point* or *configuration*. From the more practical point of view of solving the issue of heterogeneous HPC cluster programmability, an efficient DSE strategy must yield its solution within minutes, not hours or days as is the case with current high-level FPGA programming frameworks.

Towards the end of this document, in section 4.5, we will use this specific interpretation of *effective* and *efficient* DSE to derive, and thereafter prove, a more formal theorem that embodies the thesis statement just given. In section 1.3 that follows, we use the same interpretation of the *thesis statement* to describe the three specific contributions made through this work.

1.3 Contributions and Publications

The **primary** contribution is the size reduction of the *search-space of optimizing program transformation sequences* generated by the TyTra compiler. The reduction is from an *exponential* design-space to an *additive space* without compromise to the effectiveness of the TyTra compiler. The reduced search-space is shown to be the *globally pareto-optimal frontier* of the *exhaustive search-space*. This contribution takes is given in terms of a formal theorem and proof, both which are presented in section 4.5.

The **secondary** contribution is a validation of the theoretical results in section 4.5 through experimental evaluation of our resulting DSE strategy, on a number of representative applications form the domain of scientific computing. This contribution is given through the experimental methodology presented in section 5.1 which shows that our DSE strategy is not only a function having a polynomial complexity, but also that the constant time-factors in this polynomial are practically small enough, for representative applications, to yield a total design space exploration time that is virtually instantaneous.

The **tertiary** contribution is the specification of *categorical data types* and *categorical semantics* for the TyTra Coordination Language, its type system, the cost-performance estimates returned by the TyTra back-end compiler, as well as the *design space exploration* portion of the TyTra compiler. This contribution borrows methodology form the realm of programming language research, such as defining categorical data types, to enable the theoretical development that supports and enables our primary contribution. The presentation of these concepts and methods is contained within chapter 4.

Previous publications

1. A part of the initial work on the TyTra Coordination Language and DSE has been published in Wim Vanderbauwhede, Syed Waqar Nabi, and Cristian Urlea. Type-driven automated program transformations and cost modelling for optimising streaming programs on fpgas. *International Journal of Parallel Programming*, 47(1):114–136, 2019.

2. Preliminary results regarding the search space reduction have been published in Cristian Urlea, Wim Vanderbauwhede, and Syed Waqar Nabi. Efficient FPGA cost-performance space exploration using type-driven program transformations. In David Andrews, René Cumplido, Claudia Feregrino, and Marco Platzner, editors, *2019 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019*, pages 1–2. IEEE, 2019.

Chapter 2

Background

Scientific Computation is a multidisciplinary field concerned with solving large problems using computational methods. Applications from the scientific computation domain simulate complex patterns of physical or abstract interactions to gain insight into the behaviour and trajectory of such systems [JQR06]. Weather and climate forecasting applications are representative examples of scientific computation. Problems such as these are solved by running complex simulations involving large numbers of data-points representing parameters such as temperature, air pressure and humidity. Climate models in particular must account for both *local* and *global* interactions, such as the relationship between atmospheric conditions and total solar radiation. Furthermore, the input data-points are continuously updated with measurements from real-world data gathered by weather balloons and ground-stations. The sheer amount of data required to encode the state of such a complex system, as well as the computational effort needed to derive probable future states, are the main limiting factors to the granularity and precision of such applications [VT13]. Financial simulation applications are another example of scientific computation. Modern financial institutions straddle the line between finance and technology. They have a moral and legal obligation to ensure that their clients are not over-exposed to financial risk, and so they must run incredibly complex risk calculations that ingest enormous amounts of real-world data, such as currency exchange rates, outstanding debt and market prices with a much higher frequency [Tho10] than weather forecasting applications. The amount of computational power needed to run such applications is reflected in the energy consumption of the HPC clusters that run them [WOPW13]. Computational power and energy use translate directly into monetary expense. This in turn implies that there exists a significant financial incentive to build more energy-efficient HPC system, capable of running such applications. One way of improving the energy efficiency of HPC is to use more energy efficient hardware. Field Programmable Gate Arrays (FPGA) are known to offer massively parallel computational power that is also more energy efficient that the more commonly used multi-core CPUs and GPUs for certain workloads [LNLG20] [ZMS+16] but they are also significantly more difficult to program and optimize. More broadly, the underling issue is that of performance portability.

2.1 Performance portability

The cost of energy used to run HPC clusters can easily equal [Ben12] or even surpass the hardware cost. Physically building bigger and better HPC clusters with more powerful and energy efficient hardware is however only half of the story. When single-core Central Processing Units (CPUs) were replaced by their modern, super-scalar and multi-core variants, a great deal of effort was required to adapt compilers and programming frameworks to make adequate use of the additional computational power [vAVSvN09]. Likewise, the adoption of Graphics Processing Units (GPUs) required significant engineering effort [vAVSvN09] [HTWB10]. More recently, Field Programmable Gate Arrays (FPGAs) have become a viable and more energy-efficient alternative for HPC workloads [LNLG20] [VN14] as modern FPGA implementations have larger amounts of on-board memory and better floating-point performance which brings them closer to the raw memory and compute capacity of GPUs. Porting applications to FPGAs is even more difficult task [GBL10] [CLS+08] [HCG+07] than adapting legacy software to run on multicore CPUs and GPUs because the computational abstractions used are very different [HCG+07].

The common theme that describes the main challenges in adapting HPC applications to newer hardware is that of *Performance portability* which describes the ability of a software application to not only execute on a multitude of hardware targets, but to do so efficiently. Numerous compilers and programming language frameworks have been developed to address the issue of performance portability. For applications that target CPUs, languages such as Java, which popularized the phrase *"Write once, execute anywhere"* [Cur98] offer a possible solution. High-level programming languages such as Java are compiled down to a *bytecode* representation, an abstract equivalent of an *assembly language* which can be interpreted by a programming language Virtual Machine (VM) [SN05]. As long as suitable runtime implementation exists for a particular hardware target, a bytecode representation of the application can be readily executed [Cur98]. All interpreters, including those that take bytecode as input incur a performance penalty relative to natively compiled code. Just-in-time (JIT) compilation techniques can recover some of the performance that would otherwise be lost due using *bytecode interpretation* [Ayc03]. JIT compilers can, at times, deliver more performant code and better resource utilisation [Doe03] [JK13] than the more traditional *ahead-of-time* (AOT) compilers [SC19]. Clearly, in the case of CPUs, VMs and JIT compilers deliver good performance.

When it comes to GPUs and FPGAs however, VM implementations and JIT compilers, such as those we cover in section 2.2 can not deliver the same benefits because they require significant general purpose compute capacity that is *logically close* to the application, as we will explain in subsection 2.2.3. Briefly stated, VMs continuously analyse the relationship between application performance and certain properties of input or intermediate data which requires a high-bandwidth connection between the VM, and the computational hardware and memory. The JIT compiler must likewise be able to quickly alter running code.

GPUs and FPGAs have little or no general purpose compute capacity and must rely on the language runtime, executing on the host computer to orchestrate. From a bandwidth and latency perspective the host is too far away. The OpenCL framework [Gro09], for example, offers functionality somewhat equivalent to that of a VM and JIT compiler for GPUs and FPGAs. The programming language used in this case is OpenCL C, a superset of the *C* programming language. The software to be executed implemented as a number of so-called *kernel* functions as well as a *host application* that compiles, invokes and ships data to and from these kernels which are loaded on to the acceleration device. The *host application* is compiled ahead of time. When executed, the host application calls upon the OpenCL runtime to determine the actual hardware target and then triggers the compilation of the kernel functions right before they are executed. The host-side application thus has the potential to optimize the execution schedule and implementation of kernels, in a manner similar to a VM runtime. This potential is however not often realized, as the application developer, rather than the OpenCL runtime, must implement what is effectively an application-specific VM and JIT strategy every time. We contend that this might be the case primarily because *the abstraction level of popular programming languages (PLs)*, as may be found in OpenCL C Java, *inherently matches the execution model* of *super-scalar* and *out-of-order* CPUs. With multi-core CPUs as well as modern GPUs, such PLs no longer map quite as well to the complex hardware features available. There is a *impedance mismatch between* the *computational abstractions* representable in such PLs *and* structure of the *underlying hardware*. Certain language features give developers very granular control over *shared-memory data-structures* and *thread synchronization mechanisms*. This allows developers to fine-tune the performance of their applications. The same language features make it impossible for the compiler to *automatically* work out efficient optimization strategies. Consequently, programmers are expected to annotate their applications with explicit compiler directives which specify how certain applications segments are to be transformed and optimized [KSA+10].

The flexibility and reconfigurability of FPGAs which makes them such energy efficient targets also widens the semantic gap between current programming languages and hardware. High-Level-Synthesis (HLS) [NSP+16] tools seek to address this gap by compiling applications described at a *higher level of abstraction*, in a programming language typically designed for CPUs or GPUs which have *static data pathways*, down into a more traditional Hardware Description Language (HDL) [SG79] such as VHDL and Verilog [Smi96]. The HDL representation can then be used to synthesize an executable implementation for FPGAs. Current generation HLS tools however fail to deliver on the promise of performance portability, as we will see in chapter 3, because the mapping from the high-level languages to the circuit interpretation of an application requires a better approach to automatic parameter space exploration [ZMS+16] than is currently available.

2.2 Taxonomy of parallel computation

We will now explore some of the reasons why programming languages and techniques commonly used to program FPGAs fail to deliver the same degree of performance portability we have come to expect from systems which target CPUs. We will do so by presenting a taxonomy of programming languages and compiler frameworks which is based on the analogy to the three-dimensional space we inhabit.

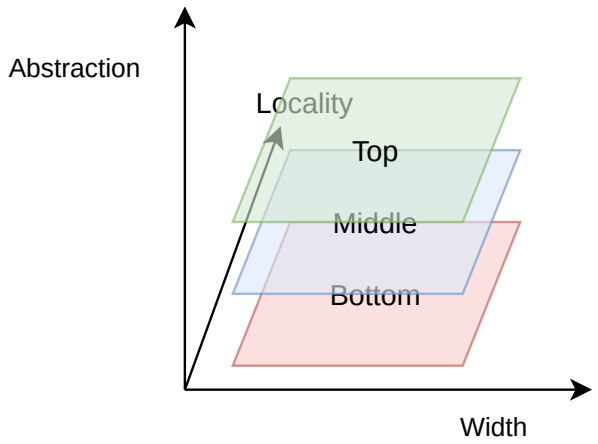


Figure 2.1: Abstract three-dimensional space of programming languages and compilers.

On the bottom-most level of our space, shown with red in Figure 2.1, we include those programming languages that assume a low-level of abstraction. Examples include the VHDL and Verilog HDLs, as well as the assembly languages that were once a popular vehicle for writing performance-critical CPU code. The middle layer shown in blue corresponds to the HLS compilers and frameworks currently used to translate applications written in a high-level language such as Java or OpenCL C, into the languages that inhabit the bottom-most layer. These two first layers will be discussed in the present section. At the top, we find programming languages and frameworks that operate at an even higher level of abstraction, such as the TyTra compiler framework, in the context of which our work rests. This layer warrants closer attention as it is specialized to account for the peculiarities of the TyTra compiler. Consequently, the top-most layer corresponds to section 2.4.

The three dimensions of our taxonomy are as follows. The *Width* axis is used to describe the degree of data parallelism in the target hardware devices. A narrow architecture, in this taxonomy, is one that can only process relatively few data items at once, such as typical consumer-grade CPU. A wide architecture, as found on modern GPUs is in contrast capable of processing many data-items side-by-side, hence our choice of terminology. The *Abstraction* dimension we have already explained as serving the purpose of categorizing programming languages and compilers. *Locality*, the third and final dimension serves to describe both a logical and a physical interpretation of distance in computation as we will explain in subsection 2.2.3.

2.2.1 Narrow and wide: data parallelism

In the context of hardware architectures, we use *width* to describe the number of bits that a *register* may hold. A CPU architecture that has large or *wide* registers may process operations on larger values, or alternatively, multiple data items with a single instruction. This can lead to achieving *super-linear* speed-ups. From a logical perspective, GPUs can be said to have wider registers. This is motivated by the nature of computation in the context graphical operations which is highly parallel. A typical GPU architecture may feature thousands of *compute units*. Each of these, in turn, may feature numerous data registers. All of these registers can be seen as fragments of a single yet wide logical register that can store thousands of data items at once.

Narrow Architectures

An example of a *narrow architecture*, in our terminology, is that of a general purpose Central Processing Unit (CPU). In contrast to wide architectures, general purpose CPUs are designed to process a relatively small number of independent data-items at once.

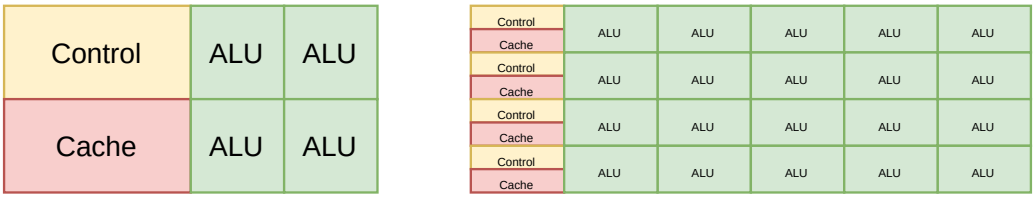


Figure 2.2: CPU (narrow) and GPU (wide) architectures, representing the number of parallel compute units

CPUs, as general purpose processors, can be seen as unspecialized circuits to contrast them to FPGAs. They incorporate a variety of *functional units* that enable them to perform numerous types of operations. A typical CPU architecture may specify that a number of functional units for floating point division, others for multiply-accumulate operations, and yet another set of to deal with a particular video format [SKH+99]. There exists a wide variety of operations that general purpose CPU may be required to perform, yet the surface area available to instantiate functional units is limited by the physics of chip manufacturing. This means that not all such operations may be accelerated through the inclusion of duplicate functional units to support them. Multi-core CPUs may even *share functional units* among the cores under the assumption that not all concurrent tasks will require access to the same units at the same time [Gee05]. Compilers that target general purpose CPUs account for these structural properties. This might explain why the most popular *source programming languages* are also largely *general purpose*, rather than *domain specific* programming languages (DSLs) [Cas18a] [MR13].

Despite the overwhelming popularity of general purpose programming languages, they are not always the best tool for the job. DSLs can provide a more natural way to express certain applications, making application optimization tasks less complex [Hud97]. The set of *optimizing transformations* that must be considered by an optimizing compiler for general purpose programming languages is naturally larger than what may be expected of a domain specific compiler. At the same time, compiler developers are naturally risk-averse. Breaking changes in compiler semantics can affect a larger number of end-users. Safety, being the number one priority, implies that the complexity of the compiler implementation must be mitigated. One way to resolve the tension between performance and safety is to reduce the set of automatic optimizing transformations the compiler considers. **Compiler flags** are options passed to the compiler by the programmer that enable or parametrize certain optimization passes. It is a well known that compiler flag selection is a non-trivial task, as many compiler optimizations can have conflicting effects [Bot12].

```
−ansi −std=standard −fgnu89−inline
−fpermitted−flt−eval−methods=standard
−aux−info filename −fallow−parameterless−variadic−functions
[..]
```

Listing 2.1: A few GCC Flags.

Compiler flags can be used to implement *coarse* optimization strategies. A set of compiler flags can be applied to an entire source-code module or file and so have a global effect on the compiler's interpretation of each such module. The *ansi* flag [WVH04] in Listing 2.1, for example, dictates what sub-set of a source programming language is considered valid. Certain optimizing transformations may take *enumerable* parameters. These may be likened to compiler flags because their parameters usually feature a reduced cardinality. A small numeric parameter may be regarded as equivalent to a small number of boolean compiler flags. In standard C compilers the *-std* flag, for example, can take one of a a fixed number of values that dictates what *standard C++ version* should be used when interpreting the source code. Valid instances of this flag include *-std=c++11*, *-std=c++1*, and so on. Compiler or **pragma directives** such as those used to specify loop transformations [KF19] are somewhat more *specific* than compiler flags. Pragmas are *language constructs* that are added to the source-code representation of an application in specific places.

```
int main() {
const int size = 256; double sinTable[size];
#pragma omp simd
for(int n=0; n<size; ++n) {
 sinTable[n] = std::sin(2 * M_PI * n / size);
}
```

Listing 2.2: Example: Use of OpenMP SIMD pragma directive

Pragma declarations alter the compilation process by instructing the compiler to processes the input in differently. The name evokes a *pragmatic* approach to programming that, when used appropriately can improve the performance of an application. The effect of pragmas directives is *local* as the example in Figure 2.3 shows. It contains a pragma directive (green) that is applied to the *basic block* (unit of code) corresponding to the *for loop* that follows.

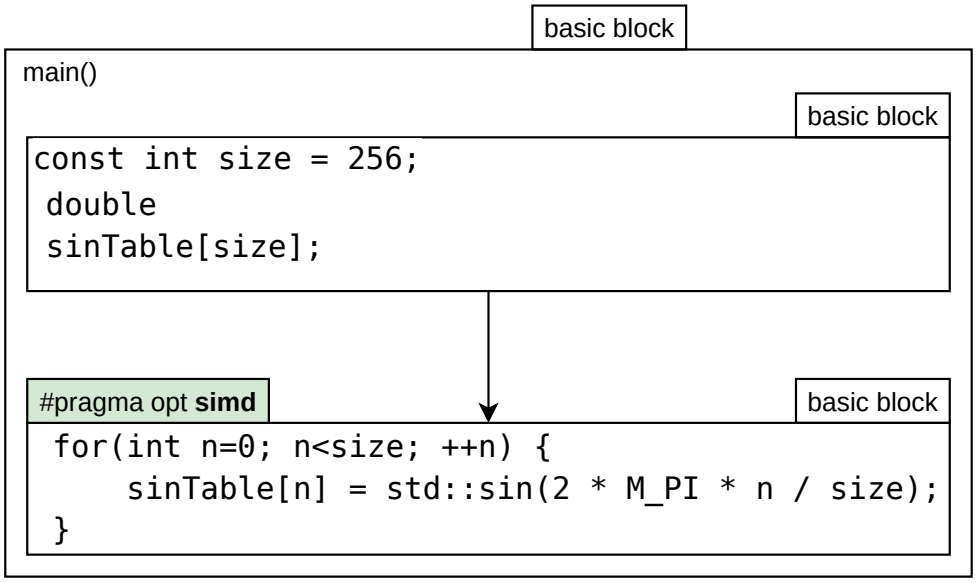


Figure 2.3: Pragma directives have a local effect, typically function, variable scope or basic-block level.

Iterative optimization can be used to search for optimal sets of compiler optimizations. The basic methodology implies adding *optimizing transformations* to a candidate set, recompiling the application and checking if the newly expanded optimization set leads to better results. *Iterative optimization*, appears to be frequently used in related work. We speculate that this might be the case because the *operational semantics view of program transformation* makes it difficult to accurately predicting the performance impact program transformations. In the absence of a fast *cost-performance model*, iterative optimization is used to approximate an optimal solution by trying out different optimization schedules without having to explore the entire search-space.

When using compiler flags, expanding the candidate set simply equates the addition of a previously unconsidered flag. At an abstract level, we can think of the empty set of compiler flags as the base case in an inductive proof. Adding a fresh compiler flag that improves the performance of the application can then be seen as the inductive case for a proof that an optimal set of transformations exists. In the case of *pragma directives*, one must not only enumerate all possible directive parameters, but also the locations in the application's source code where they are to be applied. This approach is taken by current generation HLS tools as well as by the Lift compiler which is our most closely related work, as we will show in chapter 3.

Wide architectures

We say that GPUs, FPGAs and Very Long Instruction Word (VLIW) processors have a *wide architecture* because they make use of wide registers to store and process many data items at once. In the case of GPUs and VLIW processors, the architecture usually specifies a *single instruction multiple data* (SIMD) approach to processing. In the case of FPGAs, there are two approaches. The implemented circuit can be replicated as many times as will fit on the device, allowing multiple instances of the application to be executed concurrently. The other approach is to replicate the sub-circuits that implement the computational circuits such that multiple data items from a stream or list of values, can be processed in parallel, effectively implementing a local, customized SIMD processors.

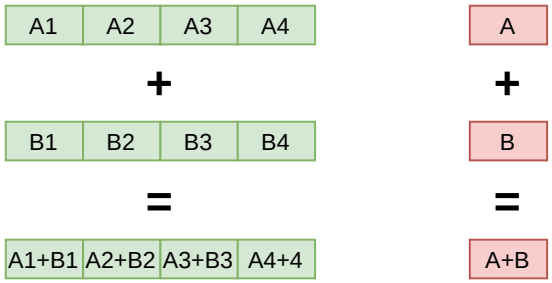


Figure 2.4: SIMD (left) vs scalar (right) addition.

These *wide architectures* are often optimized for *domain specific* workloads where control-flow divergence is not a major factor in the algorithm being executed. As SIMD architectures share an *instruction decoder* amongst many *functional units* [BS91], control-flow divergence can be a major issue. Graphical Processing Units (GPUs), are normally used to generate computer graphics provide massive amounts parallelism that can be used to accelerate many computational tasks [JLBF10]. This is possible because both graphically intensive games and computational workloads such as weather prediction [SGPJ12] or financial simulation packages [GKSC13], depend, performance-wise, on a small set of highly parallel functional units, that typically accelerate operations such as matrix-matrix multiplication [FSH04]. Early compilers and programming languages that targeted GPUs came under the umbrella term of *General-Purpose computation on Graphics Processing Units* or *GPGPU* in short. Two-dimensional graphics operations are essentially matrix operations. Likewise, three-dimensional transformations including *translation and rotation* can be expressed as matrix operations. Early GPGPU implementations relied on hardware and programming constructs/languages that were less amenable to general purpose computation [WL08]. As hardware developed, **frameworks for general purpose computation**, on massively parallel architectures such as **OpenCL**, became available and allowed applications developers to accelerate and deploy their computation to a wide range of platforms including CPUs, GPUs and even FPGAs [CAD+12]. In section 3.1.1 we will go over related work consisting of optimizing compiler frameworks including OpenCL, for wide architectures.

2.2.2 High and low: abstraction level

FPGA programming is recognized to be a difficult task and often requires the expertise of hardware programmers [BRS13]. There are many programming languages, compilers and synthesis frameworks available to reduce the burden placed on the programmer. Each of these brings specific trade-offs in performance, programmability and complexity. Broadly speaking, we can classify these according to the level of abstraction used to represent computation. Field Programmable Gate Arrays are the descendants of CPLDs [BR96]. Both are seen as *configurable circuits* and as such, at the lower-end of the abstraction spectrum, FPGA programming tools take a circuit view of computation. At the higher-end, FPGA programming tools look more like traditional software compilers.

A simple FPGA implementation might consist of a *configurable switch* that routes data between the *lookup tables* (LUTs) that encode the behaviour of logical circuit components, as shown in Figure 2.5. The LUTs embedded within the configurable switch fabric can be made to emulate any discreet function that is small enough to fit.

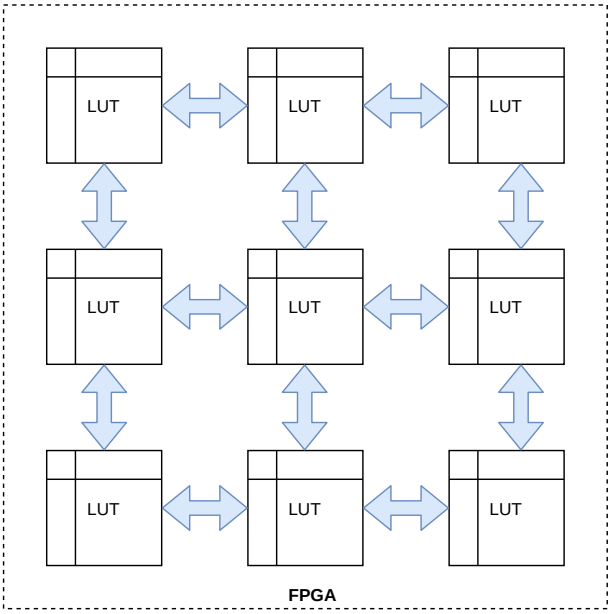


Figure 2.5: FPGAs: A switched sea of lookup tables.

Because of this view of FPGAs as circuits, the programming languages used are usually called *Hardware Description Languages* (HDL). The two best know instances of HDLs are *Verilog* and *VHDL*. The primitive constructions used in HDLs are named using circuit design terminology but can be likened to programming language constructs. *Wires* that connect components can be likened to function calls because they carry the input parameters to a circuit. *Registers* can be seen as intermediary value stores and so can be linked to *variables*. *Ports* can be seen as defining an *interface* and can thus be likened to *function signatures*.

There are also numerous differences that warrant the distinction between HDLs and other types of programming languages.

- Mainstream imperative programming languages specify a sequential instruction evaluation order. Functional programming languages are also compiled down to machine instructions which are executed sequentially. In contrast, HDLs have an entirely parallel evaluation strategy. The application defines a circuit in which all components are to process their inputs as soon as these become available.
- With HDLs program behaviour must be specified at a much more granular level. In contrast, software applications can be built up in using many layers of abstraction. Certain details pertaining to the application can be resolved at run-time, meaning that a software developer may not need to specify the entire solution at all.
- Software developers use the various *exception mechanisms* provided by their programming language of choice. In hardware, there is no generic *catch-all* exception mechanism that can trap unexpected errors. Every possible execution path must be explicitly accounted for.
- *Wires* may be likened to function calls, however, the programmer must effectively decide upon and implement the right *calling convention* for each such "function" being called.
- *Registers* that implement intermediary value stores materialize every time, unlike variables used in imperative programming languages that may be represented in the compiled application, depending on the choices made by the optimizing compiler.
- *Ports* defined using an HDL are also more likely to be used only where the interface they define is a complete match for the use-case at hand. In other programming languages, interfaces can be extended abstractly, and they can be implemented by multiple classes in different ways.

These differences are particularly troublesome in the context of HPC and scientific computing where the size and complexity of the applications is significant. Ideally, programmers should be able to use the abstractions and tools they are already familiar and the compiler should be smart enough to automatically work out the best circuit layout that matches the application's specification. To some extent, this is what High Level Synthesis tools offer: a way around the programmability issue raised by HDLs that relies on an automated translation process from high-level languages down to HDLs, known as *behavioural synthesis*. A particular class of HLS tools are so-called *C-to-Gates tools* because the front-end programming language used often resemble C. In subsection 3.1.1 we argue that imperative programming languages, including C derivatives, are a poor choice for HLS as they are semantically misaligned with the HDL representations they must generate. Fortunately, the *C-to-Gates* flavour of HLS tools are only one possible solution out of many.

2.2.3 Near and far: locality

On the **Locality** axis in Figure 2.1 we differentiate between languages designed to explore the parallelism opportunities found *locally* within any one machine, and those languages and frameworks that provide *distributed computing abstractions*. As computing science progresses, the lines between what constitutes a *singular machine* or computer, and what constitutes a *cluster* are starting to blur.

- Modern CPUs deliver greater economies of scale by implementing *chiplet designs*. Rather than attempting to fabricate large, defect-free CPUs featuring many processing cores, vendors produce smaller *chiplets* embedding a relatively few cores. Chiplets are then assembled on top of *silicon interposers* that play the role similar to that of a *data network*, keeping CPU caches, memory and peripheral devices synchronized.

- Hardware accelerators that deliver orders of magnitude more computational power than the CPU are now commonplace. GPUs and FPGAs expansion cards are used for a variety of tasks, ranging from graphics rendering to database acceleration and video encoding. All of these devices are connected to the CPU, and indeed amongst themselves using the PCI Express (PCIe) interface which is effectively a short-distance, high-bandwidth switched network. Devices exchange *Transaction Layer Packets* (TLP) over these networks. Specialized *routers* called *non-transparent bridges* can even allow multiple workstations to share acceleration hardware.

Programmers stand to benefit greatly from the convergence of these technologies. If programming languages and compilers evolve such that one may use a single set of abstractions to reason about computation at any scale, then this greatly simplifies the task of writing safe and efficient applications by lowering cognitive load and the burden of testing.

Whenever discussing multi-core machines with hardware accelerators, or perhaps datacenters containing thousands of machines, the most important factor that determines performance is that of *distance*. Physical distance dictates which data movement operations are permissible. The ultimate limit in the universe, the *speed of light*, dictates hard minimum bounds for latency. Beyond physical distance, we must also consider the notion of *logical distance* in computation. The *address* of a value stored in memory can be used as a measure of distance from some other location in memory. We may speak of the distance between two subsequent *input values* to some application, and likewise of some memory address distance between two instructions waiting to be executed. Efficient application execution requires that both kinds of distances, logical and physical, be minimized. Furthermore, *distance uniformity* matters greatly. High memory access latency, if uniform, can be detrimental to performance but in most cases it can also be masked through careful task scheduling. Sporadic and non-deterministic latency increases, on the other hand, can be impossible to mitigate.

Locality of reference as a term, describes a relationship between the temporal ordering of instructions waiting to be executed on the one hand, and the spatial ordering of input data in memory. Every concrete layout for input data implies a specific optimal temporal ordering of instructions. The exact needed mapping can however be difficult to find in practice. Hardware vendors persist in maintaining a layer of opacity in their memory *cache* implementations, for example. This makes it difficult to effectively model their behaviour.

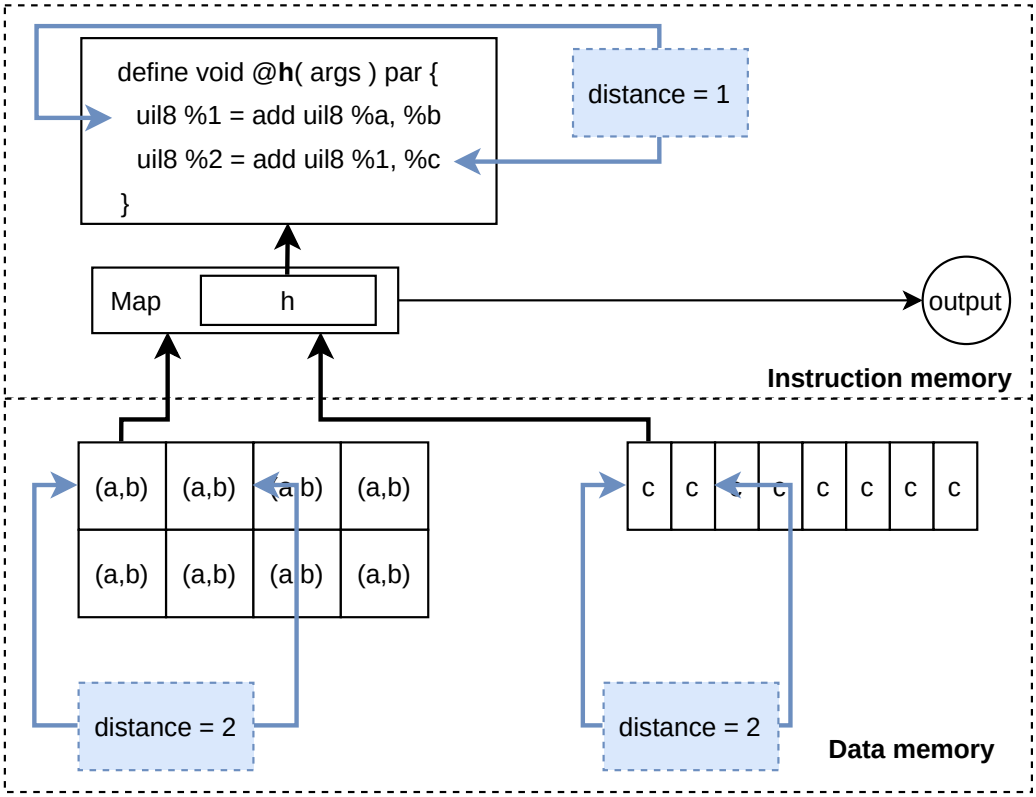


Figure 2.6: Harvard Architectures and logical distance.

Assuming the ever-present *Harvard architecture*, where instructions are *logically separated* from data, there has to be a way for instructions to *reference* the data they act upon. The issue of *locality of reference* thus relates to the discrepancy in distance between subsequent instructions to be executed, and subsequent items of data these instructions act upon. This issue is further compounded by the many layers of abstraction introduced by the algorithms, programming languages and operating systems that form the application stack. Problems and the algorithms that solve them are, for now at least, specified almost entirely by humans. Programmers are free to build up and structure these problems in a way that makes sense, that is to say has *meaning* or *semantics*,in the problem domain they are most comfortable utilizing. Input data and intermediate results can be organized into *stacks*, *heaps*, *tries* and any other variety of data structure imaginable. This flexibility in structuring data directly translates into efficient implementations, as some data structures naturally pair with certain algorithms.

Consider the case of *sorting an array* of values, perhaps using the well-known *merge-sort* algorithm. The efficiency of *merge-sort* stems entirely from decomposing the problem, resolving each sub-problem and merging the partial solutions to form the intermediate and final outputs. The algorithm can thus be seen as an operation over a *tree* representation of intermediate solutions. In contrast, the computational hardware used may assume an entirely different, more linear, execution structure. Main memory is often thought of as having a linear layout. The abstract notion of an address-space is what gives this linear view of access to data. Caching and branch prediction mechanisms rely on this linear view of memory addresses to enhance application performance, however they can also obscure the compiler's view of potential optimizations. Whereas the logical view of memory is linear, the effects of cache mechanisms is that they provide non-linear speed-ups. If the caching mechanism fits the inherent *access pattern* of the application, large efficiency gains can occur.

As technology advances, simple caching mechanisms such as *first in first out* or *least recently used* stores, that are easy to implement and model, give way to more obscure implementations. Take for example AMD's branch prediction implementation that supposedly employs a *neural network*, modelling the effects of which seems improbable for now. Further complicating the issue of determining and mitigating the effects of *locality of reference* within a compiler, there can be many nested layers of obscure caching mechanism within the processor, memory controller, or even the network when discussing distributed computation.

Breaking down the problem of *locality of reference* we can have a look at how programming languages and compilers deal with this problem when the components are relatively *near* to one-another. Dealing only with caching layers present on one work-station, and eliding the discussion on network latency and throughput somewhat simplifies the discussion. Compiler writers are acutely aware of locality issues. It is likely the most common drain of performance one can encounter. The relative performance difference between an application that is fully aligned with the caching mechanism, that is to say every memory access is a *cache hit*, and one that is not can be absolutely staggering. A more pronounced performance differential can be seen between accesses to main memory and a *solid state drive* (SSD) or an SSD and a mechanical hard-drive.

Hardware	Access time	Relative latency
CPU Registers	1 ns	1x
CPU Cache	10 ns	10x
Main Memory	100 ns	100x
Solid State Disk	100 μs	1000000x
Magnetic Disk	100 ms	1000000000x
Magnetic Tape	10 s	1e+10x

Table 2.1: Memory access latency.

It feels odd having to include such a table in a thesis document in 2020, yet the cold harsh truth is that access times are absolutely fundamental to the issue of performance. The fact that these technologies give access times that are orders of magnitude apart all boils down to some notion of distance. It may be the physical distance that the read head has to cover in a HDD, or it may be the logical distance between row-buffer operations in RAM, yet it is distance nonetheless. Modern compilers must account for all of these different types of distance. There are two major types of strategies that try to minimize access times.

- **Computation reordering.** The instructions that make up the applications are ordered or *scheduled* such that they happen to execute at the appropriate time when data becomes available in the lowest access time memory.

- **Data reordering.** Data is processed and reordered such that the hardware can efficiently access the required memory locations as quickly as the application's instructions demand.

Instruction scheduling is typically performed at compile-time, and thus falls under the remit of a traditional compiler. For *Very long instruction word* (VLIW) architectures these must schedule every instruction accurately, in what is known as *software pipe-lining* [Lam88] because the processor can not reorder instructions on the fly. Data reordering, on the other hand, can be performed by the compiler but more often falls under the remit of a *runtime* or *standard* library as they are sometimes called. Reordering data, rather than instructions, means that certain safety invariants can be maintained more easily: the application's code can be checked statically, before execution. At the same time, the exact type of input data may only be known at execution time. Reordering instructions would require an expensive recompilation, as opposed to simply copying values between memory regions. More often, data ordering is specified, statically, by the programmer, after a thorough set of profiling runs with representative examples of application input. Computation and data reordering can both be automated and used together. Modern compilers, particularly those that make use of *Just-In-Time* (JIT) compilation techniques, as is the case with the *Java Virtual Machines* (JVM) will perform automated runtime profiling which in turn triggers the recompilation and optimization of the most demanding regions of code in the application. The same type of instruction reordering can also happen in hardware. Current generation x86 CPUs implement a *Reduced Instruction Set Computer* architecture that is hidden away behind a *facade*. The *x86* instruction set that is exposed to the user is not only translated but also *transformed* by the compatibility layer as well as the *out-of-order execution mechanisms* implemented deep within the CPU. The translation and transformations applied are *programmable behaviour* specified through *microcode updates*. Effectively they are a hardware assisted JIT compiler, albeit a hidden one.

Compilers implement *transformations* that deal with both data and instruction reordering. *Polyhedral transformations* help by reordering access to memory, in a manner consistent with the computation, such that the locality of reference is improved. The compiler reorders computation, the individual instructions and statements that access data, based on identified data dependencies. *Inlining* serves a number of roles in a typical compiler. Besides enabling other transformations by co-locating blocks of code and increasing sharing, it also removes indirections caused by statically computable blocks, increasing the locality of reference of instructions to be executed. *Work stealing* mechanisms, typically implemented as part of an application's run-time support library allow computation to be dynamically rescheduled, for example by having a *thread* steal work items from another thread's work queue. This is one example of a fully dynamic approach, that can easily have an adverse effect on performance if improperly implemented.

Distributing computation across a networked cluster of compute devices adds another layer of complexity on top of the already convoluted issue of locality. There are a number of popular distributed computation frameworks in use within the realm of scientific computation. The large majority of these provide a thin layer of abstraction over the networking layer and assume that the programmer will respect the *style* and *protocols* established through documentation. In many ways, this layered approach simplifies matters greatly. Programmers can take an existing body of code and *augment it* by inserting additional calls to the distributed computation framework. *Message Passing Interface* (MPI) [SGO+98] is one example of such a framework. The MPI framework is centred around the idea of encapsulating local state and explicitly updating participants through the sending and receiving of explicit messages. Having to explicitly encode updates to the global state in messages forces programmers to think about the ordering and timing of these events, in stark contrast to the shared memory model. Distributed computation frameworks such as MPI are sometimes used to take advantage of parallel hardware even if it is not distributed amongst multiple machines. A message passed between two local processes is conceptually the same as a message passed between two networked processes. In practice however, this may be somewhat inefficient due to the overheads involved in creating a message structure and *"sending it"* across a virtual network socket. *OpenMP* [Cha01] is an implementation of the multithreading approach to parallel programming that is quite different from the message passing. It has been called the *industry standard API for shared-memory programs* [DM98], and can be quicker than MPI if the underlying architecture supports it. *Hybrid* approaches that utilize both methodology also exist [MRRP11]. Programmers often make use of OpenMP or similar frameworks to schedule local concurrent work, whilst explicitly distributing larger tasks amongst networked machines or perhaps even *virtual machines* running on physically distinct CPU cores, using MPI. Although both approaches solve the same conceptual issue: that of scheduling and distributing parallel computation, programmers must become familiar with, develop and test their applications using both.

2.2.4 The middle way

We've seen how the parallel architectures and programming solutions covered thus far rely on very different programming abstractions. These architectures can be programmed and optimized, arguably rather inefficiently, using the same or very similar programming languages. In all cases, the optimization process seems to be treated as an afterthought. The programmer first specifies the application and only then directs the compiler or HLS tool, using flags or pragma directives, towards a more efficient optimization schedule. In section 2.4 we will have a close look at the TyTra compiler which, in contrast to this approach to application specification and optimization, relies on Design Space Exploration to expose a design-space of program transformations that can be automatically searched, rather than manually traversed using programmer-provided annotations. Before moving on to the specifics of the TyTra compiler framework, we will have a closer look at what ties the underlying models of computation for all of these platforms.

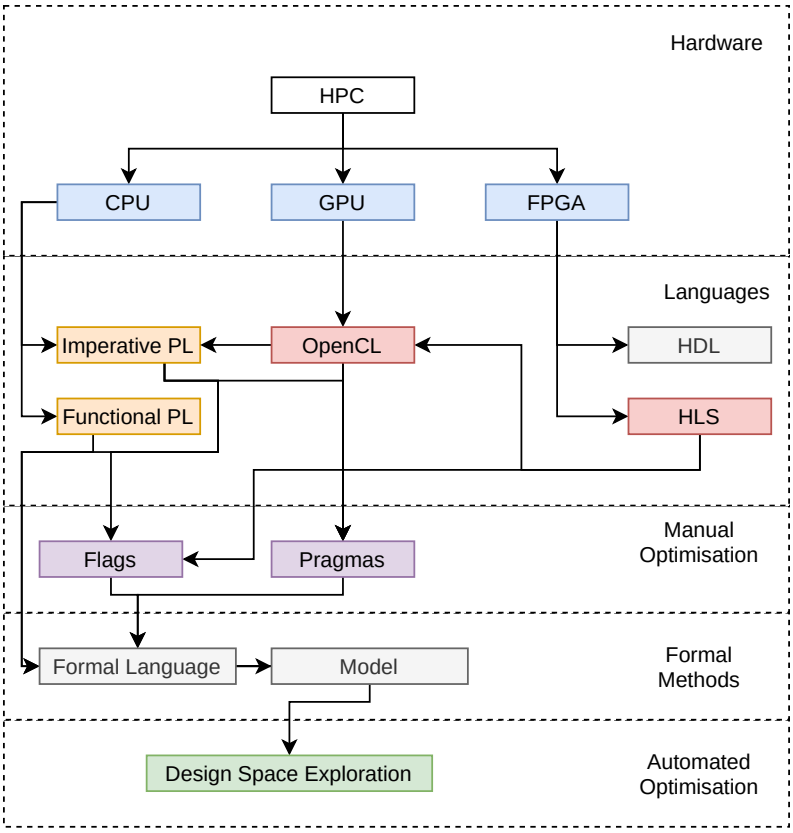


Figure 2.7: Relations between parallel architectures and tools.

Trivially, every programming language is a formal language. Compiler flags and pragma directives can also be seen as the constructs of an additional formal language. All of these formal languages can be assigned one ore more models of computation. In section 2.3 we will see how all of these parallel architectures can be related by their underlying models of computation.

2.3 Formal methods

Programming languages are *formal languages*. We can distinguish between different programming languages by discussing either their *syntax* or the *semantics* assigned to them. Of the two, the concrete syntax of a programming language is what defines a formal language.

> **Definition:** Formal Language.
> A *Formal Language* is a collection of words over some alphabet. The alphabet is a *Set* of symbols $\mathbb{A}$. A word is a concatenation of one ore more symbols in the alphabet $\mathbb{A}$. A language is thus representable as some subset of words $\mathbb{L} \subset \mathbb{A}^*$.

Depending on the level of abstraction required, as well as space and aesthetic considerations, the alphabet of a formal language can either be a literal alphabet having single character symbols, or as is more often the case, a set of *key-words* where are considered to be *atomic* symbols and other characters denoting various operators or *scope* limits. The size and complexity of a formal language's alphabet can vary wildly. Most general purpose programming languages restrict themselves to words made up of the ASCII character set, though some like *Agda* may incorporate *Unicode* symbols to evoke the mathematical underpinnings of certain operators and constructs. The concrete syntax of a programming language is primarily designed for ease of use and interaction with a real human being. There is also another notion, that of an *abstract syntax* that can be used to give a programming language an interpretation. The abstract syntax retains only the *intent* behind an application specification, that which is required by the compiler to transform the application into something that will ultimately execute on real hardware. The term *semantics* denotes the *meaning* of a word or a phrase in some language.

The *semantics* attached to a programming language are the mapping of the language's syntax into some *domain of interpretation*. If the interpretation domain is formed of *commands* or other constructs that can be likened to concrete *actions* in the real world, we classify the language as having *operational semantics*. If on the other hand, the domain of interpretation is more abstract, perhaps as a result of being defined by some other formal language, then we say the language has *denotational semantics*. The abstract syntax of a language too is often highly correlated with the semantics given to the language. Imperative PLs for example, often provided with *operational semantics* imply an abstract syntax representation where the nodes roughly correlate to *commands*. Functional programming languages, tend to favour even more abstract representations that can better relate the syntax to the *denotational semantics* of *higher order functions*.

2.3.1 Models of computation

We said that programming languages are *formal languages* and that the syntax specifies how we may form and structure applications whilst semantics allow us to map the syntax into a domain of interpretation. To do so, we must distinguish *valid* syntactical constructions, formally called *sentences*, from arbitrary words.

> **Definition:** Sentence.
> A *sentence* is a boolean valued, finite sequence of symbol, with no free variables, that is taken from a given alphabet and is part of a formal language.

A *sentence* in this context may also be called a *theorem*. A set of sentences, taken as a collective whole, define a formal theory.

> **Definition:** Formal Theory.
> A *formal theory* is a set of sentences in a formal language.

Let us look at a concrete example, that of defining the syntax and semantics of a simple language which corresponds to the notion of natural numbers. Intuitively we know that natural numbers behave in a certain way: we add or multiply them to yield other natural numbers. We can specify this using a formal theory by relating the words in the language of natural numbers. This example is so well known that it bears its own name: *Hutton's Razor*, which we will denote simply $\mathbb{H}$. We will represent the syntax for $\mathbb{H}$ using the Extended Backus-Naur form (EBNF) notation below.

$$
\begin{aligned}
\langle N \rangle \quad &::= 0 \\
&\mid \text{`}\texttt{successor}\text{'}\ \langle N \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle expr \rangle \quad &::= \langle N \rangle \\
&\mid \langle expr \rangle\ \text{`}+\text{'}\ \langle expr \rangle
\end{aligned}
$$

In this particular representation of natural number, the formal language is defined as having two classes of terms. The first class is that of natural numbers N. These terms are defined as being either the literal value 0, or any term constructed by (perhaps iteratively) applying the *successor* constructor. The second class of terms is that of expressions. An expression can either be a natural number, or the addition of two such other expressions. If we also provide a way to interpret the expressions that can be constructed using this language, then we can define a *computational model* of natural number addition.

Given a formal language L, a set M together with an interpretation (or evaluation) of L into M implies that M is a model of a theory T if for every sentence in T it's interpretation in M is $True$. Note that the set M may have many more values than the *Boolean* set containing $True$ and $False$ which was used to define a formal sentence. By expanding the interpretation *co-domain*, that is moving away from the *Boolean* set to a richer one, we can derive judgements that hold a deeper, context-dependent meaning. Let us take the set of natural numbers literals $\mathbb{N} = \{0,1,2,3...\}$ the set M used to construct a model of $\mathbb{H}$. We will use the Oxford brackets as the operator that assigns an interpretation to a word in our formal language. The subscript to the operator denotes a domain of interpretation, in this case $M = \mathbb{N}$.

$$[\![0]\!]_{\mathbb{N}} = 0$$
$$[\![successor(n)]\!]_{\mathbb{N}} = [\![n]\!]_{\mathbb{N}} + 1$$

Simple as this example may be, the theory of natural numbers represents the foundation of inductive definitions and recursive computation. The example above assigns the literal value 0 from the $\mathbb{N}$ set as the meaning of our 0 term. It also gives the semantics of the *successor* operation as a recursive interpretation. Any $successor(n)$ term can now be interpreted by first finding the meaning of the sub-term n and then making use of the $+1$ operation in the domain of interpretation $\mathbb{N}$. Using the same technique we can define the semantics of the addition operator in $\mathbb{H}$.

$$\forall x.\ \forall y.\ [\![successor\,(x) + y]\!]_{\mathbb{N}} = [\![successor\,(x + y)]\!]_{\mathbb{N}}$$

The semantics we have given to the *successor* and $+$ operations in our formal language are defined by pattern-matching: we identified certain parts of the left-hand side (LHS) of an equation with specific terms in the language, and then gave them the meaning of the expression on the right-hand side (RHS). We now see the Oxford brackets more abstractly as an evaluation function that gives, to some term t from a domain $\mathbb{A}$, semantics in a codomain $\mathbb{B}$, denoted $[\![]\!]_{\mathbb{B}} : \mathbb{A} \to \mathbb{B}$. This notion of evaluation is frequently used to provide models of computation for functional programming languages where the meaning of the algorithm or application being computed is described by the *functional dependency* between *inputs* and *outputs*. Pattern-matching can be used to accurately describe the semantics of functional languages that are said to be *pure*, meaning that the outcome of the computation depends on nothing else but the expression being pattern-matched. In the case of imperative programming languages, however, the computation may also depend on other factors that collectively make up the *state* of an application.

We will now have a closer look at some of the more common models of computation used in both imperative and functional programming languages, to better understand their strengths and weaknesses as well as how the may be better related.

2.3.2 Imperative languages

Imperative programming languages describe computation as the *sequencing* of commands. That is to say, programmers describe their applications by specifying the order in which *instructions* are to be applied to the application's *state*, replacing and transforming it until the final state, the *output*, has been determined. Applications described through the ordering of state manipulating instructions are said to be given *operational* or *small-step* semantics. An application execution state is given by the memory contents of the machine it is executing on. It can be denoted as a context Γ. An *impure* assignment action can then be modelled as a *transition function* between contexts.

$$\Gamma_0 \rightarrow \Gamma_1$$

An assignment operation might relate the initial state of the context Γ_0 to the new context following the assignment Γ_1. The initial context could be empty, or it may already contain a set of *name-value* pairs. Assuming Γ_0 is the empty context, then executing an expression such as $answer := 42$ will yield a new context Γ_1 that contains the pair $(answer, 42)$. This idea of sequentially transforming an *application state* lends itself to sequential models of computation such as those provided by *Finite State Automatons* (FSA) (equivalently *Finite State Machines* (FSM)), *Turing Machines* and *Pushdown Automatons*. Of these, *Finite State Machines* model imperative computation quite closely. An abstract state machine specifies a set of states S, a particular state s_0 denotes the *initial state*. A transition function $\delta : S \times \Sigma \rightarrow S$ specifies how the machine moves from one state to another, for every given current state, and some input from the alphabet set Σ. A sub-set $F \subset S$ of *final states* denote *termination*. All-together, a deterministic Finite State Machine model is given by the $(\Sigma, S, s_0, \delta, F)$ tuple.

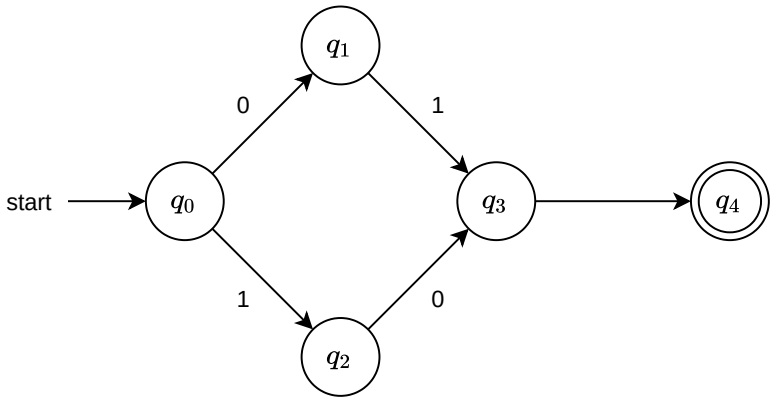


Figure 2.8: Graphical depiction of a finite state machine.

FSMs can also be represented as graphs. The arrows between nodes denote the transitions specified by δ and are labelled with the input required to perform the transition. The initial state is marked with an incoming arrow, either source-less or marked with "start", whilst final states are denoted using a double circle to distinguish them from other states.

The graphical representation of FSMs is appealing because it is easy to check the execution of some computation by walking tracing edges within the graph. These can also be used to paint an intuitive picture of composition.

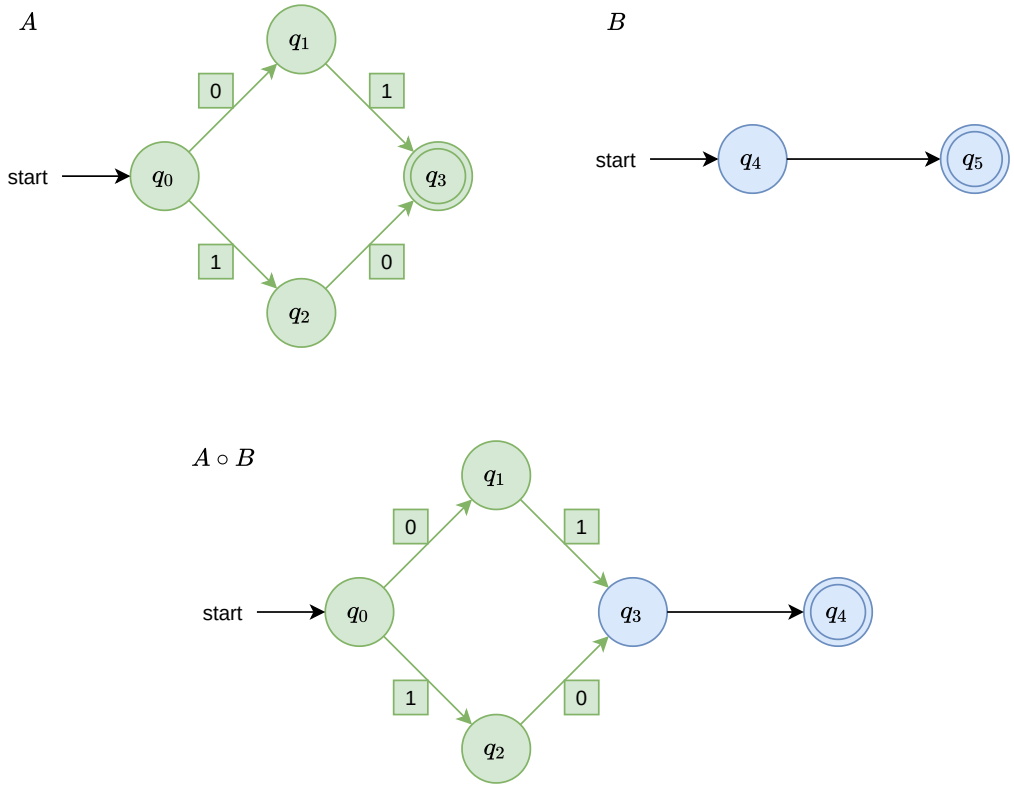


Figure 2.9: Composition of finite state machines.

The issue with FSAs is that they are *verbose*, as far as computational models go. Representing every possible state and transition is quite simply not a feasible approach in the case of larger applications. The internal workings of an alarm clock or perhaps a washing machine can be readily understood by looking at a graphical representation, yet the same does not hold true for numerical simulation applications. That is not to say that there is no benefit in the latter case. Certain *aspects* of larger applications can be readily described and understood using finite state machines. Consider, for example, the need to model the *memory use* of an application. This can be done using an FSA representation. The use of FSMs to track certain application properties is quite widespread that. Certain programming languages include FSM specific keywords and operations in their very specification. One example is that of the *nesC* language supported by the *TinyOS* programming environment. The programming model behind this language specifies that FSMs are derived from syntactic descriptions of *event-driven computation* [KMG08]. *Push-down Automata* are effectively Finite State Machines that come attached with a *stack*. Having access to a stack for storage means we can reduce the number of state nodes. This helps to lower verbosity when compared to the simple FSM model. Reasoning about large applications is still cumbersome, however.

2.3.3 Functional languages

The conceptual difference between expressions and statements is highly representative of the difference between functional and imperative programming languages, however many programming languages implement constructs from both paradigms.

- **Expressions** are compositions of terms that can be evaluated to yield *values*. Expression evaluation is *pure* and *side-effect free*.
- **Statements** are a representation of instructions that manipulate an application's state. Binding statements for example, change the state of an application such that a variable's name is associated to a new value.

A defining feature of functional programming languages (FPLs) is that they represent computation as *expressions* rather than *statements*. From the point of view of FP, both *data values* and *functions* are simply *expressions*, *first class citizens* of the language. Variables in FPLs are represented as *immutable values*. There are no *statements* to speak of, so there is no way to alter the contents of variables. Instead, new variables are created to *simulate* the effect of assignment statements. Stale variables are simply removed by the *garbage collector*. **Continuation Passing Style** (CPS) is a functional programming approach to dealing with the lack of *statements*. Consider the following listing.

```
#include <stdio.h>
int foo ( int number ) { return number + 3; }
void main() {
 int temp = foo ( 5 );

 printf ("The new number is %d", temp);
}
```

Listing 2.3: A simple C application.

Rather than represent computation as a series of statements that manipulate state, whenever we *compute a new state* we pass it on to a *continuation function* which represents *the remainder of computation* yet to be performed. Here is how our example might look when using CPS.

```
#include <stdio.h>
void continuation( int number) { printf ("The new number is %d", temp); }
int foo ( int number, void* continuation ) {
 *continuation(number + 3);
}
void main() { foo (5); }
```

Listing 2.4: Example C application using continuation passing style.

This style of programming may seem odd in an imperative PL as presented in Listing 2.3 and Listing 2.4, however, it is crucial for certain optimizations, such as the removal of the intermediate value (stored in *temp*) shown in those listings. The use of CPS is more natural in functional programming languages where data and functions are expressions and can be passed is as values, without the need for explicit pointer manipulation. This can be seen in Listing 2.5 where the same example is shown as expressed in Haskell, a pure functional programming language.

```
main :: IO ()
main = ( \number -> print ( number + 3 )) 5
```

Listing 2.5: Continuation passing style in Haskell.

The syntax of functional programming languages is often chosen to evoke the familiar notion of *mathematical functions*. On a semantic level, many functional languages can be related to mathematical formalisms such as the *lambda calculus* or *combinatory logic*. Such formalisms lend functional programming the very useful property of *compositionality*. In FPLs, partial programs can be implemented as abstract functions. These in turn can be composed to form more complex solutions whilst minimizing the risk of introducing errors. Let us now have a closer look at one of the mathematical formalisms just mentioned, the *lambda calculus*, It is a universal model of computation that can be described using just three term constructors.

- **Variables:** x
- **Lambda abstractions:** $\lambda x\,.\,x$
- **Applications:** $e\ e$

These express *functional abstraction*, *application* and enable *composition*. The usefulness and simplicity of the lambda calculus mean that it can be used as a *core language*. Functional programming languages that feature more complicated computational abstractions can be *compiled down* to the lambda calculus or one of its many extensions. The sample Haskell code we gave to showcase CPS, in Listing 2.5 and be compiled down to the following lambda calculus expression.

$$(\lambda\,.\,x \rightarrow x+3)\ 5$$
$$(5+3)$$
$$8$$

Figure 2.10: Lambda abstraction and application. Execution Trace.

The lambda calculus execution model is quite simple. Code can be executed by progressively reducing the lambda calculus terms until reduction is no longer possible.

There are numerous other calculi bedsides the lambda calculus. The *SKI combinator calculus* is another example that also has three term constructors:

- The identity combinator , **I** that returns its argument: $I\ x = x$
- The constant combinator, **K** that returns only the first of two arguments: $K\ x\ y = x$
- The substitution operator , **S** which initially applies the first argument to the third, takes the result and applies it to the result of applying the second argument to the third: $S\ x\ y\ z = (x\ z)\ (y\ z)$

What is more interesting to FPGA programmers is that *SKI* combinators have a straightforward mapping to the representation of computation as circuits. The **I** and **K** combinators for samples might be implemented by the following circuits. The equivalent circuit to the **I** combinator is a wire that carries the untainted signal and the **K** combinator is represented as a circuit that returns one of the inputs and discards the other.

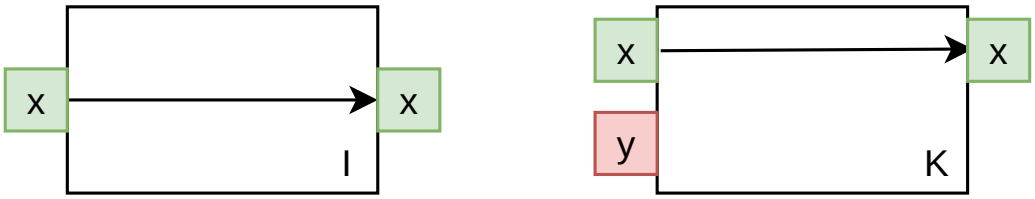


Figure 2.11: **I** and **K** combinators.

The various types of functional calculi can all be used within the same compiler, to make use of the combined power their individual abstractions. Just as a higher-level functional programming language can be compiled down to the lambda calculus, so too, can the lambda calculus be compiled down to the *SKI combinator calculus* through *abstraction elimination*. For example, the lambda abstraction is representable as the **I** combinator: $\lambda x.x = \mathbf{I}$. Better still, we can interpret **SKI** combinators in terms of other **SKI** combinators. The **I** combinator can be represented as **SKK**. Interpretations such as this can be used to simplify the representation of an application. In an optimizing compiler this can be very useful indeed. Any transformations that follow can simply deal with **S** and **K** terms given the knowledge that all **I** terms have been removed.

The computational models exposed by the lambda and SKI calculi are indeed useful as part of an optimizing compiler but come with issues of their own. One problem is that they represent computation at a far too granular level of abstraction. Another issue is that although applications represented in these calculi can be readily composed, not all compositions are necessarily valid. Both of these issues can be side-stepped by letting the programmer work at a slightly higher level of abstraction, using computational abstractions such as *monads* that can be compiled down to the lambda/SKI calculus.

A **monad** is a computational structure that can be used to *sequence computation* and thus simulate *stateful computation*. There are two fundamental operations that monads must implement. **Return** an operation that *embedded a value* into the monadic context **Bind** an operation that receives as input: **a value embedded** within a monadic context as well as a function from *values* to *values embedded in the same context*. Bind *sequences* the two inputs, by applying the second input to the *unpacked* version of the first.

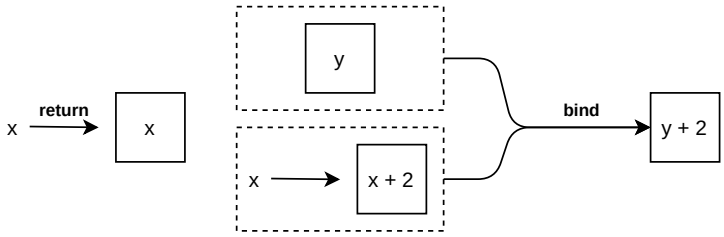


Figure 2.12: The Haskell Monad interface.

The *context* into which the **return** operation embeds a value can be used to track numerous *effects* that would have otherwise been implicit in an *imperative programming language*. Examples of such effects might be: Computing with *exceptions*; Modelling computation that can *fail to terminate*; Input/Output operations such as printing or accepting user input; Computational nondeterminism. Each of the effects we have enumerated corresponds to a particular monad. Computation that *might fail* can be represented using to the *Maybe monad*. In Haskell, this is modelled through the following data-type:

```
data Maybe a = Just a | Nothing

returnMaybe :: a -> Maybe a
returnMaybe x = Just x

bindMaybe :: Maybe a -> ( a -> Maybe b) -> Maybe b
bindMaybe (Just x) foo  = foo  x
bindMaybe Nothing = Nothing
```

Listing 2.6: The Haskell Maybe Monad.

Notice that the *type annotations* clearly denote which terms in the language correspond to *pure values*, free of any side-effects, as well as which terms might *fail*. The former are represented by simple type variables such as *a* and the latter are denoted by types such as *Maybe a*. The compiler can make use of these type annotations to *track effects* as they appear throughout the source-code. The *type-checking phase* in a compiler can make use of this information to reject applications that might fail silently.

2.3.4 Bridging models

We have now seen how the computational models of imperative and functional programming languages differ. In subsection 2.3.3 we have also seen that there exists a bridge from a functional model of computation to an imperative notion of evaluation known as the *Continuation Passing Style* transformation. Building a bridge that leads the other way, from imperative models to the pure land of functional programming, in our particular context of application optimization at least, is more complicated but not impossible. Doing so requires a change in perspective and a limitation of scope. The shift in perspective requires that instead of modelling the entire set of application semantics, we look only at a certain set of properties related to the *performance* and expected hardware *resource use* of the application. Such properties collectively define a *cost-performance model* which can be understood to be a partial evaluation of the formal language used to encode the specification into a domain of cost-performance estimates. The cost-performance estimates produced by such a model will then be related, via denotational semantics, to the optimizing transformations that can be applied by a compiler written in a functional programming language, rather than the semantics of the application being compiled. The limitation in scope we mentioned is also related to this change in perspective. If we are to establish a sound relationship between the compiler's optimizing transformations and the cost-performance estimates produced by our model, then we must limit ourselves to a certain class of applications which satisfies the following requirements.

- The application being optimized may be represented in any imperative programming language and contain *side-effects* or *non-deterministic computation*, however, such behaviour must be limited to the smallest possible scope, typically that of a function call.

- The cost-performance model must be as accurate as possible. It must neither *over-estimate* the performance of a give application nor *under-estimate* the quantity of hardware resources needed to implement it. If either of these adverse situations arise, an optimizing compiler may not be able to correctly pick the best transformation sequence for a given application.

At first glance these requirements may appear to be quite restrictive and this is true in a general sense. In our particular context of optimizing scientific applications for FPGAs however, these constraints can be shown to be satisfied. The OpenCL programming framework that is typically used to accelerate scientific computations for GPUs and FPGAs, for example, limits the scope of non-deterministic computation to so-called *kernel functions*, as we will see in subsection 3.1.1. The requirement for an accurate cost-performance model is satisfied by the TyTra compiler framework, which we present next in section 2.4. The exact relationship between the cost-performance model, optimizing transformations and the application itself will be define by the Design Space Exploration strategy we contribute to the TyTra compiler in chapter 4 using the basic category theoretical notions presented in section 2.5 that concludes this chapter.

2.4 The TyTra Compiler Framework

Within this section we will discuss some of the more important features and structural details belonging to the TyTra compiler and the intermediate representations it supports: TyTra CL and TyTra IR.The TyTra Compiler Framework [VNU19] was developed with the explicit purpose of producing an efficient and *correct-by-construction optimized* program variant from an initial specification of a streaming data-flow applications. The compiler framework will ultimately target Heterogeneous HPC platforms and has the initial goal of optimizing the execution of scientific applications on FPGAs. In this sub-section we will present background information on the following TyTra compiler aspects, which all constitute prior work, except where otherwise specified.

In subsection 2.4.1 we show the TyTra compiler workflow, the nature of the optimizing transformations the compiler must consider, as well as a high-level view of the cost-performance model used to explore the space of optimizing transformations. In subsection 2.4.2 we give an overview of the TyTra Coordination Language (CL) which represents the *data-flow structure* of the applications under optimization. The representation we use is slightly different from that in previous work [VNU19] as our more efficient approach to DSE elides the need to represent concrete optimization decisions within the TyTra CL itself. In subsection 2.4.3 we present the TyTra Intermediate Representation (IR) which encodes the low-level implementation details needed to generate an executable version of the application, as well as the execution performance and hardware resource use estimates needed for DSE.

2.4.1 TyTra Compiler workflow

The TyTra Compiler Framework enables FPGA Design Space Exploration by generating program variants using type transformations [NV15b]. This workflow assumes a High-level Language (HLL) implementation of the application to be compiled as its input. From this, the TyTra Front-End Compiler, detailed in this section, below, produces the TyTra CL and TyTra IR representations. From the latter representation, the cost performance model, detailed in section 2.4.1 produces an accurate cost/performance estimate.

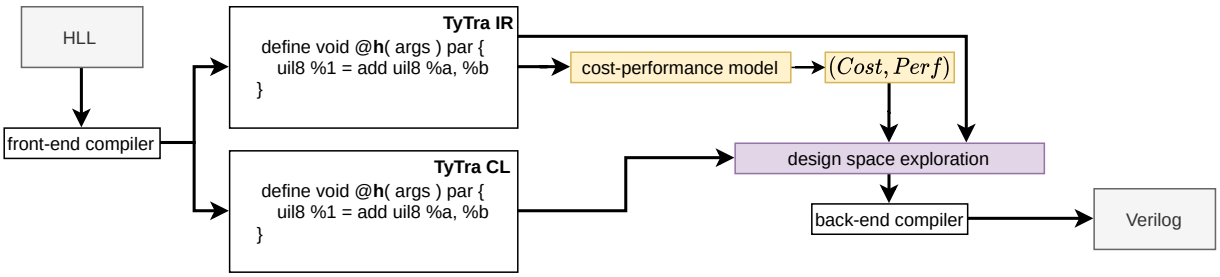


Figure 2.13: High-level view of the TyTra Compiler Workflow.

Whilst a preliminary description of a naïve but complete automated design space exploration approach for the TyTra Compiler is given in [NV15b], an *automated method of generating program variants* was left as future work. We claim the following contribution to the TyTra Compiler frameworks: a method of reducing the search space that must be traversed to yield the optimized program variant. Through this reduction of the search space size we produce a more *efficient* compiler, meaning one that requires a much shorter optimization cycle. A more efficient optimizing compiler is also a more *effective* as it can be used to optimize larger applications.

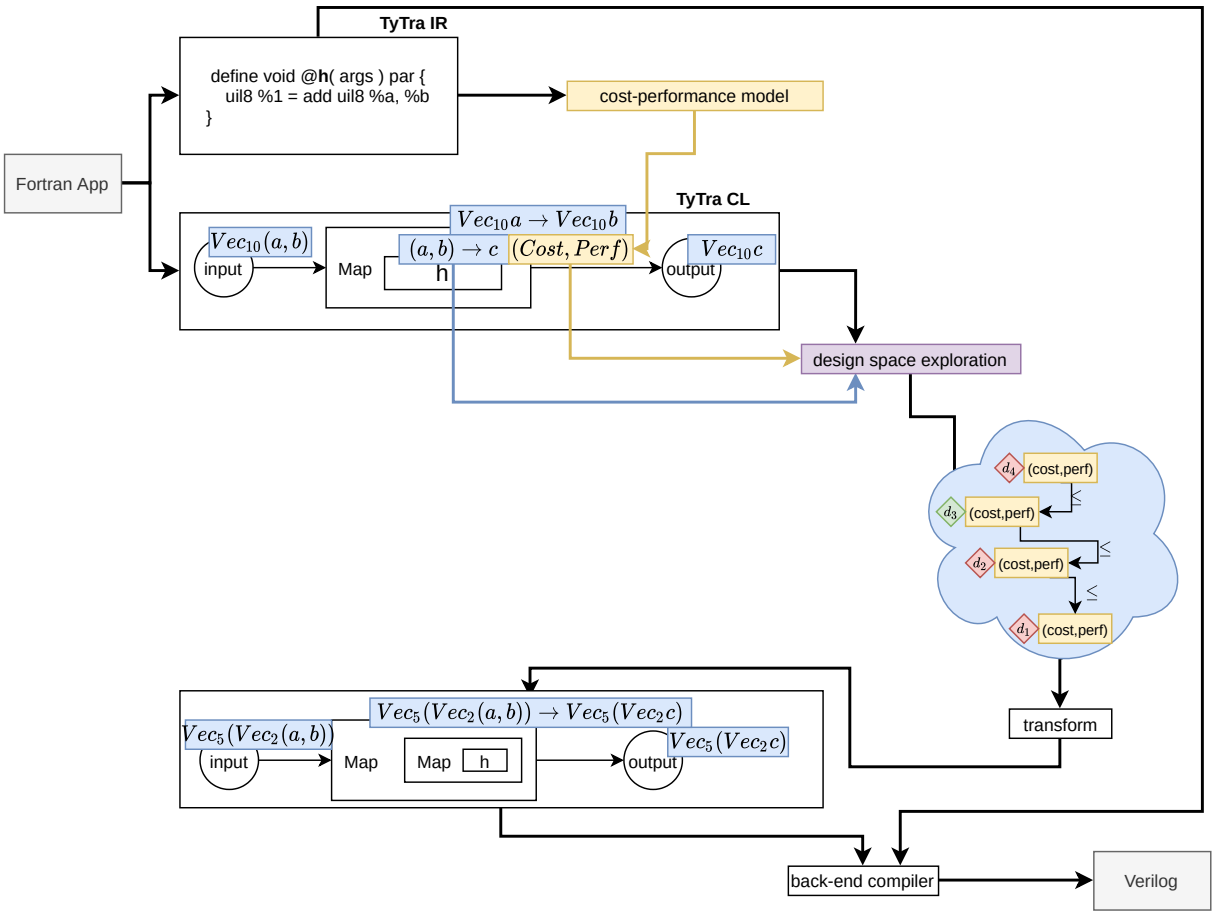


Figure 2.14: Detailed view of the TyTra Compiler Workflow.

We propose to automate the process of obtaining performance portability within the TyTra correct-by-construction compiler by altering the workflow shown in Figure 2.13 in a way that reduces the overall search-space whilst retaining the globally optimum solution. This would enable scientists to effortlessly re-target legacy applications towards modern hardware, and in particular FPGAs, thus bringing further benefits in terms of performance and power-efficiency.

Front-end compiler

The TyTra *Front-End compiler* is responsible for identifying and extracting patterns of parallel computation from an application's source-code representation [VD17]. It works by separating *parallel computation constructs* from all other *implementation details*. This allows the compiler to effectively reason about parallelism, while at the same time lowering the complexity of implementing the compiler. Other High-Level Synthesis tools rely on the user to supply *pragma directives* that identify computations amenable to optimization, as we show in section 3.1. In contrast, the TyTra Front-End compiler performs data-flow analysis and conversion to identify such computations. The analysis and data-flow conversion stages generate two representations of the application:

1. The *TyTra CL* representation (see subsection 2.4.2) which is a functional coordination language. Syntactically it resembles the Haskell programming language [VNU19]. It features: *let-statements* and higher-order functions such as *map* and *fold*, stream representation changing operations: *zipt* and *unzipt*; as well as *stencil computation* operations. A *dependent type system* allows the compiler to reason about *size-indexed types*.

2. The TyTra IR representation. In contrast with the TyTra CL, the TyTra IR is defined at a lower-level of abstraction. It contains the implementation details pertaining to the *function bodies*, or as we call them opaque functions, in addition to the computational structure expressed in TyTra CL that only represents how opaque functions are glued together using higher-order functions such as *map* and *fold* , as shown in subsection 2.4.2. The language itself syntactically resembles LLVM IR and it has *static-single-assignment semantics* as explained in subsection 2.4.3. The lower-abstraction level allows us to interpret TyTra IR through an accurate cost-performance model in a straight-forward manner, not hugely more complicated than simple instruction counting.

The patterns of parallel computation that the TyTra Front-End compiler extracts have a good correspondence to *parallel algorithmic skeletons*, which we describe in greater detail in section 3.2.1. Briefly speaking, parallel algorithmic skeletons represent patterns of computation that may be mapped to one of many optimized yet ad-hoc, parallel implementations. Through the present work we contribute a formal definition of category theoretical semantics for the TyTra CL, in section 2.5. Doing so requires that we alter the structure of the coordination language as presented in [VNU19]. The restructured CL, as it appears in subsection 2.4.2 and its categorical semantics enable us to reason about notions such as expressed parallelism *intensionally* rather than *extensionally*, a distinction made more clear in section 2.4.4.

Back-end compiler

The TyTra *Back-End compiler* implements functionality that is dependent on the *operational* details of the target hardware architecture. This includes an accurate and fast *cost-performance model* which can estimate the latency, throughput and hardware resource use metrics [NV17]. It also implements a prototype *Verilog code-generator* [NV15b]. Both classes of features accept TyTra IR representations as input.

Cost-Performance Model

The TyTra Cost-Performance model is essential to deriving an efficient DSE strategy. Having access to a quick and accurate cost-performance model means we can compare and choose program variants without going through a lengthy hardware synthesis cycle. The model makes use of the TyTra IR which has Static-Single-Assignment semantics [NV15a], meaning that it expresses computation as a *data-flow* structure with immutable variables. It explicitly encodes the functional properties of each *expression* that make up what is called an *opaque function*. In prior work, the cost-performance model is used in the first and last stages of DSE. In the *specification stage*, the performance estimate is used to generate the overall search space. In the *selection* phase, the hardware resource estimates are used to eliminate candidate solutions that could not be possibly implemented on the target hardware device, whilst both the performance and cost estimates are used to rank competing solutions.

Each *instruction* within the TyTra IR has well-defined properties which can be determined by the *cost-performance model*. On the performance estimate side this includes such measures as the *number of clock-cycles* required to execute the instruction. An example is shown below, in Figure 2.15, where the addition and multiplication instructions both take 4 clock cycles.

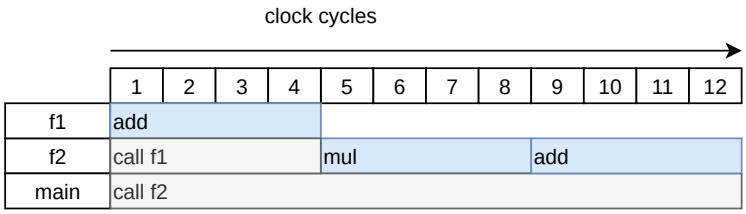


Figure 2.15: Performance measured in clock-cycles.

Under the hood, the cost-performance model is *actually more granular*. It accounts for the number of clock-cycles required by an instruction *to begin processing* and denotes this as the *delay*. The number of clock-cycles required to produce *one item of output, for each item of input received* is called *latency*. Each instruction may be *internally pipelined* meaning that there may be hidden internal buffers and functional units that hide latency.

To account for this behaviour, the cost-performance model exposes further parameters, including the *firings per output* and aggregates of such properties, further details of which can be found in [VNU19].

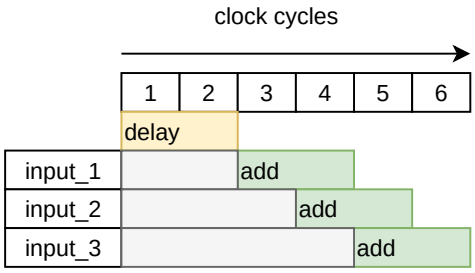


Figure 2.16: Simplified presentation of the performance model.

On the *cost* side, every instruction is assigned numerical values representing how much of each type of hardware resource is required to implement the instruction. This may include: the number of Digital Signal Processing (DSP) slices, the number or embedded memory blocks (Bram), the amount of on-device memory required (Dram), the number of logical slices or *Lookup-Tables* (LUTs).

Code Generation

Modern optimizing compilers such as *Graal* [DSW+13], *Numba* [LPS15], *LLVM* [LA04], the *Glasgow Haskell Compiler* [JHH+93] split up the compilation process into a number of phases, the last of which is code-generation. By treating the issues of *optimization* and *code-generation* separately, compiler implementations can be made more efficient and maintainable. All-though code-generation is conceptually the last stage in a compiler, it is often the case that it is followed up by further, lower-level translation processes to either machine code or some other intermediate representation.

The TyTra Code Generator produces an HDL representation of the application which requires further processing using vendor provided tools. Because HDLs are intended to be human-readable representations, one may view TyTra as a *source-to-source compiler* from Fortran to Verilog. Neither internal intermediate representation, TyTra CL nor TyTra IR, can be directly executed. The same is true of the Verilog representation generated by TyTra which requires a further *hardware synthesis* step that generates the *bitstream* representation that is ultimately uploaded to the FPGA [NV15a]. The Verilog representation of the application that is code-generated expresses a *circuit view* of computation. FPGAs are widely seen as a form of a *configurable circuit* that can be made to behave according to a specification [GS08] [GSAK00] [Bol08] [CH05]. Circuits must be fully defined and inter-connected in order to work, which means this model of computation demands *strict evaluation semantics*. Because the TyTra Back-End compiler converts the application into a circuit [NV15a], we can interpret this saying that TyTra assigns strict evaluation semantics to the application during code-generation.

2.4.2 TyTra Coordination Language

Within this section we introduce the TyTra CL by showing the syntax and semantics of the language and briefly discussing the type-system. We will also cover the manner in which type-level transformations induce a set of corresponding optimizing transformations at the term-level.

Structure

The TyTra Coordination language is a *general purpose*, *dependently typed*, and *functional* coordination language. The use of the *Coordination Language* term is to highlight its role and contrast it to that of a general purpose programming language: The CL representation of an application encodes the manner in which the computation is structured. It does not provide the implementation details for the opaque functions but simply specifies the data-types used and the data-flow structure. The TyTra CL representation of an application can be roughly divided in two sections: A header that contains the *data-type* annotations, and a body consisting of *term-level assignments*, *both of which are depicted in Listing 2.7*. Type annotations are a *mapping* from variable *names* to *type-level expressions*. Term-level assignments are mappings from term *names* to *term-level expressions*.

```
pow :: Float -> Float -> Float
cos :: Float -> Float
sin :: Float -> Float

ker0 :: (Float, Float) -> Float
ker1 :: Float -> (Float, Float)
ker2 :: (Float, Float) -> Float

vin1 :: Vec s Float
vin2 :: Vec s Float

pout = map ker2 (map ker1 (map ker0 (zipt (vin1, vin2 ))))
```

Listing 2.7: TyTra CL representation example.

Type annotations use the *::* operator, whereas term-level assignments are denoted by the = operator. In both cases, the term or type name is the left side of the operator, whilst the definition is on the right. Type-level expressions consist of either *atomic types*, such as *Int* in the example above, or a more complicated type, built using one or more TyTra *type constructors*. The *Vec s Int* type in Listing 2.7 is constructed using the *Vec* type constructor, which is parametrized by the *Int* type and indexed by a size *s*.

TyTra CL Terms

Term expressions are either the occurrence of an input variable; the application of an *opaque* function to previously defined term expression; or a nested application of *higher-order functions* that have been adequately parametrized. The following grammar defines lawful term construction in the TyTra CL.

⟨*action*⟩ ::= 'Id'
| 'Opaque' ⟨*name*⟩
| 'Map' ⟨*action*⟩
| 'Fold' ⟨*action*⟩
| 'Elt' ⟨*index*⟩
| 'Stencil' [⟨*index*⟩]
| 'Zipt'
| 'Unzipt'

⟨*name*⟩ ::= ⟨*string*⟩

⟨*index*⟩ ::= 0 .. N

⟨*expr*⟩ ::= 'Var' ⟨*name*⟩
| 'App' ⟨*action*⟩ ⟨*expr*⟩
| 'Tup' [⟨*expr*⟩]

Figure 2.17: TyTra CL term-level expression grammar.

We can observe that the CL terms are split into two categories: *actions* and *expressions*. There are three *data-constructors* for expressions.

- **Variable** constructors wrap a *name*, signifying a *stream* of data.
- **Tuple** constructors wrap multiple expressions into a single expression that support *projection* (indexing) operations.
- **Application** constructors apply *actions* to *expressions*.

The last of these constructors bridges the action and expression *kinds* by equating fully applied or *saturated actions* to expressions.

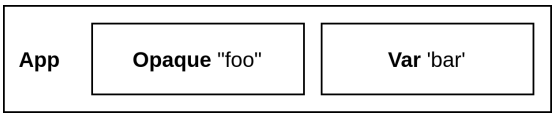


Figure 2.18: TyTra CL expressions and actions

As we rely heavily on the isomorphism between *data* and *computation*, we will sometimes use a different graphical presentation, in which we denote the **App** constructor as an arrow.

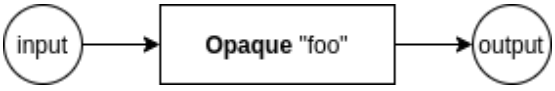


Figure 2.19: Alternative graphical representation for TyTra CL expressions and actions.

Actions correspond to either user-defined *opaque functions* or one of the built-in higher-order operations defined for the TyTra CL.

- **Opaque functions** are referred to by name. The compiler keeps track of the type of these functions in the type context. Implementation details are not represented in the Coordination Language at all. Instead, the compiler makes use of the simple understanding that opaque functions are deterministic and *side-effect free*, a term we explain in subsection 3.1.2.

- **Id** represents the polymorphic *identity* function that yields any input expression provided.

- **Map** actions, are higher-order functions. A *map* action may be parametrized by an *opaque* function or any other action. When given an opaque function $f : A \rightarrow B$ as input, $mapf$ denotes a function that takes any sized vector of type A and produces a vector of the same size parametrized by B. In other words $mapf : Vec_sA \rightarrow Vec_sB$.

- **Fold** actions are somewhat similar to *maps* in that they too are higher-order functions. The crucial differences being that: A *fold* action reduces a vector of values to a scalar. A fold actions are parametrized by binary *Function Types*. For a given function $g : A \rightarrow B \rightarrow A$ the function obtained by applying fold has the type $foldg : A \rightarrow Vec_sB \rightarrow A$. The first argument is the initial state of the *accumulator*. The second is the vector of values to be folded.

- **Elt** is a short name for *element*. In a way it is a *highly* specialized version of a *fold* action. The parameter specifies an index into a *Vector Type* to be returned as the output. The type of $elti$ is thus $elti : Vec_sA \rightarrow A$ with the understanding that $i \leq s$.

- **Stencil** takes as an argument a list of *relative* indices. When applied to a *Vector Type* it produces a *vector of tuples*, each tuple being composed of those elements in the vector found at the relatively offset positions as specified by the indices.

- **ZipT** and **UnzipT** actions serve to reshape a *tuple of vectors* into a *vector of tuples* and vice-versa.

Term transformations in the TyTra compiler are derived from the type-level transformations we discussed in the previous sub-section. More specifically they are derived from the class of transformations having the kind $* \mapsto *$, which excludes the atomic type constructors, as well as the size constructors. The TyTra compiler derives term-level transformations from type-level transformations that work on the structured, not the atomic, types in the language. Coming back to the examples of type transformations we provided in the previous section, we have another look at *split* and *merge*, this time explicitly highlighting that these functions on types are parametrically polymorphic.

Types and Transformations

Alongside the computational structure, the CL contains type definitions for the opaque functions, program input parameters and the program's output. Types pertaining to intermediary data structures, storing partial results, are generally inferred by the type-system. What the CL does represent, which concerns not only *opaque functions* but *input expressions* and *higher-order functions* are certain properties that can be expressed as *data-types*. The TyTra CL resembles a subset of the *Haskell* programming language, with the major distinction that terms are assigned *dependent-types*.

We must first distinguish Types by their *Kinds*. A *Kind* is another, less ambiguous name for the *Type of a Type*. Kinds are necessary to ensure that *Type Constructors*, which can be alternatively viewed as *functions on types* are themselves *well typed*. In other words the *kind system* ensures we only pass the right kind of type where it is expected.

The coordination language features the following kinds: *Names*, *Sizes* and *Types*. *Names* are simply *Strings* used as a proxy for unique identifiers. *Sizes* are natural numbers defined in the usual manner as Peano numbers with a constant *Zero* literal and the *Successor* function.

$$\frac{}{\text{Zero} : \mathbb{N}}\ \textbf{Zero} \qquad \frac{\text{p} : \mathbb{N}}{\text{Succ p} : \mathbb{N}}\ \textbf{Succ} \qquad \frac{\text{n} : \mathbb{N}}{\text{Size n} : \text{Size}}\ \textbf{Size}$$

Figure 2.20: TyTra CL Size Constructors.

Types are denoted with the shorthand symbol $*$. *Atomic Types* are *nominal* meaning that the compiler distinguishes atomic types by name and not by structure, in the same manner as it distinguishes opaque functions.

$$\frac{\text{name} : \text{Name}}{\text{Type name} : *}\ \textbf{Atomic Type}$$

Figure 2.21: The TyTra CL Atomic Type Constructor

The non-functional properties of to atomic types, such as how quickly data can be accessed and the amount of memory required to store a given value can only be assessed indirectly, through the *cost-performance model* the choice of *representation* for data can be highly platform dependent. Given that our ultimate goal is to achieve true performance portability, it stands to reason that platform details should be encoded in the cost-performance model, which is obviously already tied to the underlying hardware. Representing types *nominally* allows the compiler to remain flexible with regard to what constitutes the set of *primitive* values.

At a more fundamental level, the compiler can only be concerned with the *structure of computation*. This equates to viewing program transformations as *parametrically polymorphic functions* which in turn implies a quantified view of type parameters.

$$\frac{\text{name} : \text{Name}}{\text{Type name} : *}\ \textbf{Atomic Type}$$

$$\frac{A : * \qquad B : *}{(A, B) : *}\ \textbf{Tuple} \qquad \frac{A : * \qquad \ldots \qquad Z : *}{\text{Tup A} \ldots \text{Z} : *}\ \textbf{\textit{k-ary} Tuple}$$

$$\frac{s : \text{Size} \qquad T : *}{Vec_s\ T : *}\ \textbf{Vector} \qquad \frac{A : * \qquad B : *}{A \to B : *}\ \textbf{Function}$$

Figure 2.22: TyTra CL Structural Type Constructors.

Tuple types are a simple example of a compound data-type which appears in most programming languages. Values having a *Tuple Type* are sometimes referred to as *pairs* or *products*. The TyTra CL defines *k-ary* tuples, that is to say tuples with *k* components. Without loss of generality we may simply define a binary Tuple Type constructors. We recover full *k-ary* tuples by defining an *associator* in Figure 4.15. For now, it suffices to point out that Binary Tuples are denoted by separating their components by a comma and encapsulating within a set of parentheses.

Vector types can be thought of as homogenous *k-ary* tuples. A *Vector Type* of size *k* contains *exactly k* elements of the same type. More interestingly, *Vector Types* are indexed by their size. *Indexed Type Constructors* define an entire *family* of Types. Notice that as the size is of kind *Size* the *Vector Type Constructor* is effectively an unary type-constructor, the size being an index and not a *type parameter*.

Function types are associated with the *opaque functions* referenced within the coordination language. They are constructed with the binary *Function Type Constructor* denoted by the usual infix arrow.

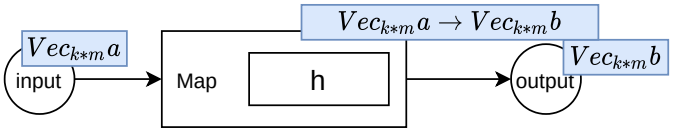


Figure 2.23: Graphical depiction of a TyTra CL expression and types.

We depict the *mapping* of an opaque function *h* over some input of type $Vec_{k+m}\,a$ in Figure 2.23. The blue boxes in the upper-right corner of each term represent the types associated with those terms. The type of map operation is determined from that of the opaque function and the input as shown by the type derivation rules in subsection 4.1.3.

Type transformations are functions that act upon types. Within the TyTra compiler we are primarily concerned with type transformations that witness isomorphisms between types. That is to say they *change the representation* of the type while maintaining its meaning. A caveat we must avoid is the confusion between type transformations, or *functions on types* from *function types*, meaning types of kind $* \rightarrow *$ in the TyTra CL.

- Type transformations are functions implemented within the TyTra compiler.
- *Function Types* describe functions belonging to the application being compiled.

We denote type transformations by their names, usually as single letters in uppercase such as F or G. In type derivation rules specified we also denote type transformations by a different arrow, as below:

$$\frac{\mathrm{A} : * \qquad \mathrm{F} : * \mapsto *}{\mathrm{F\ A} : *}\ \textbf{Type Transformation}$$

Figure 2.24: An abstract type transformation.

An important class of type transformations is that of the *Type Constructors* we presented in the previous sub-sections. The *Atomic Type* constructor is a function that takes a type of kind *Name* and it produces a type of kind *Type*, in other words: $Name \mapsto *$. More complicated types that are inductively defined, such as nested vector types are likewise constructed using functions on types, or type transformations. The only difference is that we can think of the output type as resulting from an iterative application of type constructors, or equivalently, a composition of type transformations that is applied to the input.

$$\frac{\dfrac{\mathrm{A} : * \qquad Vec_k : * \mapsto *}{Vec_k\ \mathrm{A} : *} \qquad Vec_m : * \mapsto *}{Vec_k\ Vec_m\ \mathrm{A} : *}$$

Figure 2.25: Nested type constructors can be simply seen as a single constructor.

This last example shows exactly what is meant by saying that type transformations witness type-level isomorphisms: If we are to split the type derivation above according to the bottom-most horizontal line, we can state that the type derivation on top (the repeated application of the vector type constructor) and the bottom one are isomorphic. The transformations that witness the isomorphism on vector types are:

- **Split:** $Vec_{m \times k}\ b \rightarrow Vec_k Vec_m\ b$
- **Merge:** $Vec_k Vec_m\ b \rightarrow Vec_{k \times m}\ b$

The fact that *Split* and *Merge* witness the isomorphism between a flat vector of size $k \times m$ and a $k-$sized vector of m-sized vectors is shown graphically in Figure 2.26 below.

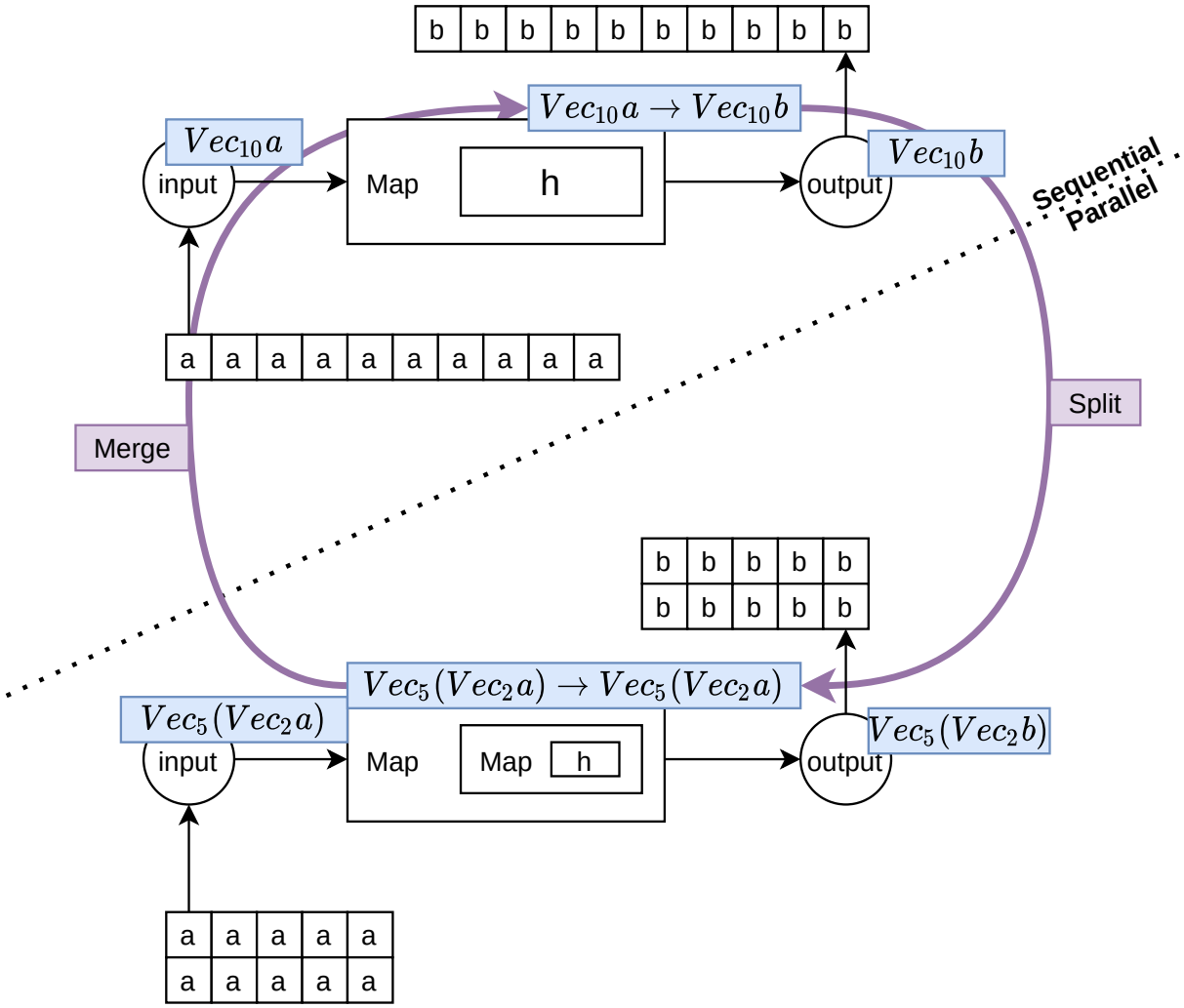


Figure 2.26: The effect of Vector type Split and Merge operations on terms.

In other words, the type-level equivalence gives rise to a *term-level program transformation* that can be used to trade additional resources for performance. We will primarily denote the *term-level* transformations through Haskell type signatures as below.

```
split :: forall b. (k :: Size, m :: Size) -> Vec (m * k) b -> Vec k (Vec m b)
merge :: forall b. ( k :: Size, m :: Size) -> Vec k (Vec m b) -> Vec (m * k) b
split ∘ merge = merge ∘ split = id :: forall a. a -> a
```

Figure 2.27: Type-level merge/split rules. Their composition yields the identity transformation.

2.4.3 TyTra Intermediate Representation

In this section we will briefly describe the TyTra Intermediate Representation (TyTra IR) as introduced by its authors [NV15a]. We claim *no contribution to the development of the TyTra IR in this work* as the *design-space exploration strategy* we contribute simply relies on the *cost-performance estimates* derived from the TyTra IR.

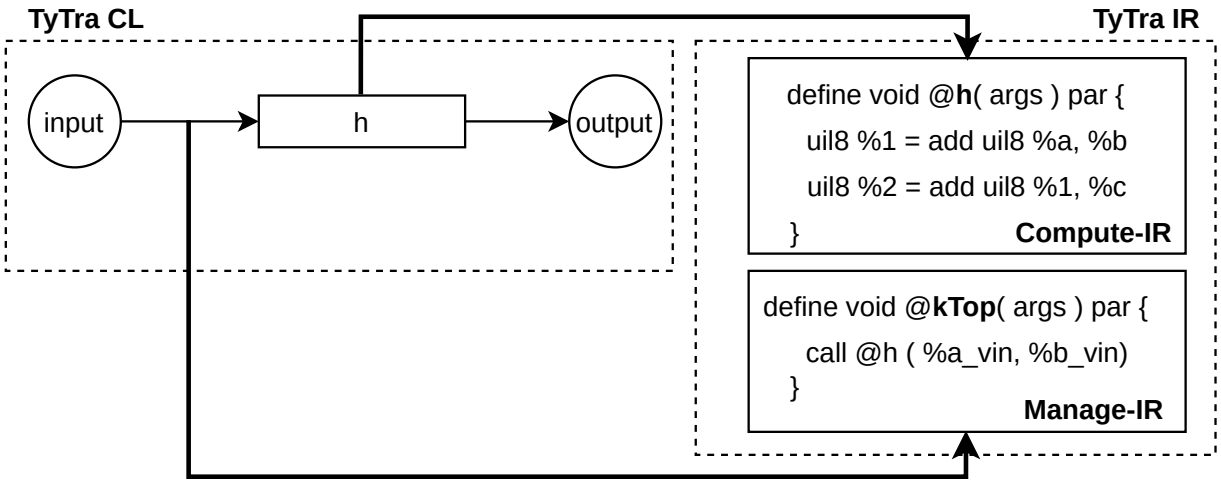


Figure 2.28: TyTra IR, its sub-languages and the relationship to TyTra CL

The TyTra *Coordination Language* represents the data-flow structure of applications. This structural information can be used by the optimizing compiler to determine an overall set of *semantics-preserving* program transformations. A *sub-set* of these optimizing transformations may yield better-performing *program variants*, however a compiler would have no way of telling which of the transformed variants *performs better* nor which *sub-set* satisfies the hardware resource constraints imposed by the target device. The TyTra IR, on the other hand, represents the low-level information missing from the TyTra CL. The concrete syntax of the TyTra IR resembles the intermediate language used by the LLVM compiler, called LLVM IR. Semantically, the TyTra IR is a statically typed, *static single assignment (SSA)*, representation which can be further divided into two *sub-languages* [NV15a].

- **Manage-IR**, the subset of the language that roughly corresponds to the structural information also expressed within the TyTra CL representation.
- **Compute-IR**, the remaining part of the TyTra IR describing the computational operations that occur within an *opaque function*, as well as the dataflow streams between these operations.

Operations within the *Compute-IR* subset are expressed at a very fine level of granularity which enables the cost-performance model to derive accurate performance and hardware resource use estimates. The fine granularity also enables the back-end code-generator which *emits* a Verilog representation of the application.

2.4.4 TyTra Semantics

The TyTra compiler framework proposed a *correct-by-construction* methodology for program optimization [NV15b]. This is an inductive approach which grantees that the compiler's output program variant can be attributed *the same functional semantics* as those attributed the initial/input program variant.

- The initial program variant, represented in the TyTra CL, is assumed to be a correct specification of the the application to be optimised and compiled. We check this assumption for every input application, through the use of a *type-checking and inference mechanism* we have developed as part of this work, in support of our previous publication [VNU19].
- Every *term-level* optimizing transformation applied by the TyTra compiler is derived from a *type-level* equality [NV15b], meaning that, for every *correct input application* the compiler produces a *correct, transformed, application*.

Under the **operational** view of semantics, two applications or operations, are indistinguishable if they are *observationally equivalent* meaning that for all possible inputs, they produce the same outputs [Weg72]. Under the **denotational view of semantics**, two applications are equivalent if they are *denoted* or *represented* by the same terms in the *domain of interpretation* [SS71]. In the case of pure dataflow programming languages, *operational semantics* **are entirely equivalent**, in a general sense, to *denotational semantics* [Fau82].

By using *denotational semantics* we can check that two term-level constructs, operations or applications are indeed identical by checking that they are denoted by the same *type*. This however requires the assumption of *parametricity*. A function is said to be *parametrically polymorphic* if it acts upon a data type generically, by inspecting the data-types structure and not its contents. Cousout et al define the *semantics of programs* as "the set of all possible behaviours of those programs when executed for all possible input data" and *abstract interpretation* as "a method for designing approximate semantics of programs which can be used to gather information about programs in order to provide sound answers to questions about their run-time behaviours" [CC92].

Intensional and Extensional Semantics

While the TyTra Coordination Language is given streaming semantics that map very well to both the application domain and targeted hardware platforms, this interpretation alone is not enough to automatically derive efficient transformation routines. We further require a view into those properties that denote the performance of a transformed program, as well as some measure of computational resources such a variant may consume.

The **TyTra Cost-Performance model** gives us this particular view of computation. It takes into account not only the *coordination-language* but also the lower level intermediate representation of an application in *TyTra IR* to produce values that represent the performance, roughly speaking the throughput and latency measures, of a program variant, as well as measures of hardware resource use. Having access to a measurement apparatus is not the whole story. We must relate the semantics of a program variant to its cost-performance properties in a sound and reliable way. Ideally, we would link both a program *meaning*, and its *cost-performance properties* in a unified view of computation.

This issue is not only known but also well explored. As Brookes and Geva point out: *"Extensional semantics are typically appropriate for proving properties such as partial correctness, but an intensional semantics at a lower abstraction level is required in order to reason about computation strategy and thereby support reasoning about intensional aspects of behaviour such as order of evaluation and efficiency"* [BG91]. Going further, Brookes and Geva introduce a notion of intensional semantics by modelling the meaning of an application, in the extensional sense, to be a function from and to data values, and relating to it the intensional notion of computation as a function from computations to values [BG91]. In spirit, this is what the TyTra cost-performance also accomplishes: it attaches a measure of performance and resource use (the value) to the computation.

Streaming Semantics

Streaming computation can be described by the *Iterator pattern* which according to Gibbons and Oliveira implies an "interface for element-by-element access to a collection of independent values" [GdSO09] and "has two simultaneous aspects: *mapping* and *accumulating*". The TyTra Coordination Language implements the following element-by-element access constructs. *Mapping* which refers to the act of applying a given transformation to every item of data that we iterate over. Every item of input is transformed, in precisely the same way by iterating over a sequence of inputs, and applying the same function to all values. As each item is processed independently, it is easy to see that a genuine *mapping* is a parallel operation. *Accumulations* on the other hand are not as approachable. Multiple items of input are required to produce each item of output, meaning that there is some dependency between these values. Certain *accumulations*, also known as *reductions*, can be processed in parallel if the underlying operation that relates any such mutually dependent values is *associative*.

The structured assembly of *maps* and *accumulations* allows for the construction of complex applications and is a hallmark trait of functional programming languages which can describe such structures through higher-order functions [Hug89]. The TyTra compiler implements an extended version of this formalism that features *zip unzip* and *stencil* operations.

Zip and Unzip operations witness the relationship between two streams of values, and a single stream of *value pairs*. *Stencil Operations* group values selected from a stream, producing a stream of tuples or pairs. Stencils thus allow computations that depend on multiple values to be nonetheless expressed as pure *mapping* operations. The streaming semantics of the coordination language also reflect in the type system. Sequences of values to be processed are represented by *dependently typed vector types*. *Vector types* are parametrized by other *types* and are *indexed* by their size. A *size index* provides static access to a vector's dimension, which in turn enables a number of key optimizations. The size index serves as a convenient proxy by which an exhaustive *search space* of *term level transformations* can be defined. This search-space is generated and traversed by a *design space exploration* process to yield an optimal sequence of program transformations.

Abstract Semantics and Interpretation

To *interpret* an application usually means to *evaluate* the expression it defines or otherwise fully execute it, such that the final output value is produced. An interpretation of the term expression *1+2+3+4*, for example, yields the sum of these numbers, the value *10*. The concrete manner in which this is achieved may vary. One possible interpretation is shown in the trace of this calculation below.

```
1 + 2 + 3 + 4
>> (1 + 2 ) + 3 + 4
>> (3 + 3 ) + 4
>> (6 + 4)
>> 10
```

In contrast, *Abstract interpretation* serves to produce an abstract or partial result. Such approaches are useful in determining a certain *property* of the output value, rather than the property itself. Suppose we wanted to determine if the same term expression above yields a result that is greater or equal to the value *6*, but did not care as to what the exact result of the computation is.

```
1 + 2 + 3 + 4
>> (1 + 2 ) + 3 + 4
>> ( 3 >= 6 == False , 3 + 3 + 4 )
>> ( 6 >= 6 == True , 6 + 4 )
```

Notice that this answer can be computed without having to fully evaluate the term expression, provided that the only permitted operation is addition, and that all input values are positive. Giving an application *abstract semantics*, meaning that we define an *abstract interpretation* mechanism, thus allows certain answers to be computed more quickly than otherwise possible.

Powerful as abstract interpretation is, it is not a universal answer to application optimization. In the example just presented, abstract interpretation could be used because of the specific restrictions imposed on the construction of the expression being evaluated, which are also highly related to the nature of the question being asked. Counter-intuitively, although the question asked in the second instance is more complex, the answer can be found more quickly. In most circumstances it would seem as if the reverse situation is true. Running two interpreters sequentially such that the first produces an abstract solution, and the second yielding the final output value, would appear to be slower than simply computing the output directly.

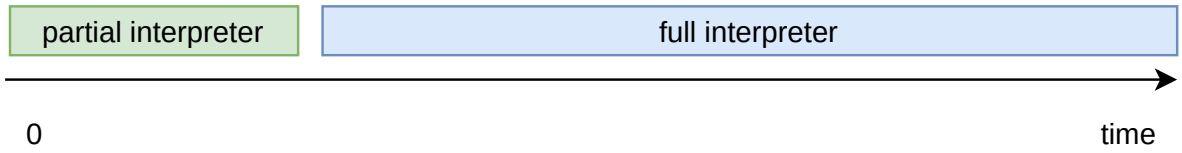


Figure 2.29: Abstract and full interpretation (sequential).

This sort of intuition is correct if what is being computed is a single, iteratively defined value. The benefits of abstract interpretation are more likely to manifest as *average runtime speed-ups* when computing collection of values, possibly related by some property, rather than a single value that depends upon the entirety of the input.

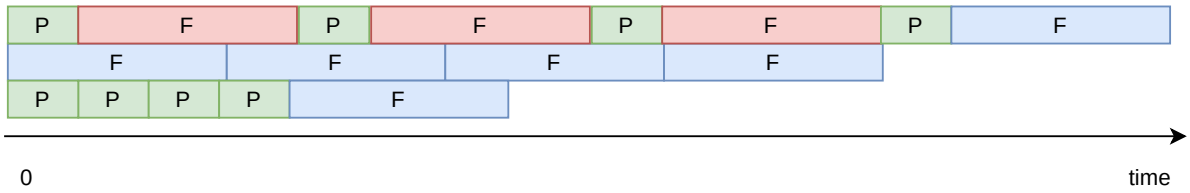


Figure 2.30: Abstract interpretation (average time).

If the partial output computed by the abstract interpreter is enough to determine weather or not the full output value would satisfy the property of interest, then we may be able save on the overall average run time of the application never executing the full interpreter when this is warranted. This is precisely the case with *Design Space Exploration* where *most* of the design space has to be thrown away, with potentially a single *design point* being the sought-after solution. Benefits stemming from abstract interpretation are counter-balanced by the need to ensure our abstractions are sound. The example we presented benefits from quicker execution only when all Integer values provided as input are *positive*. Abstract interpretation can be used to speed up computation if there exists an *abstraction relation* between the *full results* and the abstract or *partial results*.

2.5 Introduction to Category Theory

Category Theory, sometimes lovingly referred to as *abstract nonsense* [1] is a formalism introduced by Eilenberg and Mac Lane [EM45] to unify and simplify the presentation of many mathematical systems and structures. It has found numerous applications in a variety of scientific fields, amongst which Computing Science is perhaps the most representative. Type theory, and functional programming language research in general, borrow many category theoretical concepts. In this section, we present formal definitions for some of the more basic notions in category theory, on top of which we will build our primary contribution. Readers interested in acquiring a deeper understanding of category theory may wish to look towards Mac Lane's work [ML13] on category theory or Skillicorn's *Foundations of Parallel Programming* [Ski05]. The presentation given here, being specialized to our particular needs is fairly terse and may fall short of conveying the underlying beauty and expressive power of category theory.

At first glance, Category Theory may appear to be a complicated formalism that requires many years of study. In truth however, Category Theory is surprisingly uncomplicated. A very small set of primitive notions can be used, composed, and layered on top of themselves, yielding a language that is powerful enough to model even some of the more abstract parts of mathematics and other sciences. A *category* denotes a *collection of objects* on the one hand, and a *collection of morphisms* between those objects on the other. Categories are subject to a number of simple laws, to be defined shortly. The laws revolve around the central idea of *composition* and *abstraction*. Diagrams are frequently used in Category Theory, both as a vehicle for intuition as well as a means to prove certain properties. In Figure 2.31 we can see a simple category with three objects and a number of morphisms.

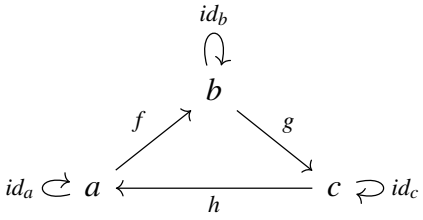


Figure 2.31: The category **C** with three objects.

The *objects* in **C**, the category shown in Figure 2.31 are denoted by the symbols a, b and c. The entire collection of objects may also be denoted $Ob(\mathbf{C})$ where $a,b,c \in Ob(\mathbf{C})$. The category in Figure 2.31 contains three such objects, denoted by labels: a,b,c. The *morphisms* in this category are likewise denoted with lower-case letters f, g and h. For every pair of objects $a,b \in Ob(\mathbf{C})$ there exists a set $Hom_{\mathbf{C}}(a,b) \in \mathbf{Set}$, called the *hom-set*, whose elements are *morphisms* from a to b.

[1] `https://en.wikipedia.org/wiki/Abstract_nonsense`

Every morphism has two associated functions, *src()* and *dst()*. These return the *source* and *destination* objects assigned to them. The source and destination of a morphism correspond to the concrete notions of a *domain* and *co-domain* as they pertain to mathematical functions. The f morphism in Figure 2.31, for example, has the following *source* and *destination* objects.

$$src(f) = a$$
$$dst(f) = b$$

If the source object of a morphism matches the destination object of another, then such morphisms can be composed. Given any three objects, a, b, c in a category $\mathbf{C}$, composition is defined as the function $\circ : Hom_{\mathbf{C}}(b,c) \times Hom_{\mathbf{C}}(a,b) \rightarrow Hom_{\mathbf{C}}(a,c)$. The composition of morphisms can be seen as a more general variety of *function composition*. In so-called *small categories* where the collection objects is precisely defined as a *set*, the morphisms or arrows between these objects are in fact functions. Given a category $\mathbf{C}$ with three objects and two suitable morphisms between them, shown in Figure 2.32 we can see that the composition of morphisms simply yields another morphism.

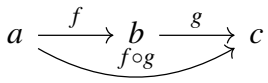


Figure 2.32: Composition or morphisms f and g in $\mathbf{C}$.

The product $\times$ operation is simply a way of saying that the composition function takes two arguments $Hom_{\mathbf{C}}(b,c)$ and $Hom_{\mathbf{C}}(a,b)$. The composition of morphisms **must** be an *associative* function, meaning that for any three morphisms f, g, h that compose in the right way, the following equality holds.

$$f \circ (g \circ h) = (f \circ g) \circ h.$$

Figure 2.33: The composition of morphisms must be associative.

For every object a in a category $\mathbf{C}$, there must *also exist* a morphism $id_a : a \rightarrow a$, called the identity morphism, which maps each such object onto itself. Given any $x \in \mathbf{C}$, we can expect that the corresponding identity morphism id_x preserves the *source* and *destination*. In other words: $src \circ id_x = src(x)$ and $dst \circ id_x = dst(x)$, as follows from the laws and definitions of *composition* and the identity morphism and gives rise to the *left* and *right unit* laws.

> **Definition:** Left and right units.
> Given a morphism $f : a \rightarrow b \in \mathbf{C}$ the identity morphism id_a is the *left unit* of *composition* while the id_b identity morphism is the *right unit*. The left and right unit's satisfy the equalities: $id_b \circ f = f$ and $f = f \circ id_a$

The identity morphisms in a category are in a one-to-one correspondence to the objects of that category and are often omitted from the graphical presentation. Alternatively a category can be depicted by simply showing the labelled arrows that depict morphisms, including the identity morphisms and denote objects by simple dots.

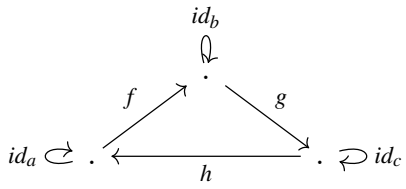


Figure 2.34: The objects of a category are identified by their identity morphisms.

A category is said to be **small** if instead of having general collections of objects/morphisms it specifically has *small set* of objects/morphisms. A *small set* in simply means a proper set. Small categories are the objects of a special category called **Set**. Morphisms in **Set** are *total functions between sets*. We are mostly interested in working with small categories as they provide enough expressive power to model the *Types* and terms of the TyTra CL as we will see in subsection 4.1.3.

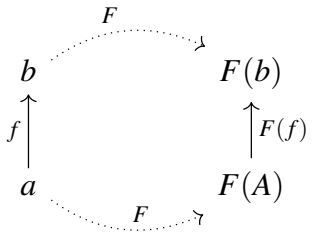


Figure 2.35: Functors: structure preserving maps between categories.

Just as **Set** is the category with sets as objects, there exists a category, called **Cat**, in which the objects are themselves categories. The morphisms in **Cat** are also called *Functors*.

> **Definition:** Functor.
> A functor F between two categories **C** and **D** is a structure-preserving map $C \Rightarrow D$, meaning that it maps the objects and morphisms in **C** to the objects and morphisms in **D** whilst preserving the *source* and *destination* of the morphisms, morphism composition and the associativity of composition.

If the two categories that a Functor maps between coincide, meaning that $F : \mathbf{C} \Rightarrow \mathbf{C}$ then it is called an *Endofunctor*. Endofunctors are particularly useful in computing science as they can be used to define *categorical data types*, as we will see in section 4.1.

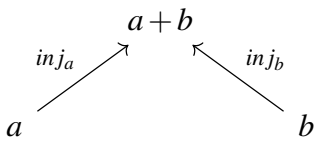
In a category **C** where the objects are *types*, an Endofunctor $F : \mathbf{C} \Rightarrow \mathbf{C}$ models a *type constructor*. The source object to F is then a *type* parameter, whilst the destination object is the newly constructed *type*. Type constructors may take more than one parameter, in which case the source object is the *product* of two other objects in **C**.

> **Definition:** Product object.
> Given a category **C** and any family of objects $x_{i\in I}$, the *product* of these objects, if it exists is denoted $\prod_{i\in I} X_i \in \mathbf{C}$. This product of $x_{i\in I}$ is a unique object (up to unique isomorphism) and comes equipped with a family of *projection* functions $p_i : \left(\prod_{i\in I} x_i\right) \longrightarrow x_i$.

Product objects can thus be used to represent the notion of a *tuple type*, also known as *product type*, from *type theory*. The projection operations are the *accessor functions* that give us access to the components of a tuple. In type theory there also exists the notion of a *sum type* which corresponds to a *sum object* in category theory. Just as with *product types*, sum types take two type parameters. Sum and product objects in category theory are *dual* constructions. To understand the relationship between dual constructions, let us first define an *opposite category*.

> **Definition:** Opposite Category.
> Given a category **C**, the opposite category C^{op} is defined as having all objects and morphisms in **C**, with the slight difference that morphisms have their direction reversed. For every morphisms $h : a \rightarrow b \in \mathbf{C}$ the corresponding morphisms is denoted $h^{-} : b \rightarrow a \in C^{op}$.

The *product object* in a category **C** corresponds to the *sum object* in C^{op}. This relationship explains why *sum objects* are also called *co-products*. An Endofunctor $F : \mathbf{C} \Rightarrow \mathbf{C}$ is called a *Covariant* Functor because it preserves the direction of morphisms. A Functor $G : \mathbf{C} \Rightarrow C^{op}$ that reverses the direction of morphisms is called *contra-variant*. Whereas product objects are defined by their projection operations, *sum objects* are defined by *injection morphisms*. Notice that the arrows denoting injections point to the sum object, whereas the projection arrows pointed away from the product object.



Whereas product objects can be used to model *tuple types* which hold multiple values, each having the respective type of the corresponding type parameter, *sum objects* correspond to a *choice of type constructor*. The resulting type can only hold one value, that of the type parameter selected by the injection that was used.

The relationship between product and sum objects, as *dual constructions* is described entirely by the definition of a contra-variant functor we briefly mentioned. A category may have both *sum* and *product* objects. If additionally the products distribute over sums, then we have a distributive category.

Definition: Distributive Category.
A *Distributive Category* is a category $\mathbf{C}$ that has both *finite products* and *finite coproducts* in which the distributive law holds: $\forall a,b,c \in \mathbf{C}.\ a \times b + a \times c \to a \times (b+c)$.

Distributivity is a very useful property, it allows us to reason about the equivalence between a list of tuples and a tuple of lists, which gives rise the *split* and *merge* program transformations in TyTra. To define lists however, we also require our category of types to be a monoidal category.

Definition: Monoidal Category.
A *monoidal* category is one that is $\mathbf{C}$ equipped with: A functor $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, called a *tensor product*; A *unit* object $1 \in \mathbf{C}$. A *natural isomorphism* $a :\ ((-)\otimes(-))\otimes(-) \xrightarrow{\simeq} (-)\otimes((-)\otimes(-))$ called an *associator*; Two *natural isomorphisms*: $\lambda :\ (1\otimes(-)) \xrightarrow{\simeq} (-)$ called the *left unitor* and $\rho :\ (-)\otimes 1 \xrightarrow{\simeq} (-)$ called the *right unitor*.

The *tensor functor* maps objects and morphisms from the *product category* of $\mathbf{C}$ with itself, to the objects and morphisms of $\mathbf{C}$. The unit object is the *neutral element* of the *tensor product* operation.In a *monoidal category* the *triangle identity* (Figure 2.36), and the *pentagon identity* (Figure 2.37) must hold.

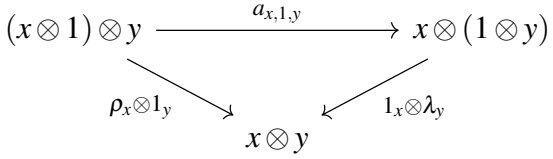


Figure 2.36: Triangle identity.

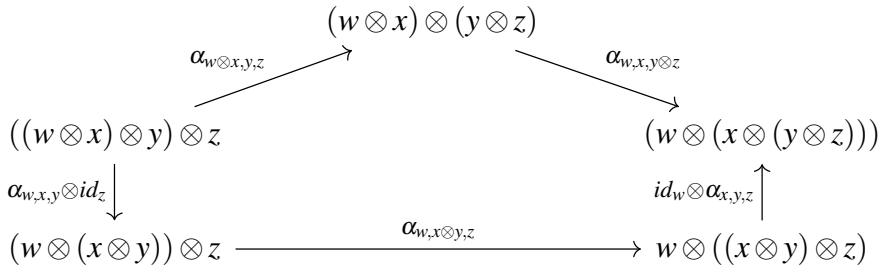


Figure 2.37: Pentagon identity.

A distinctive type of monoidal category is the *cartesian monoidal category*, often simply called a *cartesian category*.

> **Definition:** Cartesian Category.
> A *monoidal category* **C** is a *Cartesian monoidal category* if the monoidal structure is given by the *product* and it's terminal object is the *unit* of the product..

Having defined a *Cartesian Category* we can also define *Closed Categories* as:

> **Definition:** Closed Category.
> A *category* **C** is *closed* if for any pair of objects $a, b \in Obj(\mathbf{C})$ $hom(a,b)$ is an object in that category.

Closed Cartesian Categories are of great interest because they allow us to construct *exponential objects* as defined below. Exponential objects represent sets of morphisms between objects, as objects in the same category. This is required if to represent *function types* and *function application* as the objects in a category of types.

> **Definition:** Exponential Object.
> Given a *category* **C** with objects $x, y \in \mathbf{C}$ in which all binary products with y exist, an **exponential object** is an object denoted x^y equipped with an *evaluation morphism* $apply : x^y \times y \to x$ which is universal.

Given a few further restrictions, a closed category can be a *closed monoidal category*. A *closed monoidal category* **C** has for every object $x \in \mathbf{C}$: A functor $(-) \times x : \mathbf{C} \to \mathbf{C}$. That is to say, the action of creating the product with any object of the category is a functor; A functor $[x, -] : \mathbf{C} \to \mathbf{C}$ forming the hom-object. Additionally, these two functors are *adjoint*.

> **Definition:** Adjoint Functors.
> Given categories **C** and **D**, functors $L : \mathbf{C} \to \mathbf{D}$ and $R : \mathbf{D} \to \mathbf{C}$ are called *adjoint* if there exists a *natural isomorphism* between *hom-functors* of the form: $Hom_{\mathscr{D}}(L(-), -) \simeq Hom_{\mathscr{C}}(-, R(-))$. In this case, L is called the *left adjoint* whilst R is the *right adjoint*.

Adjoint functors can serve as the theoretical basis for correct program transformations in the TyTra compiler [subsection 4.4.2]. They also serve as the fundamental concept required to link TyTra CL term expressions to cost-performance estimates [subsection 4.4.3] and derive an efficient DSE strategy that fuses the design-space generation process with program variant selection [subsection 4.4.5].

Before we can appreciate the usefulness of adjoint functors, a few more definitions are required. An *F-Algebras* attributes meaning to *expressions* built using functors.

> **Definition:** F-Algebra.
> An *F-Algebra* in the category **C** is the the triple (F,x, α) where F is an endofunctor on that category, the object $x \in \mathbf{C}$ is the carrier, and the morphism $\alpha : F(x) \to x$ is the evaluation function.

Conversely, an *F-CoAlgebra* is the *dual* construction of an *F-Algebra*. It defines a way to build up more complicated expressions, using a functor F from simpler ones.

> **Definition:** F-CoAlgebra.
> An *F-CoAlgebra* in the category **C** is the the triple (F,x, α) where F is an endofunctor on that category, the object $x \in \mathbf{C}$ is the carrier, and the morphism $\alpha : x \to Fx$ is the co-evaluation function.

F-Algebras, dually *F-CoAlgebras*, can also be seen as the objects of a category. Certain special objects in a category can be identified from their relationships with other objects. An **Initial Algebra**, if it exists, is the initial object in a category of $F - Algebras$.

> **Definition:** Initial Object.
> The *initial object* of a category is that unique object from which there exists a structure-preserving morphism to every other object in that category.

If one represents $F - Algebras$ as a, b, c , the objects in the category of $F - Algebra$, the *initial F-Algebra* is denoted $()$. Given that $()$ is the initial object, there are morphisms from $()$ to every object a, b, c.

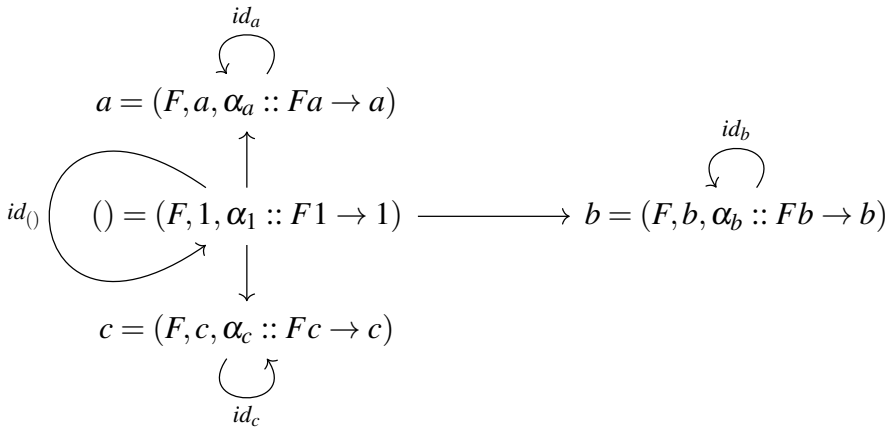


Figure 2.38: Morphisms from Initial F-Algebra to all F-Algebras in that category.

Dually, we may speak of a **Final F-CoAlgebra** if the there exists a *terminal object* in the category of *F-CoAlgebras*. A *terminal object* is the dual object to the *initial object*. Stated another way, the *terminal object* in a category **C** is the initial object in the opposite category C^{op}.

> **Definition:** Terminal Object.
> The *terminal object* of a category, if it exists, has a unique structure-preserving morphism pointing to it, from every other object in the category.

Because *F-Algebras* and *F-CoAlgebras* are objects in their respective categories, we may also speak of the morphisms in those categories.

> **Definition:** Catamorphism.
> Given an initial F-Algebra $() = (F, 1, \alpha_1 :: F\ 1 \to 1)$ and a choice of some other F-Algebra in the category of F-Algebras, $a = (F, a, \alpha_a :: Fa \to a)$, the unique structure-preserving morphism $() \to a$ is called a **Catamorphism**.

Note that we have overloaded the use of a label. The outer use denotes an F-Algebra, whilst the inner use denotes the carrier object. Catamorphisms are more commonly known as *fold* operations. We will later see that Catamorphisms can serve as the basis of an efficient implementation strategy for type inference [section 4.4.4] and cost-performance modelling [section 4.4.4] within our DSE strategy.

The dual construction to a Catamorphism is another recursion scheme called an **Anamorphism**. Anamorphisms are also referred to as *unfolds*.

> **Definition:** Anamorphism.
> Given an terminal F-CoAlgebra $\perp = (F, \perp, \omega_\perp :: \perp \to F\perp)$ and a choice of some other F-CoAlgebra in the category of F-CoAlgebras, $a = (F, a, \omega_a :: a \to Fa)$, the unique structure-preserving morphism $\perp \to a$ is called an **Anamorphism**.

Whereas a Catamorphism *removes structure*, an *Anamorphism* creates structure starting from a singluar value. The structure is defined in terms of the functor on which its underlying coalgebra is defined. The composition of an *Anamorphism* with a *Catamorphism* defines a **hylomorphism**, the theoretical foundation for the build/fold rule we have seen in section 3.2.2. This construct is useful because it gives rise to numerous optimization opportunities. *Fusing* the evaluation of a *Catamorphism* with that of an *Anamorphism* means that we can avoid the construction of certain intermediate data-structures, as we will see in subsection 4.4.5.

Chapter 3

Related Work

In the first section we cover related work that is of a practical nature. We will look at a number of *optimizing parallel compilers* and *behavioural/program synthesis* tools to highlight the difference in approach to tackling performance portability, and discuss the implications this has on the efficiency and effectiveness of an optimizing compiler. In the second section we present the theoretical related work. Our primary contribution is a practical result that stems from the theorem and proof we give in section 4.5, and so we can claim no significant contribution to category theory itself. Our theorem does however rely on a particular interpretation of key notions from the fields of *category theory*, *type theory* and *parallel programming language* research which is related to the theoretical work discussed in this section.

3.1 Practical

High Level Synthesis allows software developers to specify their application using a high-level programming language. The compiler/HLS tool maps PL constructs to equivalent HDL ones, in a process known as *behavioural synthesis*. Depending on the difference in the level of abstraction representable in the *high-level language* used to describe the application, and the *low-level language* used to *code-generate an implementation*, there can exist many alternative paths that the compiler can take whilst performing *behavioural synthesis*. There are two broad classes of parallelism that must be considered.

1. *Instruction-level parallelism*. Primitive gates, in HDLs can be trivially given meaning as an entirely parallel circuit. Instructions in *C-like* languages, which are most often used as source HLS languages, may only be evaluated in parallel after performing a potentially costly dependency analysis.

2. *Data-level parallelism*. HLS tools primarily rely on *pragmas* and platform dependent *intrinsic operations*.

By clearly separating *instruction-level parallelism* from *data-parallelism*, the burden placed on *behavioural synthesis* is greatly reduced. *Pragmas* and *intrinsic operations* can be seen as forming a *meta-language*, that is largely disconnected from the source-code that represents the actual application. This makes dependency analysis easier, as it no longer has to consider the effects of data-level parallelism. At the same time, this separation of concerns, between the language that expresses the application and the meta-language of data-level parallelism and compiler optimizations makes certain optimization opportunities inaccessible to the compiler. In concrete terms, the programmer must specify the level of data-level parallelism that should be expressed in the final, optimized, application. This is done by annotating the source code with appropriate pragma declaration, in the absence of *immediate feedback*. The programmer must compile and explicitly test the application such as to determine the effects of the newly added pragmas.

3.1.1 Imperative Languages

Most of the vendor-supported HLS tools make use of the same *OpenCL* [Gro09] standard used to program GPUs as the source framework from which FPGA implementations are synthesized.

OpenCL

OpenCL [Gro09] is a standard programming framework that provides a way to develop portable application for GPUs, CPUs, and FPGAs. What it does not provide, is a way to ensure that these applications perform to their full extent, i.e. performance portability. OpenCL C code must be tailored to the target device by specialist programmers, such that it delivers the expected performance. The OpenCL standard provides a number of constructs for program optimization that are baked into the language, in addition to the more loosely coupled techniques we have already covered: *compiler flags* and *pragma directives*. In OpenCL terminology, programs are consist of a host-side application and a number of **kernels** that execute on the compute accelerator.

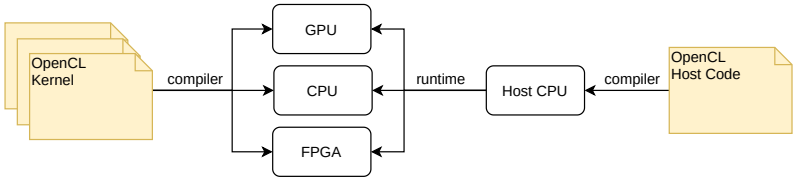


Figure 3.1: OpenCL compilation and runtime.

Kernel instances form **work groups** which are **scheduled** in a way that makes the best use of the underlying computational resources by the OpenCL **runtime** executing on the *host CPU* as shown in Figure 3.1. The OpenCL optimization workflow is shown in Figure 3.2, reproduced from *"OpenCL Programming Guide for Mac"* [App].

Figure 3.2 reveals the iterative and *highly-manual* nature of optimizing OpenCL applications. On the one hand, this is needed because of the *imperative* aspect of the *OpenCL C* programming language which is a *super-set* of the *C* programming language. The most important additional features are: *memory namespaces*, *intrinsic vector operations* and the more recently standardized *channels* feature. On the other hand, the OpenCL optimization process is also largely manual because of the execution model disparity between the different platforms that can be targeted using OpenCL. These problems are perhaps further compounded by the fact that hardware vendors have a perceived financial incentive to *not* standardize their programming interface.

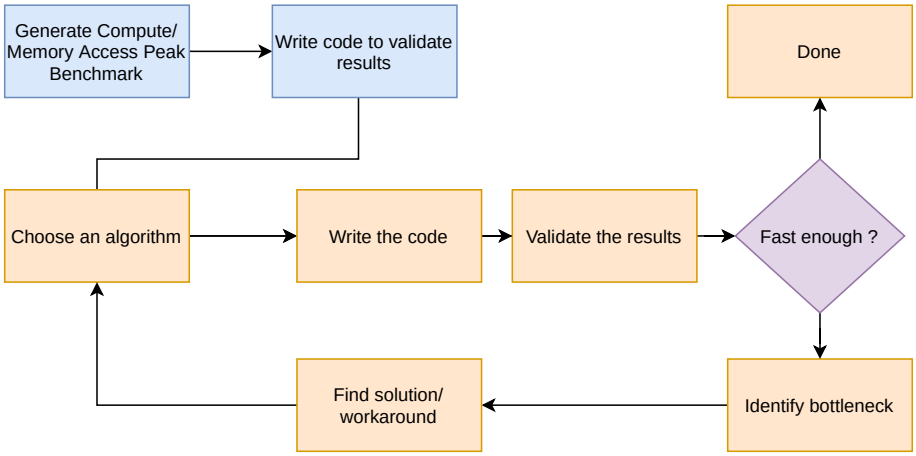


Figure 3.2: OpenCL optimization workflow.

The code we show in Listing 3.1 features an OpenCL specific optimization: *memory pinning*. This allows a programmer to explicitly dictate what memory-spaces a buffer should use.

```
cmPinnedBufIn = clCreateBuffer( cxGPUContext
, CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR, memSize
, NULL, NULL);

cmPinnedBufOut = clCreateBuffer(cxGPUContext
,  CL_MEM_WRITE_ONLY | CL_MEM_ALLOC_HOST_PTR, memSize
, NULL, NULL);
```

Listing 3.1: OpenCL Pinned Memory

This workflow highlights the iterative nature of optimizing OpenCL applications which, to a large extent, mirrors that of optimizing applications through the manual selection of compiler flags or introduction of pragma declarations

When using OpenCL, the programmer must drive the optimization process by deciding:

- The optimal size for a work group
- Which work groups can be automatically or manually scheduled
- What memory space should be used for each of the intermediary buffers

These decisions are heavily influenced by the target hardware architecture. Whilst OpenCL is most popular for GPU programming, implementations exist for CPUs and FPGAs as well. Xilinx's SDAccel [Xil14] and Intel's AOCL [CAD+12] allow programming FPGAs using the standard OpenCL syntax. Optimization of the application is largely in the hands of the application's developer, whom must specify where and how the application should be transformed by the compiler through the introduction of the previously mentioned *pragma* declarations. The most common and effective transformation is *loop unrolling*

```
__kernel void example( __global const int * restrict x, __global int * restrict sum){
 int acc = 0;
 #pragma unroll
 for (size_t i=0; i < 4; i++) {
  acc += x[i + get_global_id(0) * 4];
 }
 sum[get_global_id(0)] = acc;
}
```

Listing 3.2: Altera OpenCL Loop Unroll pragma.

The introduction of the *loop unroll pragma* can lead to increased performance by telling the compiler that a for loop body can be safely instantiated multiple times, allowing multiple data items to be processed concurrently. In the example show in Listing 3.2 the AOCL compiler can further improve performance by *coalescing* the concurrent read operations required to execute evert loop body instance. These optimizations can only be applied if dependency analysis can show there are no *hazards* within the loop body. Intel's HLS Compiler [1] is in many ways similar to OpenCL solutions that target FPGAs. It is a HLS tool that processes C++ into RTL optimized for Intel FPGAs. As with OpenCL, Intel HLS also requires explicit unroll annotations to inform the compiler about instances of kernel invocations that should be parallelized. Syntactically the pragma declaration is identical to that in Altera OpenCL.

```
#pragma unroll
for(int i=0; i < N; i++) {   ...  }
```

Listing 3.3: Intel HLS Loop Unroll

[1] https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01274-intel-hls-compiler-fast-design-coding-and-hardware.pdf

Weller et al [WOL+17] demonstrate that FPGAs, programmed via OpenCL, show promising performance when implementing partial differential equations based scientific models. However, they also indicate that they had to write the OpenCL kernel code in a slightly different manner, depending on the targeted hardware device, in order to obtain good performance. This requirement for the manual tweaking of the implementation, to account for hardware-dependent behaviour, is present in Intel's HLS tool as well. Take for example the advice provided within the Intel HLS best practices guide [2] under section 3.1.5 , related to the performance of passing a small arrays of values as an argument to, and a return value from a *component*, or function. On FPGAs, packing all values together into a *struct* and thereafter passing that structure in and out of the component, by value, is likely to lead to better performance than indirect access through an array pointer. This advice may seem counter-intuitive to a software developer that used to writing code for a CPU where such packing and unpacking would only introduce overhead. The downsides of segregating the programming language used to define the program's behaviour, from the *meta-language* used to describe the program optimization begins to show through in Listing 3.4.

```
struct int_v8 {
 int data[8];
}

component int_v8 vector_add(int_v8 a, int_v8 b) {
 int_v8 c;
 #prama unroll 8
  for (int i = 0; i < 8; ++i) {
   c.data[i] =   a.data[i] + b.data[i];
 }
 return c;
}
```

Listing 3.4: Intel HLS example with a small array operation.

The *loop unroll factor* which serves as a parameter to the loop unroll pragma on line 7 has a concrete value of 8. The same value is used to define the size of the *struct* on line 2 which is in turn used to pass the entire array of values in and out of the component. Finally, this value is also included as a textual hint within the structure's name, defined on line 1. An experienced programmer may recognize the connection between all occurrences of the character '8' within this block of code; however, the disconnect between the source and meta languages means the compiler may not be aware of this connection.

[2] https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls-best-practices.pdf

Vivado is Xilinx's own HDL synthesis and analysis suite. Increasing parallelism requires developers to annotate their solutions with appropriate directives. Whilst Vivado provides a diverse range of directives, such as partial loop unrolling and automatic loop fusion, due to choice of internal representation, the user has to take up responsibility for ensuring the transformations preserve correctness [3]. Winterstein et al. show a more detailed picture of the types of manual transformations a user must perform such that Vivado can perform HLS [WBC13].

Maxeler

Maxeler [PA12] is yet another good example of a mature HLS framework. In contrast to OpenCL for FPGAs or Intel's HLS tool, the Maxeler approach is centred around a custom dataflow language that is embedded within the Java programming language. The programmer writes *Java code* which is then used to *code-generate* a *.max* file that describes the computation to be performed and how data is to be marshalled to and from the device. Embedding a data-flow description within the programming language itself allows the compiler to reason about and manipulate expressions that exist on two levels, at once:

- The *object level* expressions that represent the actual computation to be performed. These are the expressions and values computed as part of the *"real work"* performed by the application.
- The *meta level* expressions that dictate how the application's terms are manipulated.

The additional compilation step, from *Java code* to *.max*, imbues the application with a certain degree of reflection or introspective capability. Introducing a further *intermediate representation* allows information to flow between the *object* and the *meta* levels much more freely than *pragma directives allow*. In particular, information can flow in both directions between the object and meta language constructs. Even with a dataflow approach, the process is still largely manual.

The "Maxeler DFE Debugging and Optimization Tutorial" enumerates [4] the following steps: Create a C application; Determine code segments to accelerate; Debug through simulation; Debug through hardware synthesis; Optimize. These steps are the same as those we needed for OpenCL application development on FPGAs. There are however certain advantages with a dataflow language. For example, the *by value copy operation* in Listing 3.4 translates into a simple assignment, eliding the need to pack and unpack operations before and after every transfer.

[3] `http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Improving_Performance.pdf`

[4] `https://www3.diism.unisi.it/~giorgi/teaching/esercitazioni/esercitazioni217/maxdebug-tutorial_45f1310.pdf`

OpenACC

OpenACC which stands for *open accelerators*, is a *directive-based* programming model for heterogeneous parallel computation. The programming model is somewhat similar in spirit to OpenMP in that it requires the user to annotate code with an expression of the intended degree of parallelism. OpenACC has a larger scope than OpenMP a it targets both GPUs and CPUs. OpenACC can be used alongside OpenCL and CUDA [HGMS+12] [RLFdS12], but it can also be used as a replacement for these programming frameworks.

As is the case with OpenCL, CUDA and OpenMP, the OpenACC approach to application optimization is also largely manual. The user annotates existing code with pragmas which instruct the compiler on how the application should be parallelized. The benefit is that the core algorithms, implemented in the application, can be generally left untouched whilst enabling the application to run on CPUs and GPUs alike. The unstated requirement here is that the implemented algorithm must be amenable to optimization via the transformations specified by these directives. The *kernels* OpenACC directive, for example, can be used on a loop construct. It instructs the compiler to extract a *kernel function* from the loop body and attempt to accelerate the computation by automatically parallelizing the implementation of the extracted kernel.

```
#pragma acc kernels
for (int j = 1; j< n-1; j++)
{
 for (int i = 1; i <m-1; i++)
 {

  a_new[j][i] = 0.25 * a[j][i+1]
   + a[j][i-1] + a[j-1][i] + a[j+1][i+1];

 }
}
```

Listing 3.5: OpenACC stencil computation acceleration.

In contrast, the TyTra compiler automatically extracts kernel functions without the use of pragma directives. This means that the optimization can be fully automated but poses a problem for design space exploration. Automatic kernel extraction in TyTra means that the generated search-space can be many times larger than a manually identified search-space, where the expert programmer has already identified the kernels that are most likely to impact performance.

On the other hand, the *cost-performance model* in the TyTra compiler can be used to determine an optimal parallelization strategy automatically, whereas OpenACC compilers rely entirely on user-provided guidance. The *kernels* directive shown in Listing 3.5 tells the OpenACC compiler what loops to parallelize, but it does not indicate weather or not those loops are worth parallelizing, and if so, to what extent they should be unrolled. Furthermore, every annotated loop-nest is treated independently. This means that entire data set needed by each loop-nest may be moved from the CPU to the GPU accelerator and back again, only to be migrated once more for a subsequent loop nest.

```
int iter = 0;
#pragma acc data copy(A), create(Anew)
while ( iter < iter_max)
{
 #pragma acc kernels
  for (int j = 1; j< n-1; j++)
  {
   for (int i = 1; i <m-1; i++){
    a_new[j][i] = 0.25 * a[j][i+1]
     + a[j][i-1] + a[j-1][i] + a[j+1][i+1];
   }
  }
  // [..]
  iter_max++;
}
```

Listing 3.6: OpenACC stencil computation acceleration with data-flow hints.

The mitigation strategy in OpenACC is of course, another set of pragma directives. The *copy* directive instructs the compiler to copy the entire data-set needed by the loop nest to the GPU before the while loop begins executing, and to copy it back once all iterations have been performed. The *create* pragma instructs the compiler to allocate storage for the intermediate value store *a_new* on the GPU, and to keep it on the GPU until the while loop is done executing. Although the addition of these pragmas does not change the structure of the algorithm's implementation, it nonetheless depends on this structure. In effect, the programmer must specify enough information about the application's data-flow, as to indicate to the compiler what the best optimization strategy is. Given an accurate cost-performance model, such as that in the TyTra compiler, the information encoded by these additional pragmas can be recovered automatically.

DSE for kernel loop optimisation

Zhong et al. [ZVL+14] also use Design Space Exploration to optimize applications that consist of multiple loops for execution on FPGAs. Their method of reducing the design space to be explored also relies on *data-flow dependence* information, however there are a number of key differences between our respective approaches.

Solution	***Complete Exploration***	***Dataflow Detection***	***Scope***
[ZVL+14]	✗	part of DSE	Kernel-level
Ours	✓	before DSE	Application-level

Table 3.1: Difference in approach between our solution and [ZVL+14].

The first difference is that the TyTra *cost-performance model* gives us accurate throughput, latency and resource. Thus, with the improved DSE methodology we propose that makes use of these estimates, a fully working solution can be found with zero HLS invocations and deployed for execution with a single invocation. In contrast, the method presented in [ZVL+14] requires a number of HLS tool invocations to sample the cost-performance space and guide the search. Because of the *accurate cost-performance mode* in TyTra, our optimized DSE strategy is also complete meaning that we do not miss any of the *pareto-optimal* design points. In contrast, the authors of the related work report that: *"we can observe that the approximate Pareto-optimal curve by our method does not cover all the Pareto-optimal design points [...]"* [ZVL+14].

The second difference has to do with the characteristics of the language in which applications are expressed. As we derive *structured design spaces* form the structure of TyTra CL applications, there is no need for an additional *"Dataflow Detection", as in the related work, which "checks whether dataflow feature can be enabled for the application"* [ZVL+14].

The third and most crucial difference is that the focus in this particular related work is on the *optimization of kernels functions*, invoked as part of a larger application. Zhong et al. attack the problem of kernel optimization by transforming the *loop operations* that occur within them. The related work thus solves the much smaller problem of determining the optimal *loop unrolling schedule* for each kernel function in isolation. In contrast, we optimize the parallel structure of the *overall application* by taking account of the interactions between *all kernel functions*. The kernels, or what we call *opaque functions* in TyTra CL are separately optimized to achieve the highest degree of internal pipe-lining [NV19].

3.1.2 Functional Languages

The single most important difference between imperative and functional programming language has to do with the notion of *side-effects*. Computations with side-effects are of particular interest to us because the bridge the gap between the *operational or imperative* and the *denotational or functional* models.Imperative programming not only allows, but often encourages the use of *side-effects* to perform computation. Writing to a *global variable* can be seen a type of *side-effect*. An application might make use of global variable writes to coordinate the efforts of various concurrent threads of executing code. Functional programming, on the other hand, is entirely premised on the idea that *side-effects* should never occur.

Algebraic Effect Handlers can be used to represent, or rather simulate, *side-effects* in functional programming languages, whilst maintaining the property of safety that functional programmers enjoy. Algebraic effect handlers embody an often used, two-step, problem-solving technique:

1. Decomposing the main problem into a collection of sub-problems and specifying a sequence of actions that solve them. More generally, the collection of actions may be any *structure*, for example a tree rather than a list.

2. Implementing an interpreter that evaluates the structure of actions or sub-problems.

Certain problems may be broken down into sub-problems that have possibly many solutions. In such cases, we would represent the solution as a *tree of alternative sub-solutions*. The interpreter would then pick the appropriate action, for each sub-solution, before finally *composing sub-solutions*. If this description appears to be familiar, it is perhaps because:

1. Generating a *structure of sub-problems*, or equivalently, generating a *structure of sub-solutions* are both examples of **Anamorphisms**

2. Folding sub-solutions into an overall solution is an example of a **Catamorphism**

3. Effect structures are like configurable Monads. An *effect-free* computation is the equivalent of a *monadic return*. The *monadic bind* operation is just one type of effect, the *sequencing effect*.

4. Generating sub-solutions and then picking one, is a description of **Design Space Exploration** we have given repeatedly in the present work.

The similarities or breath of overlapping scope among these approaches to bridging the *imperative-functional* gap can be readily observed. As we will see in the following chapter, these methods and techniques are can be modelled and explained trough Category Theory.

Lift Language and Compiler

Lift The Lift Project also aims to bring performance portability to CPUs, GPUs [Ste15] for linear-algebra, sparse matrix and stencil computations, by transforming applications using rewrite rules. The Iterative application of these re-write rules generate a search space of program transformations with varying degrees of expressed parallelism and resource use. Due to the wide set of target architectures the search space can grow at a rather alarming pace. The authors have proposed allowing developers to specify their own heuristics, or tactic, to help guide the search space. The Lift projects shares many of the same ambitions as for the TyTra project and is thus one of the more closely related works we must consider. At a surface level, the approach to program optimization in the Lift project is not too dissimilar to our own: a sub-set of optimizing transformations in Lift overlaps with those in TyTra. Fundamentally however, there are two key differences in approach.

1. The optimizing transformations in Lift are specified as rewrite rules based on: *a) algorithmic skeletons and b) target or problem specific optimizations*. We recover a more basic definition for optimizing transformations in terms of category theoretical notions, by extending the approach of defining *categorical data types*, covered in subsection 3.2.2.

2. The approach to solution discovery in Lift is presently two-fold. The first approach to optimizing transformations is specified as a *search-space* that the application developer must manually traverse by writing an *optimization schedule*. The second approach, involving an *automatic* solution discovery method is covered in [SFLD15]. Here the key difference is that the approach in Lift is *stochastic*, whereas our own is analytical.

We believe that these differences in approach are largely due to two factors. The first factor is that the Lift project primarily targets GPUs whilst TyTra focuses on FPGAs. This, we believe, has a large influence on the design of a programming language and its compiler. As TyTra primarily targets FPGAs, we were lead down the path of phrasing the optimization of an application as both a *spatial layout* and *temporal scheduling* problem, whereas in Lift the problem boils down to *temporal scheduling* only. This causes Perhaps counter-intuitively, this makes solution discovery somewhat easier. The *spatial layout* sub-problem naturally leads to the generation of additional *constraints* on the search-space. Additionally, the configurability of FPGAs means that application performance and resource use can be more accurately modelled, whereas GPUs are somewhat *opaque* to performance analysis. The second has to do with the *ad-hoc* nature of the transformations rules. The automated search algorithm in Lift makes use of *"macro rules which perform multiple small steps at once by applying a set of rules in a predefined order"* [SFLD15]. These macro-rules appear to be manual optimization integrated by Lift's authors, yet the example given that, that of a *map-reduce fusion* macro-rule indicates that these are a sub-set of what can be *automatically* derived from *type-level equivalences* in TyTra.

Expanding upon the first difference we see that the optimizing rewrite rules in Lift are of two primary types. The first type of rule can be roughly identified with *parallel skeletons* [SFLD15]. Examples include the fusion (Figure 3.3) and cancellation (Figure 3.4) rules shown below.

$$map\ M \circ map\ N \rightarrow map(M \circ N)$$
$$reduceSeq\ M\ N \circ mapSeq\ P \rightarrow reduceSeq(\lambda(acc,x).M\ (acc, P\ x))\ N.$$

Figure 3.3: Fusion rules in Lift.

$$join \circ split\ I \mid split_{A,J} \circ join_{A,J} \rightarrow id$$
$$joinVec \circ splitVec\ I \mid splitVec_{A,J} \circ joinVec_{A,I,J} \rightarrow id$$

Figure 3.4: Cancellation rules in Lift.

The second type of Lift rewrite rule is derived from OpenCL specific transformations, examples of which are shown in Figure 3.5. These map the abstract operations defined by an expression into concrete implementations that match the semantics of the OpenCL framework.

$$map\ M \rightarrow mapWorkgroup\ M \mid mapLocal\ M \mid mapGlobal\ M \mid mapSeq\ M$$
$$reduce_{A,I}\ M\ N \rightarrow reduceSeq_{A,A,I}\ M\ N$$
$$reorder_{A,I\times J} \rightarrow reorderStride_{A,J} I \mid id$$

Figure 3.5: The OpenCL Specific *map*, *reduce* and *reorderStride* rules in Lift.

Both types of rules are meant to transform a *high-level* expression in a singular and very specific way. The two classes of rules are related through *denotational semantics*. OpenCL-specific map routines are, for example, denoted by the same term as the abstract *map* operation, as shown below in Figure 3.6.

$$[\![mapWorkgroup]\!] = [\![mapLocal]\!] = [\![mapGlobal]\!] = [\![mapSeq]\!] = [\![mapVec]\!] = [\![map]\!]$$

Figure 3.6: Denotational semantics relating the various *map* implementations to an abstract *map* operation in the Lift compiler.

We note that the map operations shown Figure 3.6 are manually specified to be equivalent by the compiler developer. This limits the utility of the type system in Lift to the task of *verifying* that this equivalence is sound. In the TyTra compiler there are *no target-specific* implementations, however, the latter three terms in Figure 3.6 have direct equivalents. The first and last terms: *mapSeq* and *map* are simply represented as *map* operation. The middle *mapVec* implementation is represented by *splitting* the input to the *map* operation and *merging* the result. At the same time, the $[\![mapVec]\!] = [\![map]\!]$ equivalence is recovered automatically from the $merge_k \circ split_k$ identity that falls out of the $Vec_{k\times m}\ a = Vec_k\ (Vec_m a)$ *type-level equality*. This level of automation, however, is only practical if an *efficient DSE strategy*, such as the one we presently contribute exists.

The second major difference between the approach in Lift and our own has to do with automatic program optimization search algorithm. In Lift this is derived from *Bandit-based optimization* [dMRVP09], and is *"rather basic and just designed to prove that it is possible to find good implementations automatically"* [SFLD15]. Concretely this is based on the *Monte-Carlo* method, and is thus a *stochastic* routine. Broadly it can be described as an iteration of the following steps, [SFLD15]:

- For each high-level expression, determining the set of all valid rewrite rules.
- Randomly applying a sub-set of these rules.
- Executing the code-generator for each selected set of rewrites.
- Measuring the performance of each thus generated solution.

Once the best-performing solution has been determined for the sub-set corresponding to the high-level term, it is used as the *input* to the subsequent iteration consisting of the same steps. This process is repeated until a "terminal expression" is reached, although the meaning of a "terminal expression" term is not explained in [SFLD15]. Whenever a target-dependent rewrite rule takes a numerical parameter, the authors indicate that they "limit the choices [..] to a reasonable set appropriate for our test hardware". This indicates that even the automated solution requires manual intervention from the application developer. The definition of what constitutes an appropriate set is not discussed fully discussed either, leading us to believe that the set is to be determined through trial and error, on a case-by-case basis. Given the primary target device (GPU), the absence of an analytical *cost-performance model* and the *ad-hoc relation* that ties OpenCL-specific transformation rules to the high-level ones, we feel that the approach taken in Lift, although far from ideal in terms of compilation time, is appropriate for the stated goal. Without an analytical *cost-performance model* the numerical parameters can not be automatically discovered. Without a deeper *semantic connection* that can interpret *hardware or problem-specific rewrites* as *design-space transformations*, there is no choice but to fully or randomly sample the search-space thus defined.

We believe that the Lift compiler could be extended to benefit from our analytical *design-space exploration* strategy, despite the different choices of *primary hardware target* and *optimizing transformation representation*. This last comment is entirely aligned with the vision for Lift's future expressed by the authors: *"We envision replacing this exploration strategy in the future by using machine-learning techniques to avoid having to search the space at all"* [SFLD15]. In more concrete terms, we suggest that a possible way forward is to overcome the fact that GPUs are relatively opaque to *cost-performance analysis* by deriving a *reasonably accurate cost-performance model* through other stochastic methods, perhaps the application of *machine-learning*. We also suggest that a *broader set of exploration-optimising* macro rules could be automatically derived by adopting the *type-driven transformation* approach in TyTra.

3.1.3 Program and Behavioural Synthesis

We have seen that different programming languages tend to *favour* certain models of computation, yet the compilers that support these languages often make use of multiple such models as they steadily transform the source-code into an ultimately optimized piece of executable code. It would stand to reason that it may be possible to transform applications written in an *imperative programming language* into *functional programming language representations*, such as to make use of the optimizations available in both paradigms.

To a certain extent, that is precisely the case with the High-Level Synthesis tools we've looked at in section 3.1. With HLS tools *source* language is often an imperative programming language such as *C*. The *target* is usually one of the HDL standards: perhaps *Verilog* or *VHDL*. HLS straddles the fine line between programming paradigms by way of *code-generation*. The language that the application writer interacts with is more closely aligned with a particular model of *algorithms* or an abstract notion of the meaning of the application. HLS tools often adopt the syntax of programming languages intended for other hardware targets, such as CPUs and GPUs to leverage existing programmer familiarity, at the cost of perhaps a more loose coupling between the semantics assumed by a software engineer familiar with these platforms, and the semantics demanded by the *circuit model* of computation assumed by the tooling such as hardware place and route further downstream. The two primary issues with existing HLS tools is are as follows. For one sub-set of these, the issue is of a practical and syntactic nature. The design choices taken in developing the language used to express algorithms is too closely aligned to either an operational view of computation. The remaining sub-set, although it adopts a functional view of computation, fails by not accounting for a more uniform view of computational semantics.

Similar to how HLS tools generate circuits from specifications of algorithms, synthesis in the context of functional programming languages refers to generating programs from specifications of the problem they seek to solve. *ADATE* is a system for automatic functional programming [Ols95]. It makes use of a specification of constraints to synthesis correct, novel and often unexpected functional programs that feature recursive behaviour and useful auxiliary recursive functions. The synthesis process involves an incremental approach: better programs are developed through refinement. The key to success seems to be that ADATE a systematic search process that traverses sequences of transformations.

This incremental synthesis process makes use of transformations such as *expression replacement*, *function abstraction*, *case distribution* and *type embedding* whilst being guided by a measure of *program correctness*, *syntactic complexity* and *time complexity*. Correct *program synthesis* from *specifications* is a task that is in many ways related to the issue of efficient design space exploration that we are trying to solve:

- We can think of the TyTra CL as a specification of the application to be synthesized. The optimizing transformations may not generate an entirely new implementation that meets this specification, but they generate *a different* implementation that hopefully performs better.
- We may also think of the process of optimizing our DSE strategy as being a *program synthesis* task. We use the TyTra CL specification to synthesize and optimal DSE strategy.
- The DSE process itself requires that we synthesize multiple program variants and ultimately select the best performing one.

The fact that we are able to relate DSE to program synthesis in not one, but three ways hints at one benefit of functional programming. We can establish relationships between program transformations and *transformations of the compilation process*.

3.1.4 Compiler Optimization Techniques

Polyhedral transformation

In subsection 2.4.2 have seen the relationship an input vector size and a nesting of *map* operations. This indicates that our solution is in part related to the well known *polyhedral transformation* technique, on which many parallelizing compilers rely [Bas04]. Each of the possible permutations of prime factors that collectively would correspond to a type of polyhedral transformation that takes a purely sequential implementation of a loop nest to a parallel one.

Collective optimization

As our DSE optimizations source heuristics from the interactions of the *terms*, *types*, *cost-performance model* and the DSE strategy itself, it stands to reason that different applications having necessarily distinct term-level representations may nonetheless share certain *term-level patterns*. Rather than explore all interactions between these representations at the application's compilation time, we believe that a single exploratory run and the use of *memoization* can achieve the same result. This would indicate that our solution might be seen as a *speculative* (and functional programming language) equivalent of *collective optimization frameworks* such as described by Fursin et. al in [FT10].

3.2 Theoretical

3.2.1 Structured Parallelism

Parallel Skeletons

One such technique is to use of well-known parallel structures or patterns, sometimes called *Parallel Skeletons*, that naturally arise from certain functional programming constructs. Cole argued in his manifesto [Col04] for a pragmatic approach to parallel programming and introduced the **eSkel** library. This is based on the *C* programming language and the *MPI* framework The eSkel library is a perfect example of what it means to bridge the gap between programming language paradigms: an imperative programming language and parallel programming framework are used together with formalism arising from functional programming to solve a real world programmability issue. Cole's manifesto is not only instrumental the entire domain of structured parallel programming at large, but it has also played a significant role in shaping the view-points that underpin the TyTra compiler. Our only criticism is that *eSkel* does not go *far enough* in its stated goal of *raising the abstraction layer*. We quote:

"Skeletal programming is not functional programming, even though it may be concisely explained and expressed as such. Nor is it necessarily object-oriented programming, although the increasing interest in such technologies for HPC will make such an attractive embedding viable soon" [Col04]

We believe that time has indeed come and that *design space exploration* can be used to bridge the small remaining gap between functional programming and the imperative languages that continue to benefit from widespread adoption and support in industry. *Parallel skeletons* are conceptually simple yet very powerful abstractions. A skeleton is simply a *recipe*, or a *pattern of computation* that can be reused. The Java programmers amongst us might liken each skeleton to an *Interface*. Each parallel skeleton is backed by a number of alternative implementations. One implementation may be optimized for CPU execution, whilst others might target GPUs or FPGAs. The programmer's sole job is to reference the parallel skeleton that abstractly remains the type of parallel computation they wish to perform, leaving the tedious job of providing an efficient implementation to the skeleton library. The remaining issue to be solved is simply the **automation** of the mapping of abstract skeletons selected by the programmer to the *best implementation*.

Recursion schemes

Recursion Schemes are related to *Parallel Skeletons* in that they represent *patterns of recursive behaviour*. The connection between recursion schemes and parallel skeletons is also noted by Barwell et al. [BBH18] who describe a way to perform *recursion scheme identification*. Their technique looks for certain instances of recursion schemes that are known to have parallel implementations defined as parallel skeletons. In functional programming, recursion schemes are fundamental constructs that continuously reappear. Terminology can vary to such an extent that one may be intimately familiar with recursion schemes and not know it. For now, we will briefly present two of the most important recursion schemes. More formal descriptions will follow in subsection 4.1.1.

- **Catamorphisms**, also known as **folds** or **reductions**, consume a *structured collection of data values* producing a singular output value.
- **Anamorphisms** are the opposite of *Catamorphisms*. They are also known as **unfolds** or **generators**. Anamorphisms take *a data value* as input, producing a *structured collection of data values* as output.

Catamorphisms and *Anamorphisms* are generalized versions of the *fold* and *unfold* operations that operate on lists. More importantly, composing these two recursions schemes, in certain circumstances, yields another recursion scheme called a *hylomorphism* which is more efficient than running the component computations sequentially. In subsection 3.2.2 that follows, we will see a that there exists a category-theoretical construction for *data types*. In subsection 4.4.1 we will further see how the recursion schemes briefly introduced here are connected to categorical data types.

3.2.2 Categorical Data Types and Techniques

Modelling Data Types in category-theoretical terms is not a new concept by far. Whereas our approach was to recover category-theoretical semantics for the TyTra compiler after the TyTra CL, IR and cost-performance model where largely defined through previous work, others have remarked on the expressive power of categorical semantics in developing applications for parallel computers long before this work began.

In *Foundations of Parallel Programming* [Ski05], Skillicorn laid out a precise way of developing applications that can be executed by parallel computers. The approach is described as a generalization of abstract data types, which is based on categorical data types *"as a model of parallel computation, that is an abstract machine that decouples the software level from the hardware level"* [Ski05].

One of the benefits of this approach, touted by Skillicorn is that *"the categorical data type approach [..] provides both a structured way to search for algorithms, and structured programs when they are found"* [Ski05]. Through our own experience we found these statements to be true. By providing a categorical data type interpretation for the TyTra CL, shown in section 4.1 we were able to find a structured optimization search algorithm, our DSE strategy, though a process that is itself highly structured.

There are indeed many similarities between our work and that of Skillicorn. Amongst these is the central idea of using category theory to precisely describe *data types* and their operations, and then use those detailed descriptions to optimize the application. This idea appears so frequently that even Skillicorn felt compelled to state that *"A book like this does not spring from a single mind"* [Ski05] and mentioned being introduced to the idea of constructing categorical data types by Malcolm who also authored [Mal90] and [Mal89] that breach this topic by extending the ideas previously shown by Bird [Bir87] and Meertens [Mee86]. There are however also a number of key aspects that differentiate our work from that of Skillicorn, Malcom, Bird and Meertens. In *Foundations of Parallel Programming*, Skillicorn focuses on the use of categorical data types to *develop applications*, rather than on the issue of optimizing legacy applications by recovering the underlying categorical data types as we do. The difference here is rather subtle and is best described using Futamura's words below.

"There are two methods to formally describe the semantics of programming languages. One of them is to describe the procedure by which the language to be defined is translated into another language whose semantics are already known. The other is to describe a procedure evaluating the results of a statement belonging to the language to be defined" [Fut99]

Because of the hardware we target, meaning FPGA devices, and that of our objective - maximizing performance, we were directed towards taking *both views* of semantics at once. One aspect of the semantics we give to the TyTra CL are purely based on a translation to category theory terms, as in the case of Skillicorn, the other is based on the evaluation of a TyTra application as an FPGA circuit. Skillicorn makes use of the same *dual view* of language semantics to derive *a strategy for parallel cost calculus* in the context of targeting *list-processing applications* towards an abstract model of a parallel machine, a *PRAM* that consists of a set of abstract processors and a large amount of shared memory. The difference between our work and Skillicorn's, in this matter, is roughly the same as that between our work and Castro's *Structured Arrows* [Cas18b], which we discuss later in this section, namely: we utilize categorical data types and an *accurate cost-performance model* to efficiently find an optimal program transformation, whereas Skillicorn and Castro use categorical data types to derive a *cost model*. In the book, Skillicorn also touches upon the topic of optimizing *search problems* but only focuses on the specific issue of optimizing applications that perform *structured text queries*.

Distributable Homomorphisms

In [Gor96], Gorlatch identifies a class of problems, called *Distributable Homomorphisms* that can be given efficient parallel implementations, and specifies a solution by derivation in the Bird-Meertens formalism. This general solution is given under the assumption that the number of processors is bounded. We believe the same limitations may apply to our solution, and that the reason why we are were able to achieve our results is because FPGAs are inherently bounded. A search for an optimal implementation on an FPGA depends on finite limits imposed by the *space of hardware resources available*, whereas a CPU centric approach is unlimited as the only search dimension is *temporal*. Although we did not use Gorlatch's work in deriving our DSE strategy, we believe the work on *Distributable Homomorphisms* and the stated requirement that for a bounded number of processors is can be seen as a partial bridge, from Skillicorn's *Parallel Cost calculus* on the PRAM execution model, to the use of a *cost-performance model* in TyTra to find an optimal transformation schedule.

Structured Arrows

As in the case of Skillicorn's *Foundations of Parallel Programming* discussed at the beginning of this sub-section, Castro's work on *Structured Arrows* solves the *dual issue* of deriving a *cost model* from a categorical specification of *parallel application*. Castro's *Structured Arrows* [Cas18b] framework tackles the issue of parallelizing sequential code by abridging *algorithmic skeletons* to *structured parallelism* through the use of *Hylomorphisms* to represent the functional behaviour of parallel programs and to define a mechanism for deriving *cost models* for parallel applications using the operational semantics of *queues*.

This approach, we argue, can be augmented by reasoning about the compiler, its *type-checking phase* and the eventual *run-time* in the same terms as those used to describe the application, thereby leading to greater optimization potential. Castro's use of *Hylomorphisms* to describe the application's term language can be easily paired with our view of DSE as a *hylomorphism*. We show proof of this through our use of the *Böhm-Berarducci encoding* to describe the TyTra CL and the results that we obtain. The *Böhm-Berarducci encoding*, which we describe in the following paragraph, naturally reveals the *hylomorphic structure* of categorical data-types, although we have not set out the explicit goal of formally defining a core language centred around Hylomorphisms. The optimization power of *Hylomorphisms* can be understood by examining the related work on the *build/fold rule*, a *specialization* of *Hylomorphisms* to the data-type structure of lists, which we describe at the end of this sub-section.

Böhm-Berarducci encoding

The *Böhm-Berarducci encoding*, as described by Kiselyov *"translates algebraic data types and the operations on them into System F, which contains only abstractions and application base type constants"* [Kis12]. We use the *Böhm-Berarducci encoding* to recover *structured higher-order functions* that are specialized to the *structure of the application* that is being optimized. Effectively, these *higher-order functions* allow us to apply efficiently apply multiple *evaluators* by dynamically selecting the *evaluation morphism* that will replace each of the data-types constructors. Much of the same effect can be obtained using *type classes* in what are known as the *Typed Tagless Final Interpreters* [Kis10].

Build/Fold rule

Gill et al. [GLJ93] remark on the use of list data-types as glue between dependent computational phases, and they propose an automatic technique to improve the performance of such programs by removing intermediate list representations that relies on a single, local transformation, which they call the *build/fold rule*. On a theoretical level, the build/fold rule corresponds to the relationship Anamorphisms, Catamorphisms and Hylomorphisms we briefly discussed in section 3.2.1 as we now show in Figure 3.7.

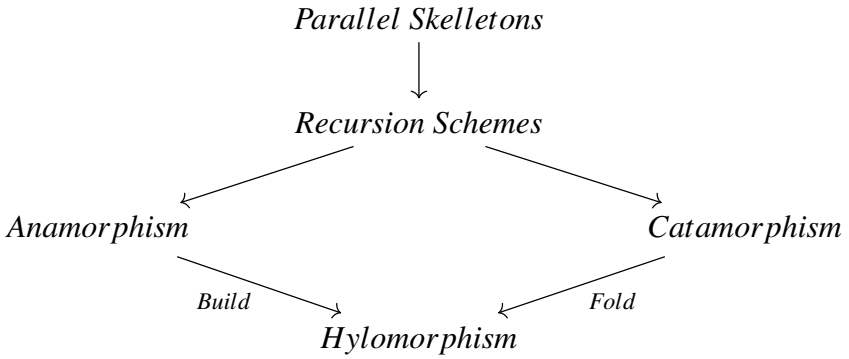


Figure 3.7: The build/fold rule corresponds to the Hylomorphism recursion scheme.

In many ways our optimized DSE strategy is a specialized version of this result, however, by using the *Böhm-Berarducci encoding* we can effectively generalize the *build/fold rule* from its simple application to *intermediate lists of results* to a broader use when constructing/consuming arbitrary data-types. Gibbons [Gib06] remarks on this build operator's argument being a Church-encoded representation of a *List* constructor. As Church-encoding is the untyped relative of the Böhm-Berarducci encoding, it stands to reason why our choice of encoding *terms*, *types*, and *cost-performance* records into *Böhm-Berarducci* terms enables us to reap the benefit of short-cut deforestation across all stages of the compiler.

3.3 Graphical summary

We concluded chapter 2 by showing that, despite the architectural differences between CPUs, GPUs and FPGAs, such devices are largely programmed and optimised using the same programming languages, tools and abstractions. At the end of that chapter, specifically in Figure 2.7, we hinted at the connection between the program transformation abstractions commonly used in frameworks such as OpenCL and more sophisticated DSE solutions. In the same spirit, we now sketch the connections between practical and theoretical related work presented in this chapter.

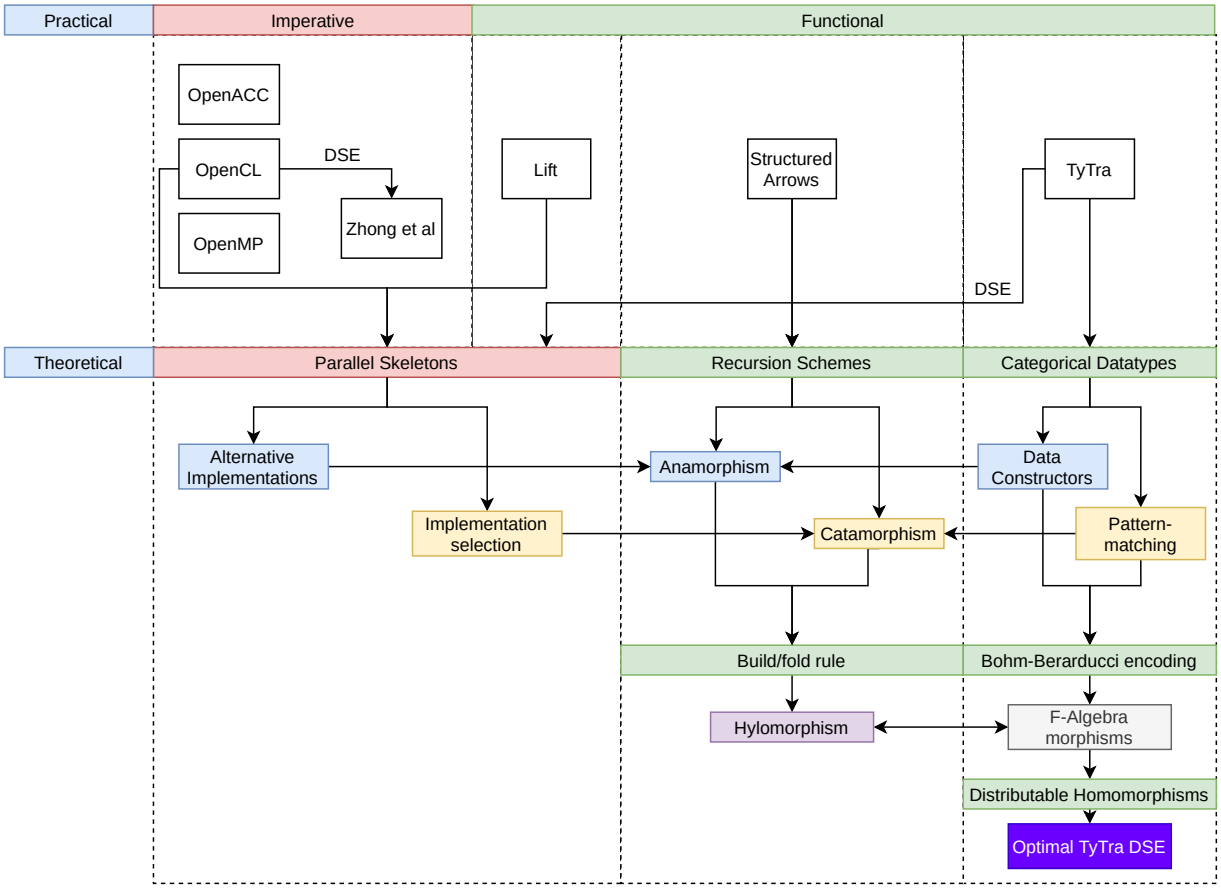


Figure 3.8: Overview of the relationship between practical and theoretical related work.

The upper-left quadrant of Figure 3.8, labelled with the *Imperative* heading, shows that traditional parallel programming frameworks such as OpenCL can be directly augmented with DSE techniques such as that of Zhong et al. [ZVL+14]. The lower-left quadrant, headed *Parallel Skeletons* shows the technique used by a more sophisticated optimization solution, the Lift compiler [Ste15] which is implemented in a functional programming language, Lift uses rewrite rules, corresponding to parallel skeletons, to optimize an imperative programming language specification of the application. Limitations imposed by two-dimensional drawings mean that the right-hand side of Figure 3.8, corresponding to Structured Arrows and our own solution only shows part of the overall story.

The approach taken by Castro in the presentation of *Structured Arrows* [Cas18b] was to give denotational semantics in terms of Hylomorphisms as a unifying construct for *Parallel Skeletons*. The application to be optimized is expressed as the composition of Hylomorphisms with *Structured Parallel Processes* which in turn are a representation of computation as the composition of Parallel Skeletons. The denotational semantics of Parallel Skeletons as Hylomorphisms is specified for the **object-language**, allowing one to understand the application being optimized as a Hylomorphic structure that can be transformed to alter its performance characteristics safely. The semantics given for the object language constrains the **meta-language** that describes the optimizing transformations the compiler may apply.

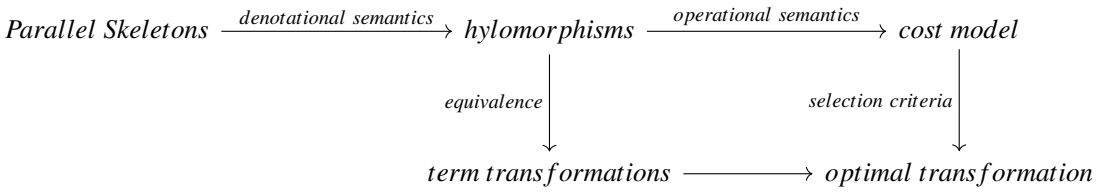


Figure 3.9: Hylomorphisms as a unifying construct for the object-language of *Structured Arrows*.

In chapter 4 which immediately follows we will take a different route that involves the use of Hylomorphisms to describe the semantics of the *meta-language* for the TyTra compiler framework rather than the object-level semantics of TyTra applications, as briefly summarized in Figure 3.10.

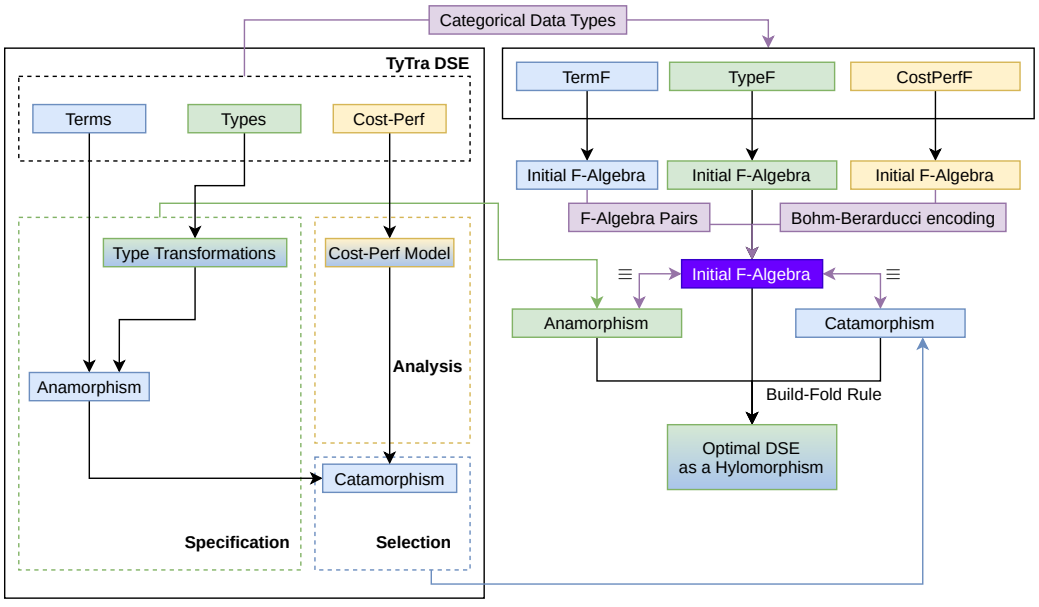


Figure 3.10: Hylomorphisms as a unifying construct for the meta-language of the TyTra Compiler Framework.

Chapter 4

Categorical Semantics

In this chapter present our main contributions.

- In section 4.1 we will define our *models* in terms of *category theoretical notions*, for the TyTra CL, its type system, and the *cost-performance* estimates. We thus extend the approach of defining *categorical data types*, covered in subsection 3.2.2 to handle the particular flavour of search problem we are faced with. These models give us an *abstract view* of what TyTra CL applications *are* and how they *behave*.
- In section 4.2 we will give a high-level view of DSE structured around the three conceptual stages we presented in section 4.1, namely: *specification*, *analysis* and *selection*. In this high-level presentation we will see what is required to design a DSE strategy for a simple example problem, exploring a design space of *cuboid shapes*. This will allow us to identify the evaluation morphisms required to implement a DSE strategy, on the categorical data types we build in section 4.1 without going into the more complex details of the *TyTra CL* and the *TyTra IR*.
- In section 4.3 we give a lower-level view of DSE as defined for the TyTra compiler, to show how the required DSE evaluation morphisms identified in section 4.2 relate to the problem of optimizing TyTra CL applications. This will allow us to connect the *programming language theoretical* notions of program optimization used in TyTra to the evaluation morphisms and categorical data types presented in the previous two sections.
- In section 4.4 we will use the categorical data type structures defined in section 4.1 to define *Categorical Semantics* for the evaluation morphisms identified as required in section 4.1, for the TyTra compiler. This will enable us to *relate* TyTra CL applications to their *globally optimal transformations*.
- Finally, in section 4.5 we formally state and prove our main theorem, namely that a correct, polynomial-time DSE strategy exists for the TyTra compiler, through the use of the *categorical semantics* presented in this chapter.

4.1 Categorical Data Types

In this first section we will briefly describe *categorical data type* (CDT) construction by building a data type for *lists of values*, roughly in line with the presentation of CDTs in *Foundations of Parallel Programming* [Ski05] we mentioned in subsection 3.2.2. This will help in explaining the utility of the category theoretical concepts covered in section 2.5 and give us the tools to identify the exact construction of *categorical data types* for TyTra *terms*, *types* and *cost-performance estimates* in section 2.5. The first step in categorical data type construction is the identification of a category and an *Endofunctor* on it, which models the properties we seek [Ski05]. In Haskell, a homogeneous list of values having the type a, is modelled as a data type $ListF\ a$ with two data constructors: $Empty : ListF\ a$ and $ConsF : a \rightarrow ListF\ a \rightarrow ListF\ a$ as shown in Listing 4.1.

```
data ListF a = EmptyF | ConsF a (ListF a)
```

Listing 4.1: Haskell data type that encodes Lists.

This list data type allows us to define operations on lists of values, in terms of operations defined for the values stored within. Given a function $f : a \rightarrow b$ we can derive a function $map : List\ a \rightarrow List\ b$, shown in Listing 4.2.

```
map :: (a -> b) -> ListF a -> ListF b
map f EmptyF = EmptyF
map f (ConsF h t) = ConsF (f h) (map f t)
```

Listing 4.2: Haskell implementation for the map operation.

$ListF\ a$ is parametrized by a type argument a meaning that for every object a in a category of *types*, $ListF\ a$ is also an object in that category. If every type a is the object in a category of types, then $ListF\ a$ identifies another object in that category, meaning that it too is a *type*. At the same time, the *map* function in Listing 4.2 identifies a morphism $f : a \rightarrow b$ with a morphism between the objects constructed using $ListF$.

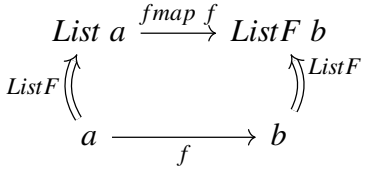


Figure 4.1: $ListF : 1 + a \times List\ a$ describes an Endofunctor.

We know that a *Functor* maps the objects and morphisms of one category to the objects and morphisms in another category. If the source and destination categories coincide, then it is called an *Endofunctor*. Taken together, the pair of $ListF$ and $fmap$ forms an *Endofunctor* on the category of types. For convenience, we can simply overload the name $ListF$ to denote both.

Suppose we made the choice of a explicit by saying that the category of types includes Integers, meaning that $a = Integer$. We can see that for any function $f : Integer \to Integer$ the *map* operation gives us a List Homomorphism, a structure preserving morphism on lists, in this case of Integers.

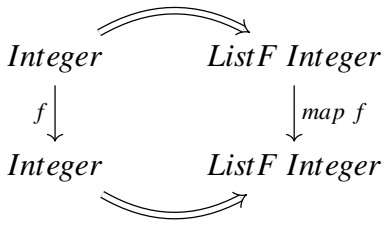


Figure 4.2: Category of Integers and List of Integers.

If both *Integer* and *List Integer* are valid type objects, we may also want to compute functions that take a *List Integer* input and produce a single *Integer* value. Let us denote one such morphism $eval_{Int}$ as shown in Figure 4.3.

$$ListF\ Integer \xrightarrow{eval_{Int}} Integer$$

Figure 4.3: F-Algebra on a list of Integers.

We see that the type signature for functions such as $eval_{Int}$ is the same as that of an *F-Algebra*, a concept we came across in section 2.5. For a given functor F that defines a data-type, and a carrier type x an F-algebra is identified with an evaluation morphism $eval_{F_x} : Fx \to x$. The type signature for an *F-Algebra* can be encoded in Haskell simply as $type\ Algebra\ f\ a = f\ a \to a$. *F-Algebras* evaluate a structure of x values: Fx by pattern matching on the data constructors that make up the functor F and replacing them with operations on the carrier type x. In this example, the functor in question is *ListF* and the carrier object is *Integer*. Suppose that the evaluation we wanted to compute was the sum of all *Integer* values in the list, there are then only two data-constructors to pattern match on, *EmptyF* and *ConsF*. These could be replaced with the 0 literal and the $(+)$ operation on Integers, respectively however there is first a small problem to be addressed. All-though the first parameter to *Cons* is of type *Integer*, the second parameter is of type *ListF a*. We know that the type for the sum operation is $(+) : Integer \to Integer \to Integer$ rather than $(+) : Integer \to ListF\ Integer \to Integer$. One way of solving this issue is to define the *F-Algebra* as the recursive function shown in Listing 4.7.

```
sumAlg :: ListF Integer -> Integer
sumAlg Empty = 0
sumAlg (ConsF h t) = h + (sumAlg t)
```

Listing 4.3: A List F-algebras that summs the contents.

A second and better option is to treat *recursion* as a separate issue altogether. This can be done by defining the list data type as the *fixpoint* of the Endofunctor. In Haskell, the fixpoint of a functor is obtained by providing the type-level equality, shown in Listing 4.4. This simply states that for any functor F the data-type defined by $Fix\ F$ is equal to an expression built using f that contains sub-terms of type $Fix\ F$.

```
newtype Fix f = Fix {unFix :: f (Fix f)}
```

Listing 4.4: Fixpoint definition (Haskell).

From the Haskell *type-level equality* in Listing 4.4 we get the two *type-level* functions that witness it, namely $Fix : f\ (Fix\ f) -> Fix\ f$ and $unFix : Fix\ f \rightarrow f\ (Fix\ f)$. Using the first, we can take our *ListF* Endofunctor, replace the recursive part of its definition with a new type parameter r, and then define a recursive list data type definition as the fixpoint of this new functor.

```
data ListF a r =
 EmptyF
 | ConsF a r
 deriving Functor

type List a = Fix (ListF a)
```

Listing 4.5: A List type is the fixpoint of ListF.

We can now redefine our *sumAlg* evaluation morphism as below.

```
sumAlg :: List Integer -> Integer
sumAlg (Fix EmptyF) = 0
sumAlg (Fix (ConsF h t)) = h + t
```

Listing 4.6: A List F-algebras that summs the contents.

We can now easily define other evaluation morphisms by simply choosing appropriate values to replace the list data constructors with. Replacing an empty list constructor with the value 1 and the second constructor with the product operation on integers, we obtain an *F-Algebra* that computes the product of all integers contained in a list.

```
prodAlg :: List Integer -> Integer
prodAlg (Fix EmptyF) = 0
prodAlg (Fix (ConsF h t)) = h * t
```

Listing 4.7: A List F-algebras that multiplies the contents.

Note that all such evaluation morphisms are in fact related in a very specific way. The fixpoint of the *ListF* functor is the *initial object* in the category of *F-Algebras* where *sumAlg* and *prodAlg* are also objects.

In section 2.5 an *F-Algebra* in a category **C** was defined as a three component tuple $(F, x, \alpha : F(x) \to x)$. Such *F-Algebra* triplets are primarily identified by their third component, the evaluation morphism, because each such morphism implies an exact choice of functor F and carrier type x. The *F-Algebras* for a given functor F that specify a data type form a category in which the *initial object* is the *initial F-Algebra* which defines that type, $Fix\ F$. The objects in this category represent every possible way of interpreting a value of type $Fix\ F$.

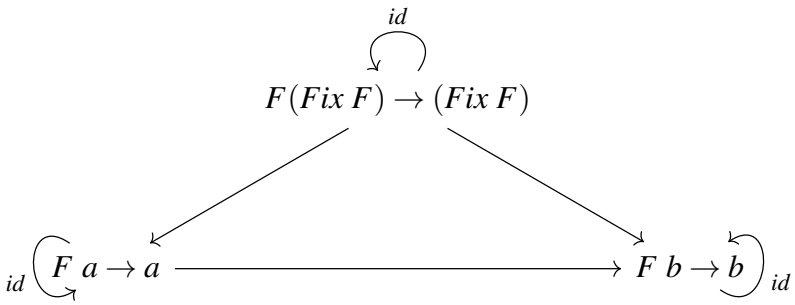


Figure 4.4: Objects and morphisms in the category of F-Algebras over a functor F.

The object at the top of Figure 4.1 is the *initial object* in the category of *F-Algebras*. Note that its type signature is given by the $unFix : Fix\ f \to f\ (Fix\ f)$ function. If we look closely at the definitions of *sumAlg* and *prodAlg* in Listing 4.6 and Listing 4.7 respectively, we see that both of them pattern-match on the outer *Fix* constructor of values having the type $Fix\ ListF$ to reveal the inner $ListF\,(Fix\ ListF)$ data structure. The effect of this pattern matching is the same as first calling *unFix* and then pattern-matching on the inner *ListF* value. This is important because the morphisms in Figure 4.1 describe a way of transforming one evaluation function into another. The arrows pointing away from the initial object towards $F\ a \to a$ and $F\ b \to b$ are exactly such morphisms. They are a transformation from operations represented as $Fix\ F$ values, to operations on a and b values respectively. Recall that we defined the list data type as the fixpoint of the *ListF* functor, meaning that the values having a $List\ a$ type are constructed using the same *Fix* type-level function as below.

```
empty :: List a
empty = Fix EmptyF

cons :: a -> List a -> List a
cons h t = Fix (ConsF h t)
```

Listing 4.8: *List a* data constructor functions.

It may seem as if we have put in a lot of effort to define a simple list data type. The value of *categorical data types* becomes apparent when we consider data types with more structure.

4.1.1 Optimisation from recursion schemes

So far we have seen that *F-algebras* can be used to evaluate a structure of values by replacing the constructors that define said structure with the operations of a carrier type. Likewise, an *F-CoAlgebra* can create structure from an initial seed value. If we are given or perhaps required to create a **nested** structure of values, such as the list of integer lists shown in Figure 4.5, then *F-Algebras* and *F-CoAlgebras* may not be sufficiently strong as to fully describe the required operation.

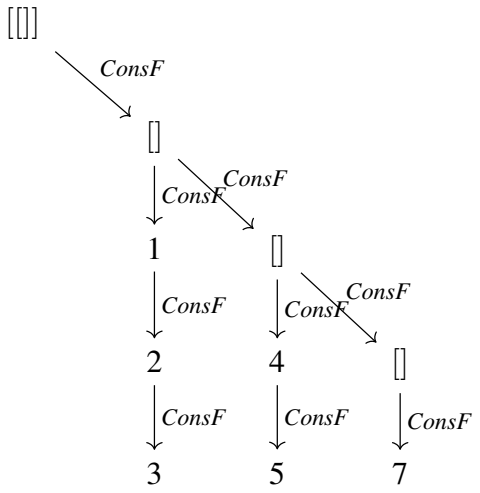


Figure 4.5: A term-level structure of nested list functors.

Producing an aggregate value, such as the sum of all numbers in the nested data-structure in Figure 4.5 implies a call to the evaluation function for each nested layer. Here one call produces the aggregate of the *inner list*, denoted by the vertical arrows standing in for the *ConsF* constructor, whilst another call aggregates the *outer list* denoted by the black diagonal lines. In the case of both *sumAlg* and *prodAlg*, the order of evaluation is not significant because both the $(+)$ and $(*)$ operators used to define them are associative and commutative on the *Integer* type. Had we chosen a different carrier type however, perhaps a matrix of values, then order of evaluation would indeed be significant. *Recursion schemes* are higher-order functions that specify an exact order of evaluation. They are parametrized by *F-Algebras* and *F-CoAlgebras*. One particularly useful recursion scheme is a *Catamorphism*, more commonly known as a *fold* operation. Another useful recursion scheme, is the *Anamorphism*, which is the opposite concept to that of a Catamorphism. The Haskell type signature for both recursion schemes is shown in Listing 4.9.

```
cata :: Functor f => (f a -> a) -> f a -> a
ana :: Functor f => (a -> f a) -> a -> f a
```

Listing 4.9: Haskell type signatures for Catamorphisms/Anamorphisms.

Catamorphisms correspond to traversals of nested data-structures. The *F-Algebra* supplied as the first argument is iteratively applied to every intermediary data structure. The exact order of evaluation chosen to implement the Catamorphism can have an effect on the overall performance. Processing certain nested data-structures may be quicker if it's done top-to-bottom, whilst others may be faster done bottom-up. The optimizing power of Catamorphisms, particularly in the case of *list operations* is discussed at length in *Foundations of Parallel Programming* [Ski05], however, Catamorphisms alone are also insufficient for our purposes. Conversely, an *Anamorphism* parametrized by some *F-CoAlgebra* can be used to recursively build a nested data-structure. The first application of the *F-CoAlgebra* generates a structure of new seed values on which the *F-CoAlgebra* can be recursively applied. Given that recursion schemes are *higher-order functions* and knowing that functions means that simple recursion schemes such as *cata* and *ana* can be used to define other, more complicated recursion schemes. A *Hylomorphism*, for example, is defined as the composition of an *Anamorphism* with a *Catamorphism*, as shown in Listing 4.10.

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> a -> b
hylo f g = h where h = f . fmap h . g
```

Listing 4.10: Hylomorphism signature in Haskell.

The composition of two recursion schemes can also be more efficient than the sequential application of its components. A Hylomorphism may be seen as altering the *Anamorphism* component to never produces those structures removed by the *Catamorphism*. The removal of intermediate representations as an optimization strategy is known as automatic deforestation [Wad88]. In the TyTra compiler, the process of generating candidate program variants is effectively an *Anamorphism*. The seed value in this case is the TyTra CL expression that was provided as the input to the compiler. This *initial program variant* is used to generate a search-space of optimized variants. Likewise, selecting the most performant and least costly program variant is a *Catamorphism*. The structure of the search-space is consumed until all that is left is a single *globally optimal* program variant. At the same time, the data types for TyTra terms, types and cost-performance estimates are all defined as the *Anamorphisms* and *Catamorphisms* provided by the *Fix* and *unFix* type-level functions shown in Listing 4.4. We can see now that our goal is to describe the entire *design space exploration* process as a composite recursion scheme that provides an optimal evaluation order. Doing so requires careful planning as the data structures we must work with are significantly more complicated than a simple or nested list. The recursion scheme we are looking for is parametrized by different *F-Algebras* and *F-CoAlgebras* and relies on multiple functors including: *TermF*, *TypeF* and the *cost-performance* base functors we will construct next, in subsection 4.1.2.

4.1.2 Terms and Transformations

Before we can specify *Categorical Semantics* for the TyTra compiler framework, we must first define *categorical data-types* for the structures on which it operates. We will begin by analysing the TyTra CL term language implementation in the Haskell programming language.

```
data TermI = TermVar String
  | TermApp TermI TermI
  | TermTup [TermI]
  | TermZip
  | TermUnZip
  | TermStencil [Integer]
  | TermMap TermI
  | TermFold TermI
  | TermElt Integer
```

Listing 4.11: Haskell data type for TyTra CL Terms.

The Haskell representation for the data type in Listing 4.11 defines an *Abstract Syntax Tree* for TyTra CL expressions. We can see that the *TermI* data type is constructed by listing out a number of *data constructors*, each having an unique name. The *TermVar* data constructor, for example, can be used to represent *input variables* and *opaque functions* as both of these are entirely identified by their names, encoded as simple *String* values. From a categorical data type perspective, *data constructors* can be seen as functors. Data constructors such as *TermVar* map the objects and morphisms of one category, in this case the category of *String literals*, into the category of *terms*. Recursive data constructors are then *Endofunctors* because their source and destination categories coincide.

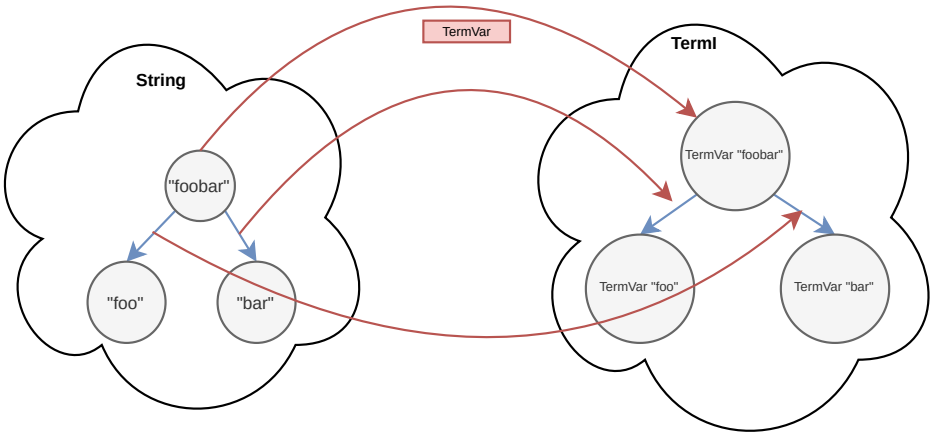


Figure 4.6: The *TermVar* data constructor is a functor from the category of Strings to the category of TyTra CL Terms

TyTra CL AST transformations are functions that take and return *term* values meaning they have the type shown in Figure 4.1.2 below.

$$TermI \longrightarrow TermI$$

Figure 4.7: The Type of TyTra CL AST Transformations.

Just as the list data type can be interpreted as the fixpoint of the *ListF* functor, so can the terms of the TyTra CL. The Haskell data type encoding of TyTra CL terms we showed in Listing 4.11 can just as well be defined as *Fix TermF*, the fixpoint of *TermF*, a functor, shown below.

```
data TermF a = TermVarF String
| TermAppF a a
| TermTupF [a]
| TermZipF
| TermUnZipF
| TermStencilF [Integer]
| TermMapF a
| TermFoldF a
| TermEltF Integer
deriving ( Functor )

newtype Term = Fix TermF
```

Listing 4.12: Haskell Functor that defines the structure of TyTra CL Terms.

The *sumAlg* and *prodAlg* evaluations of lists of integers were relatively straight-forward Catamorphisms over the *ListF* functor. In contrast to those, the evaluations of TyTra CL terms is significantly more complicated. On the one hand this is due to the richer structure of the *TermF Endofunctor*. On the other hand, a correct and efficient DSE strategy requires that we define transformations on TyTra CL expressions, in terms of their evaluation as TyTra CL *types* and *cost-performance estimates* which we have yet to define. At the *term-level* we have no way of guaranteeing that the transformation process does not alter the meaning of the application. If we represented the input and output TyTra CL AST nodes using their Functor type format, we would obtain a simple function signature more like:

$$TermF\ TermI \longrightarrow TermF\ TermI$$

Figure 4.8: Functorial Type TyTra CL AST Transformations

The *type for the argument* passed to this function *exposes* the outer-most data-constructor of whatever TyTra CL expression is passed in. Any evaluation function that takes arguments of this type are consequently limited. They can only inspect this outer-most constructor.

We can implement simple transformations such as the *identity transformation*, which performs no work, and corresponds to the *identity morphism* on *TermF TermI* objects. One possible (but purposely incorrect) implementation is shown in Listing 4.13.

```
idTerm :: TermI -> TermI
idTerm Termvar name  = TermVar name
idTerm TermApp f i   = TermApp f i
idTerm TermTup ts  = TermTup ts
idTerm TermZip   = TermZip
idTerm TermUnZip   = TermUnZip
idTerm TermStencil ixs  = TermStencil ixs
idTerm TermElt ix    = TermElt ix
idTerm TermMap f  = TermFold f   -- Incorrect behaviour
idTerm TermFold f   = TermMap f -- Incorrect behaviour
```

Listing 4.13: Incorrect identity transformation on TyTra CL terms.

Specifying term-level transformations for our *TermI* data type is a straight-forward but potentially error-prone process. The *case expression* in Listing 4.13 *pattern-matches* a data constructor on the left-hand-side of the arrow operator and then replaces it with another expression given on the right-hand-side. That being said we can easily see that this *idTerm* implementation is incorrect because it replaces a *TermMap* data constructor with a *TermFold* constructor, and vice-versa. An *abstract data type*, on the other hand, can be **entirely defined by the operation it supports** without having to also specify a *concrete term representation*.

Suppose we have already defined a category for the TyTra type system where the objects are *types* and the morphisms between these objects define *type-level transformations*. Given any *term-level* representation constructed using the *TermI* data-type, denoted as the *term* object below, a type-evaluation function *typeEval* would be the functor that maps the *term* object and its morphisms, to an object *t* and its morphisms, in the category of *types* as shown in Figure 4.9.

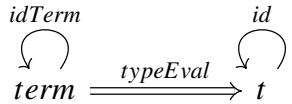


Figure 4.9: Identity morphisms on TyTra CL types

Using *typeEval* we can determine the correct *identity transformation on terms* by seeing which morphism on terms it maps to the *identity morphism on types*. Next in subsection 4.1.3 we will define this category of type objects which we presumed to exist.

4.1.3 Types and Type Constructors

The TyTra CL type system is defined *inductively* meaning that more complicated types are constructed by applying type-level constructors to already defined types. If every type is an object in the category of types, then type-level constructors are *Endofunctors* on the category of types. The TyTra CL term-level constructors specified by the *TermI* data type each identify not one but an entire *family* of Endofunctors. This is because the exact or *concrete* set of types that correspond to any given TyTra CL term is not just a function of the term's structure, but also a function of the *type context* or *set of type annotations* that are valid for the TyTra CL term in question. Let us now have a closer look at the individual data constructors and how they determine relations on types. At the beginning of subsection 4.1.2 we said that the *TermVar* data constructor is a functor from the category of String literals to the category of TyTra CL Terms. The term object identified by the *TermVar* functor and a given String literal value *"foo"* simply denoted *TermVar foo*, is mapped by the *typeEval* functor to a *family of Endofunctors* $Type_\Gamma : Type \to Type$ on the category of TyTra CL Types.

This family is indexed by type contexts meaning that for a given Γ type context, the product object (Γ , *TermVar foo*) identifies a specific $Type_{(\Gamma,foo)} : Type \to Type$ Endofunctor such that $\Gamma \vdash TermVar\ foo : Type_{(\Gamma,foo)}\ ()$ where $()$ is the initial object in the category of TyTra CL Types. Because $Type_{(\Gamma,foo)}$ is an Endofunctor, meaning that its source and destination categories coincide. It can just as well be understood as a *morphism between type objects*, or a function on types. If the *"foo"* variable name is given a concrete type definition in some context, perhaps $\Gamma \vdash foo : Int$, then the $Type_{(\Gamma,foo)}\ ()$ identifies the type-level constant function $() \to Int$, more simply known as the *Int* type object. The same reasoning can be applied to all other term constructors, as shown through the type derivation rules below. Above the line we specify the type objects assigned through Γ to the sub-terms required by each such constructor. Below the line we see the constructed term and the type object assigned to it by the *typeEval* functor.

$$\frac{\Gamma \vdash f : Type_a \mapsto Type_b \qquad \Gamma \vdash i : Type_a}{\text{(TermApp f a)} : Type_b} \qquad \frac{\Gamma \vdash \text{f} : a \to b}{\text{TermMap f} : Vec_s a \to Vec_s b}$$

$$\frac{\Gamma \vdash \text{a} : Type_a \qquad \ldots \qquad \Gamma \vdash \text{z} : Type_z}{\text{TermTup(a, \ldots, z)} : (Term_a, \ldots, Type_z)} \qquad \frac{\Gamma \vdash \text{f} : a \to b}{\text{TermFold f} : Vec_s a \to b}$$

$$\frac{\Gamma \vdash \text{a} : Vec_s a \qquad \Gamma \vdash \text{b} : Vec_s b}{\text{TermApp TermZip (a, b)} : Vec_s(a,b)} \qquad \frac{\Gamma \vdash \text{i} : Vec_s(a,b)}{\text{TermApp TermZip i} : (Vec_s a, Vec_s b)}$$

$$\frac{\Gamma \vdash \text{i} : Vec_s a}{\text{TermApp (TermStencil [}ix_1, \ldots, ix_n\text{]) i} : SVec_n a} \qquad \frac{\Gamma \vdash \text{v} : Vec_s a}{\text{TermApp (TermElt } ix\text{) v} : a}$$

Figure 4.10: TyTra CL Type System derivation rules.

The type constructors used to form the output of the *typeEval* functor describe the Haskell data type shown in Listing 4.14 below.

```
data TypeI =
| TyAtom String
| TyProd TypeI TypeI
| TyVec Size TypeI
| TySVec Size TypeI
| TyTup [TypeI]
| TyFun TypeI TypeI
```

Listing 4.14: Haskell data type that encodes TyTra CL Types.

We will now unpack this definition to identify the categorical notions that are needed to princely describe the TyTra type system.

Atomic Types are unique objects

The category of String literals, or names, is depicted at the bottom of Figure 4.11 by showing an object for every possible string literal. A given TyTra CL application may only use a finite number of these names, each identifying a unique object in the category of TyTra CL Types, such as the $Type_{Aa}$ and $Type_{Zz}$ *Atomic Types* shown at the top of Figure 4.11.

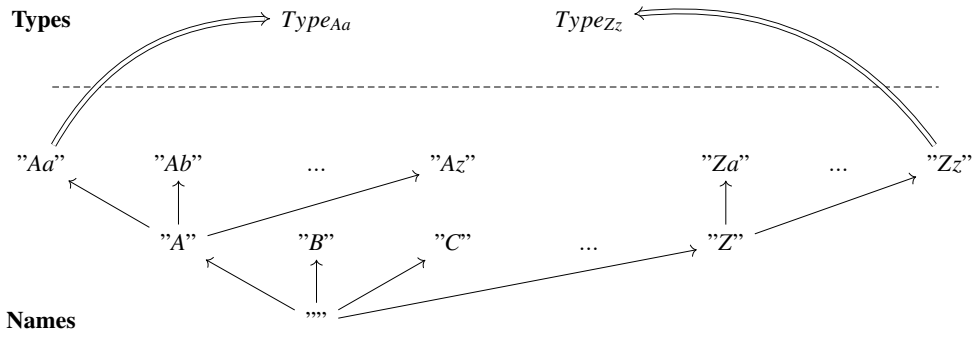


Figure 4.11: Atomic Types as objects of a category (top) and the category of names (bottom).

The *Atomic Type constructor*, depicted with a double arrow in Figure 4.11, is the *Functor* that assigns each name a particular *Atomic Type*. Note the similarity between this functor and that used to define term-level variables, depicted in Figure 4.6. As with opaque functions, a String representation for a unique type name is all that is required. The concrete properties pertaining to individual atomic types, that may be required during DSE, can be recovered from the context of cost-performance estimates through a lookup operation that takes this String representation as an argument, as we will see in Listing 4.49.

Tuples Types are Product Objects

Given two types a and b, the tuple type (a,b) denotes an aggregate value that contains a value of type a and a value of type b. If a and b are the objects of a category, then the tuple type (a,b) is the *product object of a and b* in that category. Every binary type (a,b) comes with two *projection functions* that may be used to access the component values. In the Haskell programming language, these projections are the frequently used $fst : (a,b) \to a$ and $snd : (a,b) \to b$ functions, depicted in Figure 4.1.3 below.

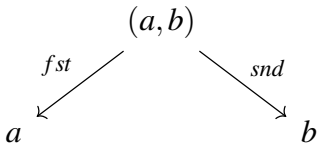


Figure 4.12: Tuple Type projections are morphisms in the category of TyTra CL Types.

A *Binary Tuple Type Constructor* must then be the *Endofunctor* that defines the following structure-preserving map: $Tup_{Ob} : a \times b \to Tup(a,b)$. We have also previously mentioned that binary tuples are sufficient to model $k-ary$ tuples. An explanation follows in the subsequent discussion of Vector Types.

Vector Types Constructors are Indexed Endofunctors

Vector Type Constructors correspond to *homogenous* tuples, for which all projections point to the same destination object. The projection functions from a *Vector Type* may yield different *values* yet all of these will have the same type and can thus be denoted by the same object in our category of types. We say that *Vector Types* in TyTra CL are **indexed** by their size and **parametrized** by a type parameter.

$$Vec_2 a = Tup(a,a)$$

Figure 4.13: A *Vector Type* of size 2 corresponds to a binary tuple $Tup(a,a)$.

Recall that *Functors* (including *Endofunctors* are morphisms in **Cat**, the category of categories. There may exist *multiple morphisms* between the objects of a language, but they are all *isomorphic*. A *Vector Type Constructor*, being an *Endofunctor* in the category of TyTra CL Types, is then a morphism in **Cat**. The meaning of the *size index* that distinguishes $Vec_k\, a$ from $Vec_m\, a$, is give by the concept of an *Indexed Functor*. For our purposes it is sufficient to state that parametrizing the *Vector Type Constructor* by a type parameter a yields a **Family of Type Constructors** $Vec_i\, a$ where a *particular type constructor* may be selected by indexing with i, the size index. Setting the index i to a value of 2 for a vector parametrized by the a atomic type, as in Figure 4.13, yields the *Binary Tuple Constructor* (parametrized in both places with a) as the particular Endofunctor from the family that defines *Vector Types*.

The Category of Types is Distributive

Vector Data-Types have two *data constructors*. A type having a *choice* of two or more *type constructors* is known as a *sum type*. The first $VNil$ yields the empty vector value, having a type that is indexed with the size value 0. The second data constructor, $VCons$, takes two parameters: type a by which the vector type is parametrized, and another vector to which the value is prepended.

$$Vec_{k+1}a : 1 + a \times (Vec_k\, a)$$

Figure 4.14: Vec is a polynomial functor.

Here the *output size index* is implicitly defined as the *successor* $(+1)$ of the *input size index*. The fact that the *output type* depends on an *input value* shows that the TyTra CL type system is *dependently typed*. The category of TyTra CL types is a distributive category. The higher-order operations: **ZipT** and **UnZipT** witness the distributive law.

$$(1 + a \times Vec_a) \times (1 + b \times Vec_b) = 1 + (a \times b) \times Vec_{a \times b} \tag{4.1}$$

Reading Equation 4.1 from left-to-right yields the *ZipT* operation. Conversely, reading from right-to-left gives the opposite operation, *UnZipT*. The distributive property of *products over sums* witnessed by these operations is useful in proving what that *encoding all k-ary tuples as binary tuple types leads to no loss of generality*. We now show the proof in graphical form which defines *Vector Types* are homogenous tuples.

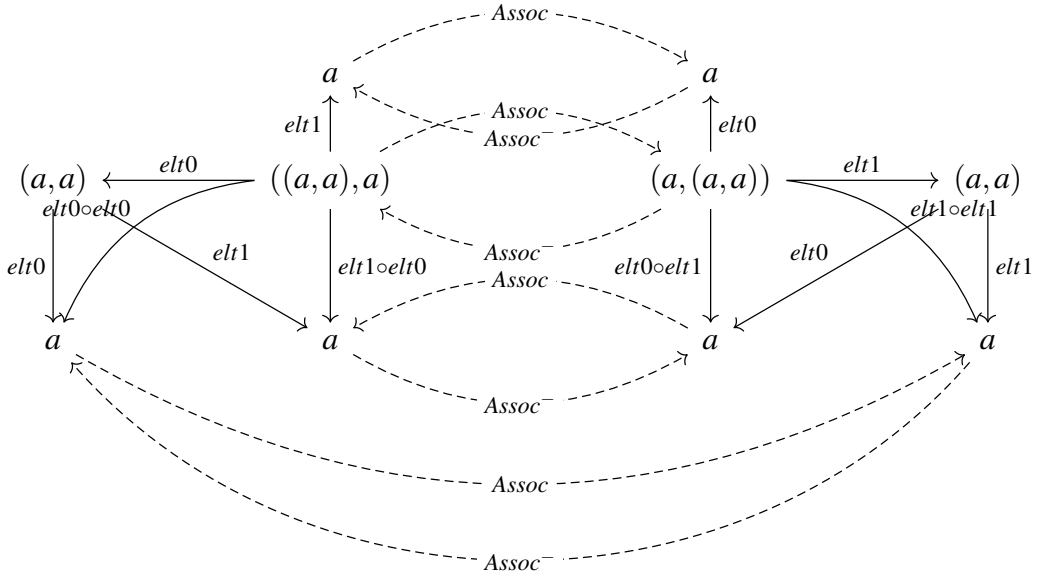


Figure 4.15: Distributivity of products over sums.

The arrows between each pair of corresponding objects on the left and right, labelled *Assoc* and $Assoc^{-}$ in the diagram are *Endofunctors*. They are structure preserving maps from the category of TyTra CL Types into itself. Both map objects to objects and morphisms to morphisms whilst preserving the sources, destinations, composition of morphisms, and associativity of composition for all morphisms.

Assoc and $Assoc^{-}$ are each-other's inverse, together they form a *natural isomorphism*. If we couple this notion with a *tensor product* operator, in our case the binary tuple constructor, as well as an *unit object* we can show that the category of types is a *monoidal category*. The *object mapping* part of the *Assoc* and $Assoc^{-}$ Endofunctors are morphisms in the category. Together they witness isomorphism between the source and destination objects.

If we have another look at the *Assoc* , $Assoc^{-}$ functors, we will see they are *adjoint* Endofunctors. They identify the isomorphisms between *k*-ary *Tuple Types*, or equivalently, *Vector Types* of size *k* functors. We can likewise define the meaning of the **Elt action** by defining a pair of adjoint functors between the category of natural numbers $\mathbb{N}$ and that part of the category corresponding to *Vector Types*.

Function Types are Exponential Objects

The TyTra type system also has function types. The categorical object that defines function types is the *exponential object*, also called an *internal hom-set* because it represents the set of all morphisms between two other objects in the category. Recall that an exponential object x^y comes with an evaluation morphism $apply : x^y \times y \to x$. An exponential object x^y that represents the hom-set of all morphisms f such that $f : y \to x$ would be represented as $TyFun\ y\ x$, in accordance to the data type of TyTra CL types we have shown in Listing 4.14. The *typeEval* functor applied to a *TermApp* expression identifies this $apply$ evaluation morphism according to the commutative diagram shown in Figure 4.16 below.

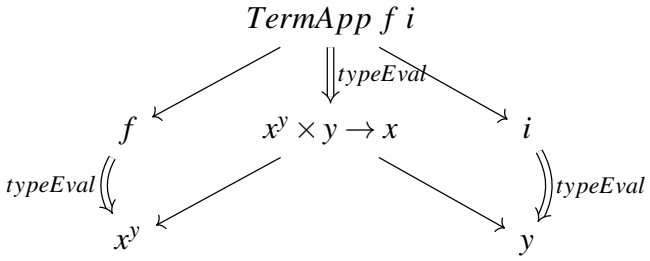


Figure 4.16: Function types.

4.1.4 Cost-Performance Estimates

The data type that defines cost-performance estimates is simply a tuple of tuples. The outer tuple has a *performance estimate* component and a *resource use* component. Each of these, in turn, is a tuple of measurements or estimates. The values in these inner tuples are *natural numbers* having the type *Nat* as shown in Listing 4.15.

```
data PerformanceKI = PerformanceKI {
 efi  :: Nat, afi  :: Nat , lat  :: Nat, sd  :: Nat,  lfi  :: Nat, fpo :: Nat
}
data ResourceKI = ResourceKI {
 regs :: Nat , bram :: Nat, dsps :: Nat , aluts  :: Nat, propDelay :: Nat
}
data CostPerf = CostPerf PerformanceKI ResourceKI
```

Listing 4.15: Haskell data type for cost-performance estimates.

Natural numbers can be represented as Peano numerals. This means defining natural numbers as having two data constructors. The first is a nullary constructor $Zero :: Nat$ corresponding to the value 0. The second is $Successor :: Nat \rightarrow Nat$, a recursive data-constructor.

```
data Nat = Zero | Successor Nat
```

Listing 4.16: The type of natural numbers (Haskell).

The corresponding functor is *NatF* such that $Nat = Fix\ NatF$.

```
data NatF a = ZeroF | SuccessorF a
```

Listing 4.17: Functor of natural numbers (Haskell).

From a category theory perspective, *ZeroF* is the initial object in the category of natural numbers **Nat**. Every natural number, apart from 0, is defined as a sequence of applications of the *SuccessorF* endofunctor to the initial object defined by *ZeroF*. The value 1 is defined as *SuccessorF ZeroF*, the value 2 corresponds to *SuccessorF (SuccessorF ZeroF)* and so on. The reason for our choice of representing performance and resource-cost estimates using Peano-encoded natural numbers becomes apparent when we recall the functor that defined list data types from Listing 4.1. There exists a very close relationship between these two functors. The *ZeroF* constructor can be mapped to *EmptyF*, the empty list constructor, whilst a sequence of k applications of the *SuccessorF* constructor corresponds to pattern-matching k applications of the *ConsF* constructor, in other words, selecting the k^{th} element of a potentially unbounded list.

Recall that the TyTra type system includes *size-indexed* vector types, rather than unbounded lists. From the Peano encoding of natural numbers we may obtain a new type that describes *bounded* or finite natural numbers up to a selected value k. For any selected k value we define a functor from **Nat** into the opposite category Fin_k that maps a k, into the *initial object*. Given the duality between the initial object in one category and the terminal object in the opposite category, the same functor maps *ZeroF* into the terminal object of Fin_k. Suppose we chose a concrete value $k = 3$. Using the method just described, we can define a data type of finite natural numbers up to the value of three, as shown in Figure 4.17.

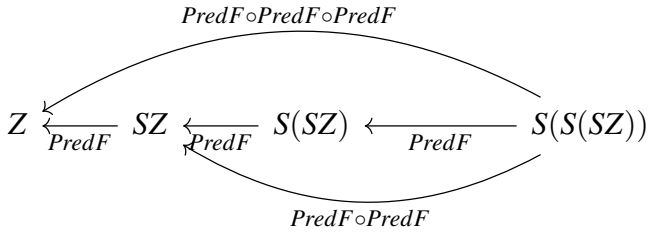


Figure 4.17: Fin_3: the category of finite natural numbers up to the value 3.

The *successor* morphism in **Nat** defined by the *SuccessorF* Endofunctor on **Nat** is mapped into the *predecessor* morphism in Fin_3 which corresponds to the *PredF* Endofunctor. The choice of a concrete value $k = 3$ means that Fin_3 has four objects in total, described by the types *Three*, *Two*, *One* and *Zero* shown in Listing 4.18. Where the **Nat** category was related to lists, a Fin_k category is related to a finite vector type Vec_k.

```
data PredF a = PredF a
data Three = Three
newtype Two = PredF Three
newtype One = PredF (PredF Three)
newtype Zero = PredF (PredF (PredF Three))
```

Listing 4.18: Finite Numbers.

The relationship between Fin_3 and a TyTra vector type $Vec_3\ a$ becomes apparent when we consider the element selection operation on vector types. We can produce exactly four values from a vector of type $Vec_3\ a$. Three of these values are of type a stored in the vector, whilst the fourth value denotes an empty vector type.

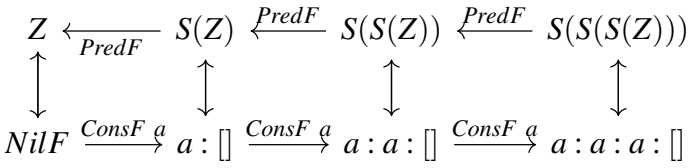


Figure 4.18: Relationship between Fin_3 and a vector type $Vec_3\ a$.

The relationship between Fin_3 and vector types $Vec_3\ a$ shown in Figure 4.18 gives rise to most of the optimizations to our DSE strategy. Consider the task of parallelizing a simple application where the only operation performed is a *map* over a vector of 5 integers. The type for this expression is $Vec_5\ Int \rightarrow Vec_5\ Int$. The *specification* or *search-space generation* phase of a DSE strategy would normally produce a list of program variants that parallelize this map operation. The input vector has a finite size of 5 meaning that at most 5 program variants from this potentially infinite list, can lead to worth-while implementations. The simple matter of fact is that no parallel FPGA implementation can speed up a mapping operation, where the kernel function has an opaque definition, above the limit imposed by data availability. This means that using the input vector's type size index, the value 5, we can generate a finite vector of program transformations with the same size index $Vec_5\ (Vec_5\ Int \rightarrow Vec_5\ Int)$. Furthermore, every program transformation in this vector is defined by its relation to the value 5 using the *PredF* functor. The fifth value in the vector might represent a parallelization factor of 5, the fourth value a factor of 4 and so on, until the terminal value 0 is reached. Because of this relation between the size index and the program transformations, we can also work out the maximum cost of any parallelization factor, for map operations, up to a size index k as shown in Listing 4.19.

```
smulRKI :: Nat −> ResourceKI −> ResourceKI
smulRKI k (ResourceKI ar ab ad aa apd) = ResourceKI (k*ar) (k*ab) (k*ad) (k*aa) apd
```

Listing 4.19: Scalar resource cost multiplication.

Note the type signature for *smulRKI* shown in Listing 4.19. The first argument is of type *Nat* rather than Fin_5 to make the Haskell implementation easier. This is neither a practical nor a theoretical issue as the corresponding Fin_5 value is simply the same object in the opposite category to **Nat**. The interpretation of this type signature as a function says that: given a parallelization factor k and a resource-cost estimate, we can work out the resource-cost estimate of a k-parallel implementation. At the same time, the *selection* phase of our DSE implementation removes program variants that would consume more hardware resources than the target FPGA can provide. Suppose we represented the hardware resource limits as a value $l : ResourceKI$. A *select* function on lists of program variants/transformations could be implemented as shown in Listing 4.20.

```
select :: ResourceKI −> [ResourceKI] −> [ResourceKI]
select l variants = case variants of
 [] −> []
 (x:xs) = if smulRKI (size xs + 1) x < l
    then x : select l xs
    else []
```

Listing 4.20: Resource cost based selection.

If the hardware resource limits imposed by l are less than any of the values contained in the *variants* input list, the output will be a list having a size index k_l, strictly smaller than k which is the size-index for the input vector that generated the list of *variants*. Otherwise, the output list will have exactly k values, meaning that in all cases $k_l \leq k$. Rather than generate and filter a list, we can simply use the relationship between k and a vector type Vec_k shown in Figure 4.18, alongside the relationship between the elements of a vector of parallel program variants generated for a *map* term and the total resource cost given by *smulRKI* shown in Listing 4.19, to equationally work out the value of k_l. Knowing that the *ResourceKI* data-type is a product of natural numbers, we can simply use integer division on every component of l against every projection of the *ResourceKI* estimation for the opaque function in question, and then take the minimum integer divisor found. If the opaque function has minimal hardware resource use estimates, we may find that $k \leq k_l$. In this we simply redefine $k_l = min\ k_l\ k$. to ensure that in the end, $k_l \leq k$.

```
addRKI :: ResourceKI −> ResourceKI −> ResourceKI
addRKI
 (ResourceKI ar ab ad aa apd)
 (ResourceKI br bb bd ba bpd)
 = ResourceKI (ar+br) (ab+bb) (ad+bd) (aa+ba) (apd + bpd)
```

Listing 4.21: Resource cost addition is defined point-wise.

Evidently, not all TyTra CL applications are as simple as a map operation over a single vector input. If a TyTra CL term t has two sub-terms, s_1 and s_t, then we need a way of working out the overall resource-cost of t from the costs of s_1 and s_2. This can be done using the *addRKI* function shown in Listing 4.21 which is defined as point-wise addition. The addition of natural numbers represented with a Peano encoding is given by the $S\,n + m = S\,(m + n)$ equation. The performance side of a *cost-performance* estimate means that the performance of t is simply the component-wise minimum of performance components of s_1 and s_2 as shown in Listing 4.22.

```
minPKI :: PerformanceKI −> PerformanceKI −> ResourceKI
minPKI
 (PerformanceKI aefi aafi  alat  asd afpo)
 (PerformanceKI befi bafi  blat  bsd bfpo)
 = ResourceKI (min aefi befi)  (min aafi  bafi )  [..]
```

Listing 4.22: Minimum of two performance estimates.

The overall performance of a term is defined as the minimum of its sub-terms because of the functional dependency of circuits, which is explained in subsection 4.3.2. In section 4.2 that immediately follows, we will see how the categorical data types and the category-theoretical semantics we have defined for TyTra can be used to implement our efficient DSE strategy.

4.2 DSE in Categorical Terms

The structure of the presentation in this section is based on the three conceptual DSE stages we presented in section 1.1, namely *specification*, *analysis* and *selection*. Our formal definition for a DSE process is given alongside its application to a *toy example*, a DSE strategy for the space of *cuboid* designs, the three-dimensional equivalent of a rectangle. This presentation will allow us to cover all required definitions without going into the complexities of the TyTra compiler and the two formal languages used therein: the *TyTra CL* and the *TyTra IR*. Deriving an efficient DSE strategy for the TyTra compiler hinges on our ability to bridge the semantic gap between the TyTra CL and the TyTra IR, this will be explained in subsection 4.1.2.

4.2.1 Specification

A design space, we said, may be seen as a *collection* of *design points*. If need only consider the *end result*, then we might represent every transformed program variant (or term) as the object in a category of programs, as categories are *collections of objects*. In Figure 4.19 below, we show these objects as grey circles. Morphisms in this category would then be equivalent to *program transformations*, and are shown using the black arrows connecting these objects.

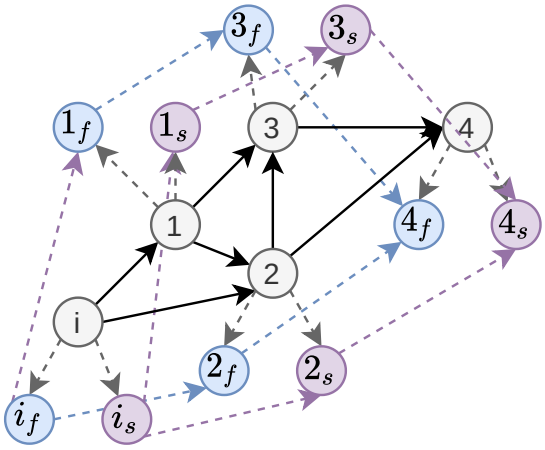


Figure 4.19: Sub-terms (blue/purple circles) as projections (grey arrows) in the category of terms, and their transformations (blue/purple morphisms).

We call the objects in this category *design points*. Each of these corresponds to one or more *design decisions*, meaning that each design point is effectively a *configuration* with multiple options being set. If the design point represents the term of a programming language, then *design decisions* might correspond to *sub-terms*.

Assuming that the terms of a language can be represented as the objects of a category with finite products, then the sub-terms can be identified with the *projection morphisms*, shown in blue and purple in Figure 4.19. It is easy to see that in such cases, there exists a correspondence between *term-transformations* and *sub-term transformations*: a pair of blue and purple arrows represents the same information as its corresponding black arrow. To illustrate why *category theory* is such a useful tool in optimizing a DSE strategy, as well as the limitations that apply to this technique, let us consider a more simple problem: optimizing design space exploration in a search space of *cuboids*. We will start from the basic specification of the cuboid as the three-dimensional equivalent of a rectangle, an example of which is depicted below in Figure 4.20.

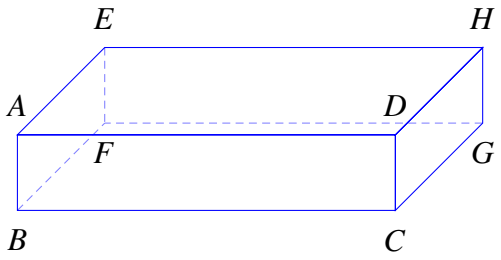


Figure 4.20: Design Space of cuboids.

If a *design point* represents a specific cuboid, which may itself be represented as a collection of *design decisions*, then we must choose a suitable representation for these *design decisions*. In the ideal case, the structure of a *design point* would map perfectly on to the structure of the object which we are attempting to optimize. For a cuboid, this might that a *design point* has one of the following definitions, depending on our choice of granularity, as a collection of:

- Corner **Point** objects, denoted A, B, C, D, E, F, G, H in Figure 4.20. The concrete collection might be a finite product of these points, the eight-tuple: (A, B, C, D, E, F, G, H).
- **Line** objects which are binary products of **Point** objects: $((A, B), (B, C) \ldots)$.
- Rectangles or **Face** objects, represented either by *products of Line* objects or *products of Point* objects, as defined above.

Reading these definitions in a top-down order, we see that they are progressively more structured. With increased structure comes *more safety*: an arbitrary set of eight points can denote many shapes, not just cuboids, yet a collection of rectangles is *closer* to the true meaning of a cuboid. Geometrically, we know that there is a set of relations that must hold true for the points, lines or rectangles that make up a cuboid. It is also true that these relations are *connected* across specification granularity boundaries by a higher-level of relation. Given two points that do not form a *valid line* (perhaps they coincide) we can not possibly form a *valid rectangle*.

This means that valid *design points* can only be constructed from, or decomposed into, *structured collections of design decisions* using *structure-preserving transformations*. The structure that must be preserved is defined by the *correctness criteria* for such designs. We already have a name for such structure-preserving transformations: they are *functors*. A **Point** object may itself be represented in the Haskell programming language as product of three natural numbers as shown in Listing 4.23.

```
data Point = MkPoint Nat Nat Nat
aPoint = MkPoint 0 0 0
bPoint = MkPoint 1 1 1
```

Listing 4.23: Point data-type definition.

The complete formal definition for a **Point** object relies on two notions. One is the *functor* pointing from the category of natural numbers, into the category of **Designs**, shown using the red arrows in Figure 4.21 below. The second is the *universal property* that defines products, shown using the blue arrows in the same figure.

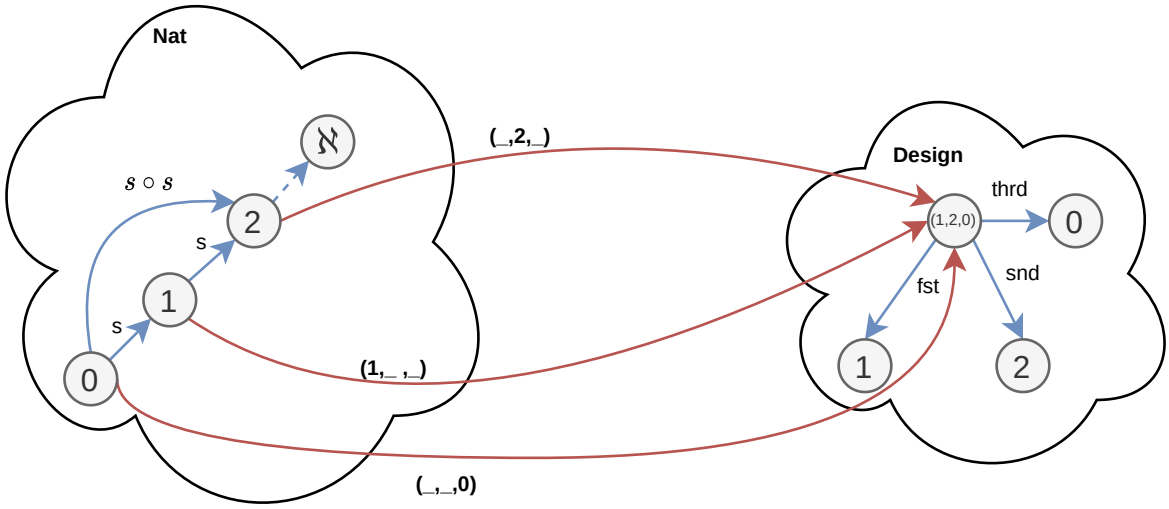


Figure 4.21: The category of *Point* objects.

Having defined the *Point* object using the *universal property* of a product object, we know that it is the **initial object** in that category. This means that every other object in the category can be identified through a morphism from that object. The universal property for product objects states that if products exist, then every other object in that category must be *factored* through that object. Here the other objects are the components values of a *Point*. They are *factored* through the product object using the projection operations: *fst*, *snd* and *thrd*. As we have suggested previously, using the notion of a *Point* alone is not enough to define what a valid cuboid is.

We would have to use an evaluation function, perhaps that shown in Listing 4.24 to determine whether a collection of points constitutes a *cuboid*.

```
isCuboid :: Design -> Bool
isCuboid a b c d e f g h =
length a b == length c d && length c d == length e f & length e f == length g h
&& length a d == length b c && length b c == length f g && length f g == length e h
 [...]
```

Listing 4.24: A predicate for correct cuboids.

Clearly, *isCuboid* defined in Listing 4.24, is cumbersome to use. Seeing that this function used the *length* property of a line to determine the validity of a cuboid, we might do well to represent the notion of a line in a new category, **Design'** where a **Line** is the initial object, defined as a product of two distinct **Point** objects.

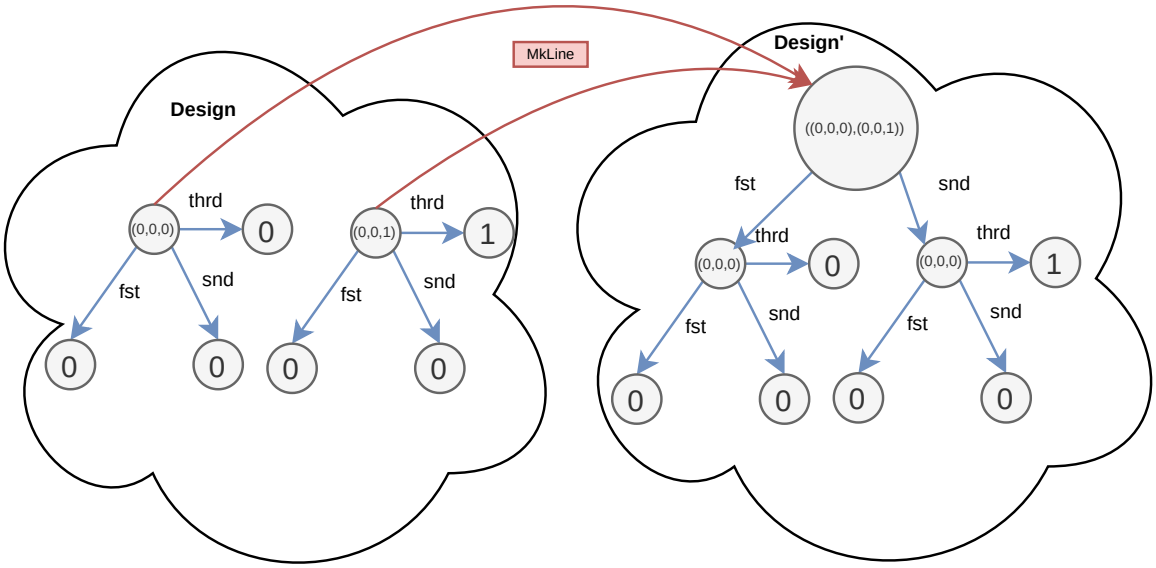


Figure 4.22: The category of *Design'* objects where *Line* is the initial object and its relation to *Design*.

By defining the constructor functor **MkLine** shown with the red arrows in Figure 4.22 we move complexity from the *Design* object into a new structure. A *Line* can be directly evaluated to determine its length, by taking the distance between its component points. We can see that through the addition of structure, we can define a *safer* design space that excludes more of the invalid designs. We have not yet shown how we can make DSE more efficient. For that purpose, we must first identify which of the properties for these designs we care about.

If the problem that must be solved is that of finding the most voluminous cuboid that will fit certain dimensions, such as length, width, and height shown in Figure 4.23, then defining a search space in terms of those dimensions is an even better approach. The trouble is that, in the general case, *checking* a property is easier than defining an object that has it.

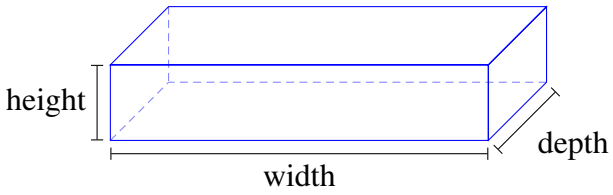


Figure 4.23: A parametrized design space of cuboids.

Recall that a definition given *in terms of some property* that must be respected is *intensional* whilst a definition given *by enumeration* is *extensional*. Given sufficient structure, we can at least relate these definitions. In the case of cuboids defined as *collections of lines*, we can use the evaluation morphism (from lines to lengths), to determine the cuboids dimensions, and then *index* the initial object that describes cuboids with the newly determined properties. This will allow to recover a definition for suitable comparison operations on cuboids, from those on lengths.

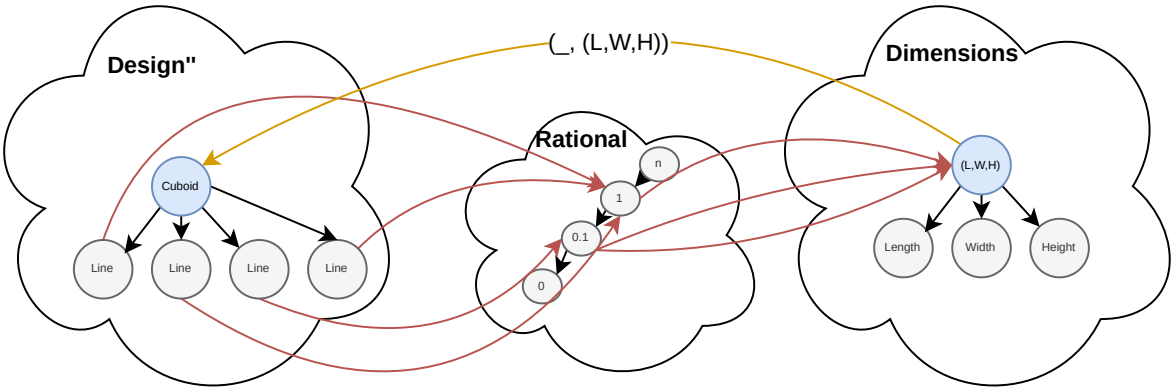


Figure 4.24: The category of *Point* objects.

In Figure 4.24 we have shown a category where the initial object, called **Cuboid**, is represented as a product of lines (only some of which are shown due to space considerations). Each of these lines, defined as a product of points, can be evaluated using the functor pointing to the category of rational numbers, to determine its length. A three-tuple of lengths fully specifies all dimensions that a cuboid may have, and so the object shown as the **(L,W,H)** in the category of *Dimensions* represents this. The functor labelled (_,(L,W,H)) *identifies* all cuboids in the *Design"* category that have those dimensions. Because functors also map the *morphisms* in a category, not just the objects, the $\leq$ operation on rational numbers induces three comparison operations on *cuboids indexed by dimensions*: $\leq_L$, $\leq_W$ and $\leq_H$; one for each dimension.

We will call the functor from **Design"** to **Rational**: *MkLength*, to distinguish it from *MkDim*, the functor from **Rational** to **Dimensions**. We know that the composition of two functors is also a functor, we'll call the composition of *MkDim* and *MkName* simply *F*. This functor *F* is *adjoint* with the functor pointing in the inverse direction: *(_,(L,W,H))*. This pair of adjoint functors trivially preserves the ordering relations that were defined through it, namely: $\leq_L$, $\leq_W$ and $\leq_H$. It also defines a *weak notion of equivalence* between a *Cuboid* and *(L,W,H)* objects. This notion is weak as there may be multiple cuboids with having the same dimensions.

Having shown that a pair of adjoint functors can relate an *intensional* definition to an *extensional* one, we now seek a way of deriving one definition from the other. Whilst this may not be possible in the general case, it is viable in certain conditions. The trick is to first define an *initial object* in the category that describes our design space, that a *canonical value* for each property that we are interested in, and then define every other object in that category through transformation of this initial object. In the case of *cuboid design spaces* a strong intensional definition would be phrased in terms of *transformations applied to a unit cube*, depicted below in Figure 4.25.

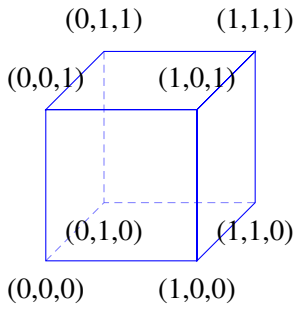


Figure 4.25: The unit cube: the initial object in the category of cuboids.

Having *statically* defined the unit cube such that all three of its dimensions are set to the unit value, we can use it as a point of reference for every other cuboid. The literal value for this unit cube is given in Listing 4.25.

```
unitCube :: Design
unitCube = MkDesign
(MkPoint 0 0 0) (MkPoint 1 0 0) (MkPoint 1 1 0) (MkPoint 0 1 0)
(MkPoint 0 0 1) (MkPoint 1 0 1) (MkPoint 1 1 1) (MkPoint 0 1 1)
```

Listing 4.25: The unit cube.

We can define a full cuboid design space by providing a way to transform this initial value, such that any valid input generates a valid cuboid as the output. If we defined a *specification function* that used such transformations to generate a design space, starting from the initial value of a unit cube, its type would look something along the lines of that shown in Listing 4.26.

```
specifyCuboids :: Design −> [Design]
specifyCuboids = [id unitCube, transform_1 unitCube, transform_2 unitCube, ...]
```

Listing 4.26: Cuboid space specification anamorphism.

If instead of specifying a concrete collection, in this case a list, we generalize the collection to be any functor F then the type becomes $specifyCuboids : Design \rightarrow FDesign$.

Any cuboid having dimensions: (l,w,h) can now be reconstructed from the unit cube by lengthening each dimension according to the ratios $(l/1, w/1, h/1)$. Notice that this type matches that of an **Anamorphism**, as defined in section 2.5. This specification definition *generates* a structure containing multiple *Design* values from a one. Specifying the *act of generating a design space* as an *Anamorphism*, not only requires that our choice of collection representation F be a valid functor, but also that the *F-CoAlgebra* on this functor F is a *structure preserving morphism*. The *identity operation*, **id** in Listing 4.26 trivially satisfies this requirement as it does not change the input Design value in any, as shown in Listing 4.27.

```
idCuboid :: Cuboid −> Cuboid
idCuboid c = c
```

Listing 4.27: Identity transformation on cuboids.

The other transformations used to specify a design space must likewise be structure-preserving. This simply means that the output Design agrees with the input Design in every property of interest. For cuboids the *correctness property* was specified using *isCuboid*. In Haskell, we can check this property as shown in Listing 4.28.

```
data Face = One | Two ... Six
doubleCuboid :: Face −> Design −> Design
doubleCuboid f c = [...]
```

Listing 4.28: Doubling transformation for cuboids.

4.2.2 Analysis

Specifying the problem of DSE using Category Theory allows us to reason about the *analysis* phase as well. Whereas the *specification* phase can be represented as an *Anamorphism* because it creates a structure of sub-problems, from the initial condition, the *analysis phase* must simply translate each sub-problem into a domain where the objective function can be specified. The *analysis* phase is thus a simple mapping operation, parametrized by an *evaluation function*.

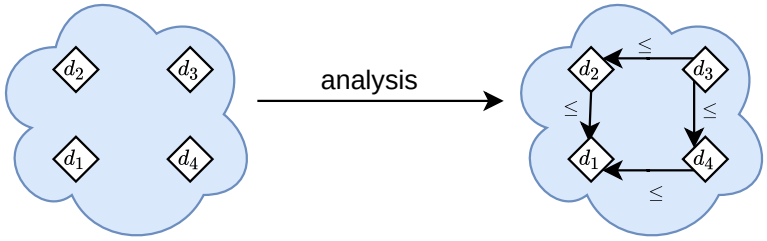


Figure 4.26: Analysis: maps the design space into an (ordered) space of properties.

The trick to deriving an efficient *analysis* phase is much the same as the trick we used to define the *specification phase*: we must decompose the problem of determining a *term's* properties into a set of smaller problems: determining each *sub-term's* properties of interest, and then recomposing the partial solutions into a greater one. Splitting up this problem is straightforward: all of our generated *design-point* retain full structural information by definition as structure-preserving morphisms.

```
transformAnnotated :: (Design, Cost, (Length, Width, Height))
  -> Dim -> Scale
  -> (Design, Cost, (Length, Width, Height))
transformAnnotated (d , cost, (l ,w,h)) dim scale = case dim of
  Length -> (transform d dim scale, cost + costL, (l * scale , w, h)
  Width  -> (transform d dim scale, cost + costW, (l , w * scale, h )
  Height -> (transform d dim scale, cost + costH, (l , w, h * scale )
```

Listing 4.29: Scaling transformation for cuboids.

In practice, we may not always have access to an entirely accurate *cost-performance model* but instead have to rely on an approximation of one. If we had, then the optimal solution would have been computable from the specification of the problem. If the *cost-performance model* is not fully accurate, then we must instead specify the minimum level of accuracy required. Formally, finding the *globally optimum* solution means there exists no other with better performance and a lesser cost. This translates into the practical requirement that the *cost-performance model* never **under-estimates the cost** and also never **over-estimates the performance** of a solution.

4.2.3 Selection

The last stage in DSE is the *selection* of the best-performing and *least costly design point* found within the design-space. This process can be implemented as a *Catamorphism* over the structure of the design space. We are searching for both the most performant and least resource-intensive solution meaning that we must solve a *multi-objective optimization problem*. The *objects* that make up a category are only distinguishable by their relation to other objects in the category. These relations are embodied by the morphisms between the objects meaning that the selection phase is can be modelled by a forgetful functor that retains only the objects that have the required morphisms. For example, let us represent the design space of cuboids as a category where objects are the product of every cuboid's dimensions. Suppose that we wanted to select those cuboids having both smallest first dimension and second dimensions.

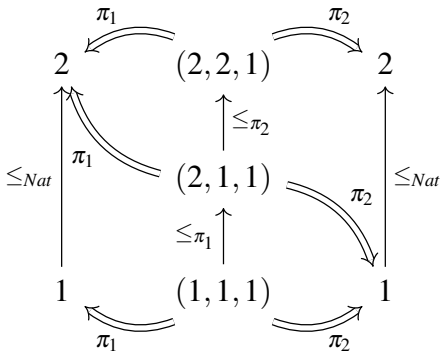


Figure 4.27: Selection via Functors.

The morphisms that related the properties of interests are $\leq_{\pi_1}$ and $\leq_{\pi_1}$ shown in Figure 4.27 above. If we stick to the initial view of DSE as a three stage process, then the objects in our category of cuboids will first have to be constructed and only then filtered. Filtering requires that we define $\leq_{\pi_1}$ and $\leq_{\pi_1}$ as the composition of either the π_1 or the π_2 projection functor respectively, and an additional adjoint functor for each, that identifies the morphism.

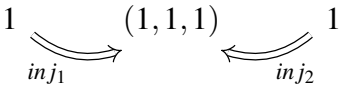


Figure 4.28: Selective object construction.

This means running and collecting the relations for each property of interest, which is clearly less than ideal. Instead, we will interpret this category differently, by fusing the *selection* phase into the *generation* phase. In this way, only the objects that satisfy the required properties are constructed.

4.3 Optimal DSE in TyTra

We've shown that a *Design Space* can be generated from a *collection of transformations*. In more formal terms, a *Design Space* is *equivalent to* or *identified by* the *transitive closure* of these *transformations*. Recall that in section 2.2.1 we considered *compiler flags* to be a *representation of program transformations*. If we consider a static *collection of compiler flags* than a compact representation for the entire collection might be a *list of boolean values*, as shown in subsection 4.3.1, where each value denotes whether a particular transformation is to be applied.

```
designSpace :: [Bool]
designSpace = [True, False, True, True]
```

Listing 4.30: A simple Design Space Structure.

With this representation we may define an *exhaustive* DSE strategy that simply enumerates all possible binary strings, measures the relative performance and resource-cost of each and then selects the best overall solution. There are a number of problems with this approach.

1. This representation for a design space includes every possible program transformation, however, for a *given input* only a sub-set of these transformations may be applicable. It is readily apparent that a program consisting purely of *map operations* would not benefit in any way from the application of *fold-specific transformations*, for example.

2. There are only two possible states for each transformation: either it is globally *enabled* or *disabled*. The same transformation that is beneficial when applied to a certain sub-term, may also be counter-productive when applied to a different sub-term.

The first of these issues can be solved with a simple representation and methodology change. We can inspect the *structure* of the application and only represent the *applicable program transformations*. Because the set of transformations under consideration is now application dependent, we can no longer identify each transformation with its index in the ordered collection, and so we must explicitly denote each transformation by its name, as shown in Listing 4.31.

```
designPoint :: [String]
designPoint = ["MapFusion", "FoldFission", ... ]
```

Listing 4.31: A Human-readable Design Space Structure.

The representation in Listing 4.31 is more efficient than, but entirely equivalent to, that shown in Listing 4.30. We are simply leveraging the *set-inclusion relation* to encode the previously explicit boolean value. If a transformation is included in the set, it is enabled.

The *map fusion* and *fold fission* transformations lend themselves to both the boolean list representation of design spaces and the string list representation of a design point. This is because they are **non-parametric transformations**, akin to the *doubleCuboid* transformation defined in Listing 4.28. The fact that they are representable as such is due to the exists of a **unique isomorphism** between a sequence of map applications and *the application of a single, fused map operation*.

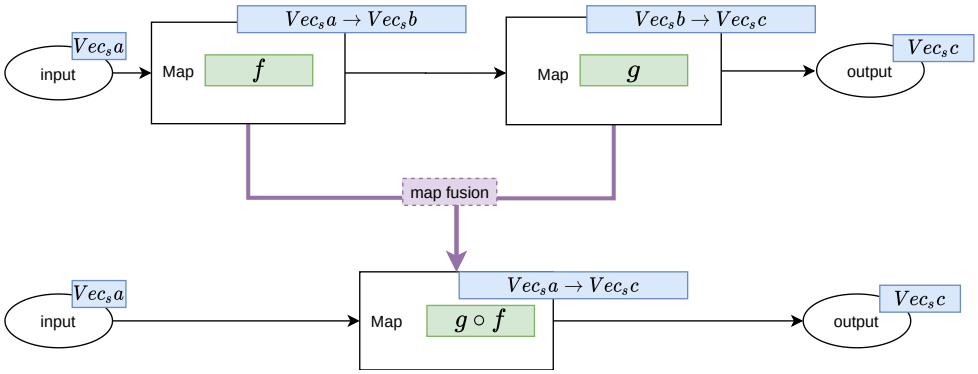


Figure 4.29: Map fusion rule.

Other optimizing transformations, are parametrized and thus can not be represented by their name alone. The *split transformation*, for example, takes a number k which specifies a degree of parallelism to be expressed. Consequently, the *split transformation* can be seen as entire family of non-parametric transformations.

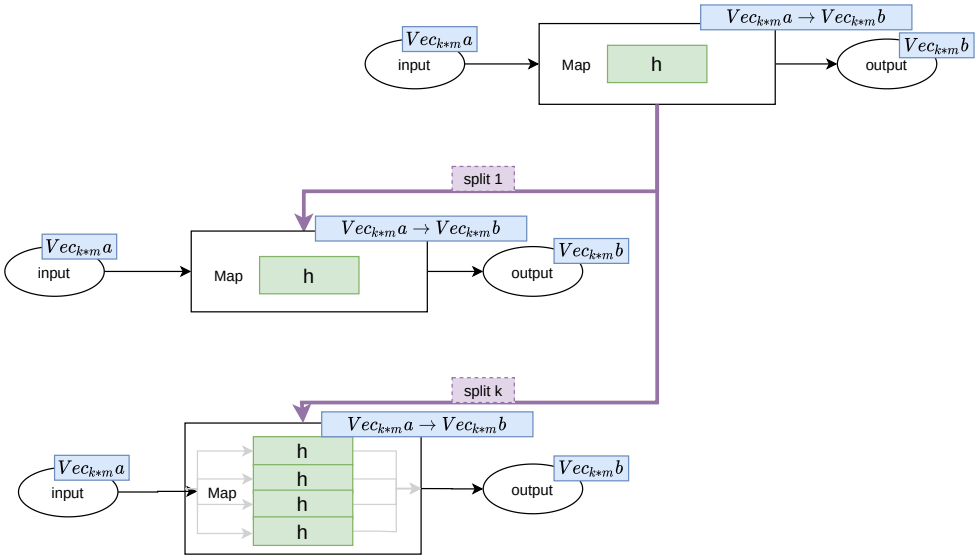


Figure 4.30: Family of *split* transformations.

The parameter must additionally be a valid divisor of the size of the input vector to a map or non-trivial fold operation because the act of splitting an input vector of type $Vec_{k*m}\ a$ using *split* k must produce an output type of type $Vec_k\ (\ Vec_m\ a\ )$. Our choice of representation is proving to be unwieldy, however workable.

Each conceivable transformation parameter is of a *finite type* meaning that we can *enumerate all of them*. The result of this is initial collection of applicable transformations. The design-points in the *application-specific search-space* are then all possible combinations of these transformations. A simple way of encoding this additional information is to attach a *path* that specifies which sub-term each of these transformations targets.

```
newtype DesignPoint = [String]
newtype DesignSpace = [DesignPoint]

designPoint_one :: DesignPoint
designPoint_one = ["MapFusion", "FoldFission", "Split_6" ]

designPoint_two :: DesignPoint
designPoint_two = ["MapFusion", "FoldFission", "Split_2" ]

designSpace :: DesignSpace
designSpace = [designPoint_one, designPoint_two]
```

Listing 4.32: DesignPoints with serialized parameters.

The second issue can be solved by further altering our representation of design spaces. Recall that in section 2.2.1 we mentioned the benefit of using *pragma directives* over *compiler flags*: they contain targeting information. This means that our design space must not only encode whether a particular transformation is enabled, but also *for which sub-terms*.

```
newtype Location = [Int]

newtype DesignPoint = [(Location, String)]
newtype DesignSpace = [DesignPoint]

designPoint :: DesignPoint
designPoint_one = [([0], "MapFusion"), ([0,0], "FoldFission")]
```

Listing 4.33: A location-aware Design Space.

If we also specify a *serialization strategy searialize* : $Design \rightarrow String$ we can revert to the second definition of a design point, shown in Listing 4.31 but retain the efficiency benefits of only generating the design space that corresponds to the application's structure. This is not the most efficient solution, but it does provide a way to generate an *exhaustive yet application-specific* search-space as we now show in subsection 4.3.1.

4.3.1 Naïve tactic

Let us denote transformations through symbols $f, g, h, .. \in S$ where S is the *set of all enumerated transformations*. We might call this set the *initial transformation set* and use it to define the **total design space** as the *power set* of $S : \mathbb{P}(S)$.

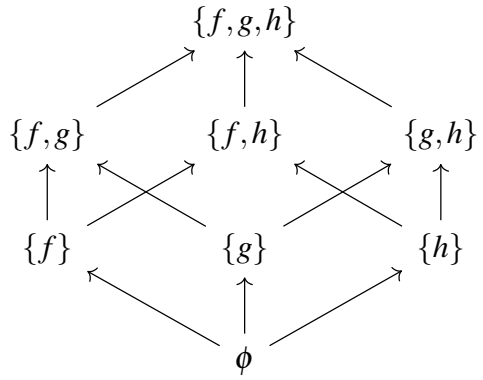


Figure 4.31: Power set of transformations.

Having defined an initial set of transformations and a way to enumerate the entire design spaces. We can start from the distinguished element of the power set: ϕ, known as the **empty set**, which, in the present context, corresponds to the *identity transformation*. It has no operational effect but, because it is equivalent to the *identity morphism* in Category Theory, it also the *left* and *right* unit of *morphism composition*:

$$\phi \cup \{f\} = \{f\} = \{f\} \cup \phi$$

We can continue by enumerating other sets $d \in \mathbb{P}(S)$ by incrementally adding transformations from S and forking the state space, spelling out a *brute-force* backtracking approach to DSE. We can model this iterative process with a finite state automaton that *recognizes* the *words* of our *transformation language* and has q_ϕ as the starting state.

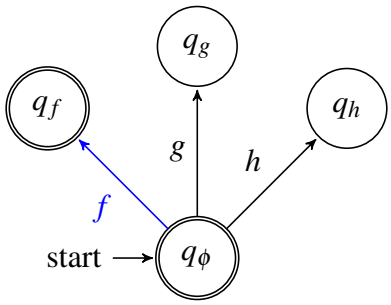


Figure 4.32: Brute force FSA accepting states.

Every other state within the FSA is reachable by adding one of the remaining *program transformations* $f, g, h, \in S$. The exact make-up of the initial set of transformations S depends upon the structure of the application being compiled.

Specification

We will assume that the types for the application under consideration have been inferred and checked using the process we will show in subsection 4.4.1. If that is the case, we can pattern match on the type-level structure to generate an exhaustive design-space. In the case of *Variable Terms*, for example, the underlying expression may be either an *opaque function* or an *input variable*.

```
generateTypeLevel :: Term -> TypeEnv -> DesignSpace
generateTypeLevel term tyEnv =
 case term of
  (Variable name) ->
   case (inferType tyEnv term) of
    VarTy tyName -> [...]
    FunTy iTy oTy -> costBased name
```

Listing 4.34: Generate Initial Transformations (Type).

Variable Terms are a very simple type of TyTra CL term expression. They retain the symbol name provided as a *String* argument to the constructor. As we pattern match on the structure of the TyTra CL *type constructors*, we necessarily increase the complexity of this discussion.

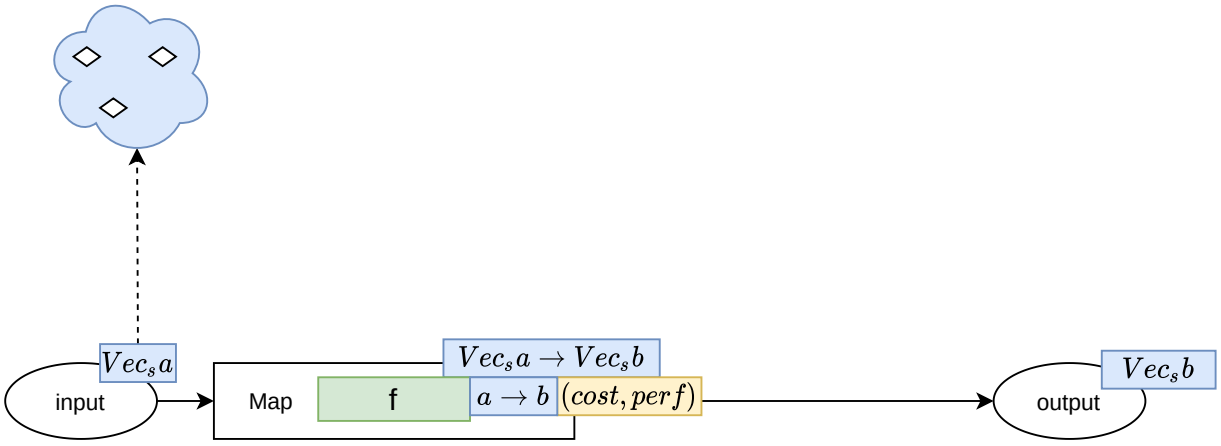


Figure 4.33: *Data Variable Terms* generate DS with expressed parallelism $\leq$ size of vector.

If a type-level inspection of the *Variable Term* reveals an input data node, this signifies that we can replicate it, as we do with opaque functions, to deliver more performance. It may then seem as if there is no need to perform DSE on such simple expressions. That situation is only valid for *Scalar Types*. With *Vector Typed* data, however, we may choose to deliver more or less components at any one time. This choice gives rise to a design subspace as the number of data items deliver per clock cycle dictates the maximum degree of parallelism for consuming nodes.

Opaque functions can be parallelized up to a factor the permits them to process one item of data per clock-cycle at their *steady-state*. This factor is equal to the *Effective Firing Interval* (EFI) which is a component of the *performance* estimate, shown with the yellow *(cost,perf)* box.

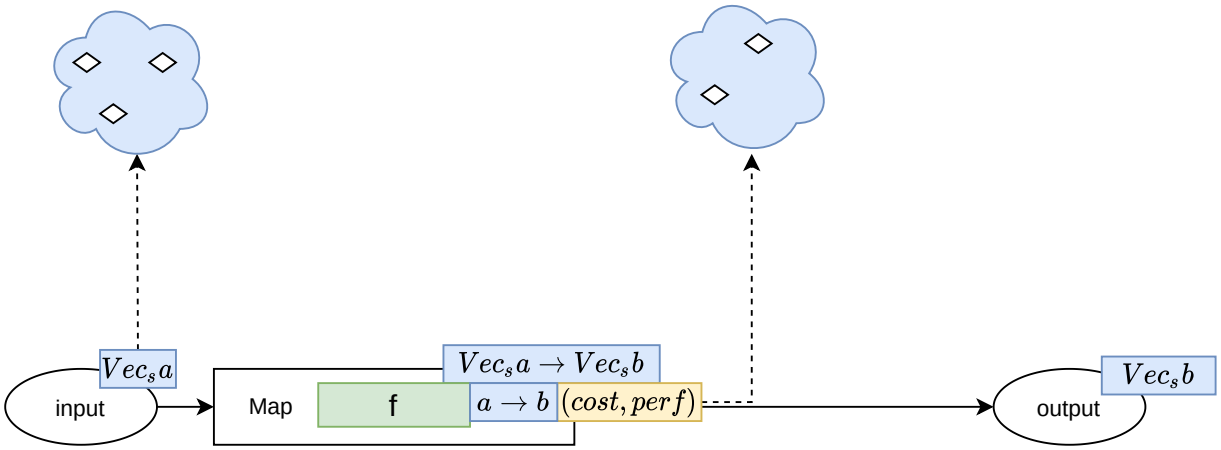


Figure 4.34: *Function Variable Terms* generate DS with expressed parallelism $\leq$ EFI.

Beyond *Variable Terms* there may be other **intermediate** or *top-level* expression such as *application node*. With these, we must compute the set of transformations *from the sets of transformations corresponding to their component expressions*.

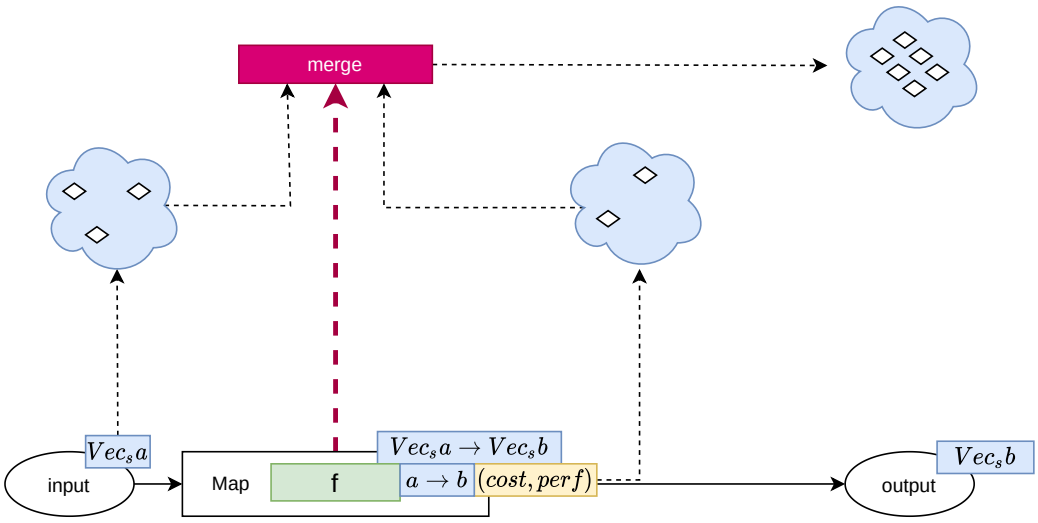


Figure 4.35: Design Space merging

As we traverse the TyTra CL AST, we accumulate an increasing design space by taking the cross-product of subspaces generated by the component expressions. This is perhaps the greatest source of inefficiency in this approach to DSE. With this running example, the subspaces belonging to the input variable contained three design points, that belonging to the opaque function contained two, leading to an overall count of six design points for the parent expression.

Analysis

Having produced an overall design space for the entire TyTra CL application, we must now rank design points by their ability to produce high-performance program variants. Decomposing this process, we notice that it involves a sequence of structural traversals on:

1. The *design space structure* to project the contained design points.
2. The structure of each *design point* to reveal the selected transformations.
3. The term and type structure of the application to apply the transformations.
4. The term-level structure of the now transformed application to run the *cost-performance model* once more
5. The structure of the *cost-performance estimate* to compare results.

The first two of these traversals are straight-forward, thanks to our choice of representing design spaces and design points as simple lists.

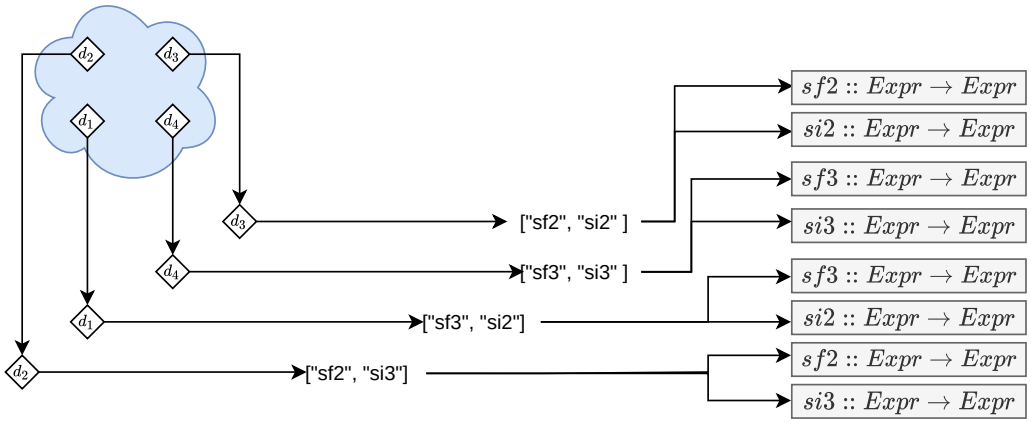


Figure 4.36: Design space structure traversal producing program transformations.

The third traversal is similar to *term-level* traversal we employed to generate the space.

```
transformTypeLevel :: Term -> TypeEnv -> Term
transformTypeLevel term tyEnv =
 case term of
  (Variable name ) ->
   case (inferType tyEnv term) of
    VarTy tyName -> si2 term
    FunTy iTy oTy -> sf2 term
```

Listing 4.35: Term/Type Post Analysis Transform.

It would be superfluous to list out the remaining traversals as it now clear that the second major source of inefficiency with this approach is the amount of effort spent repeatedly packing and unpacking the data structures used to represent the *design space*, *the term level representation of the application*, the *type information* and the *cost-performance estimates*.

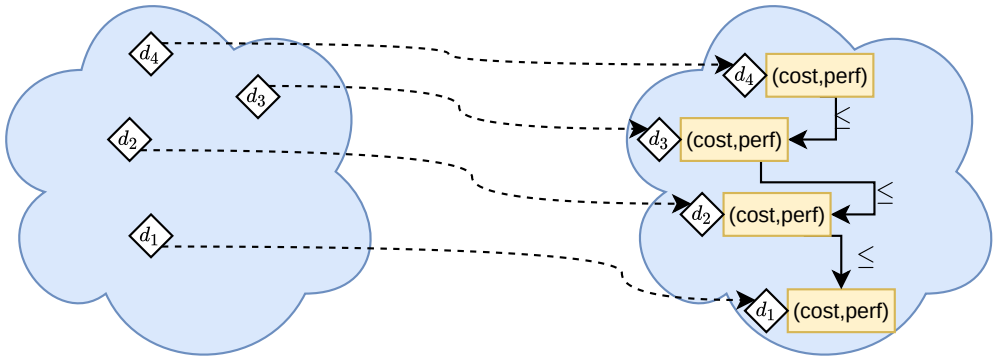


Figure 4.37: Cost-performance model output.

Selection

One last hurdle remains, and that is to select the best performing set of transformations. Since the output of the cost-performance model gives us an ordering, we need to sort our design points and filter out those that violate any constraint, such as utilizing more hardware resources than are available for our target device.

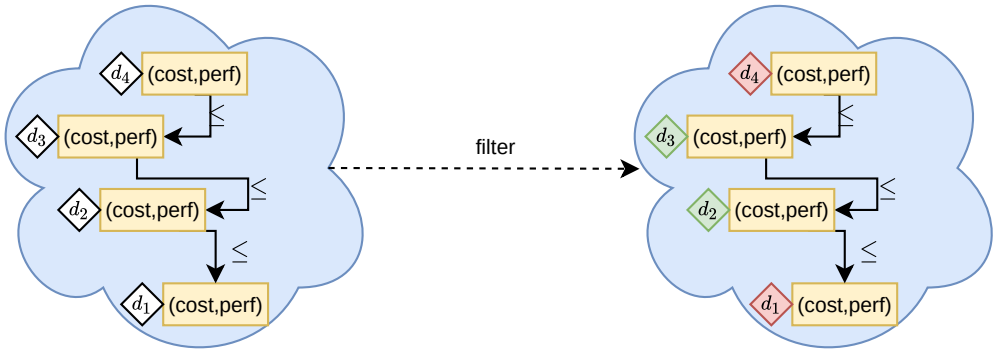


Figure 4.38: Final selection of program transformation sets.

Looking at our example, we notice that even though d_4 may have outperformed all other candidate solutions in terms of throughput, it required us to replicate f more times than our hardware resource budget allows. Another design point d_1 would have also exceeded resource bounds, and would not have given us any better performance results. The remaining solutions, d_3 and d_2 are all that remain. We'll select the least costly of these: d_3 as the final answer.

4.3.2 Expert tactic

Seeking to improve the performance of our DSE strategy, we looked into how FPGA programming experts optimize their applications and particularly into the heuristics they use.

It turns out that one of the most important heuristics applicable to *dataflow applications* is conceptually rather simple.If the application can be represented as a *computational pipeline* then the *most resource-efficient* way to maximize *throughput* is to shape the pipeline such that the flow of data is **balanced** throughout.

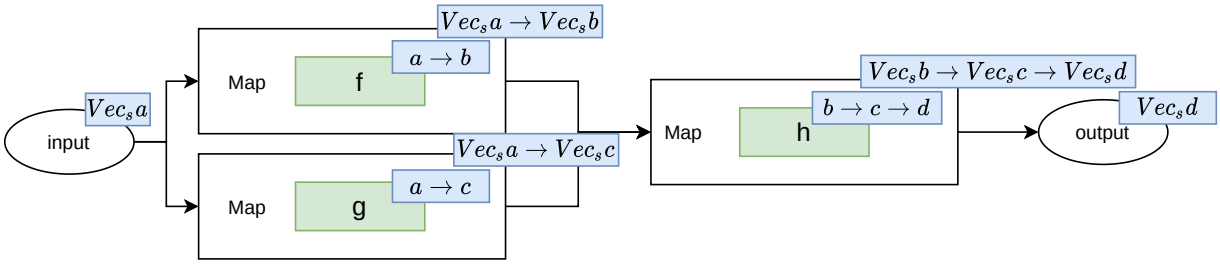


Figure 4.39: A computational pipeline.

The term-level *split* transformation derived from *Vector Type isomorphisms* can transform the pipeline depicted in Figure 4.39 by **replicating computational nodes**, *scattering* the inputs and *gathering* the outputs, such that the result still has the appropriate type.

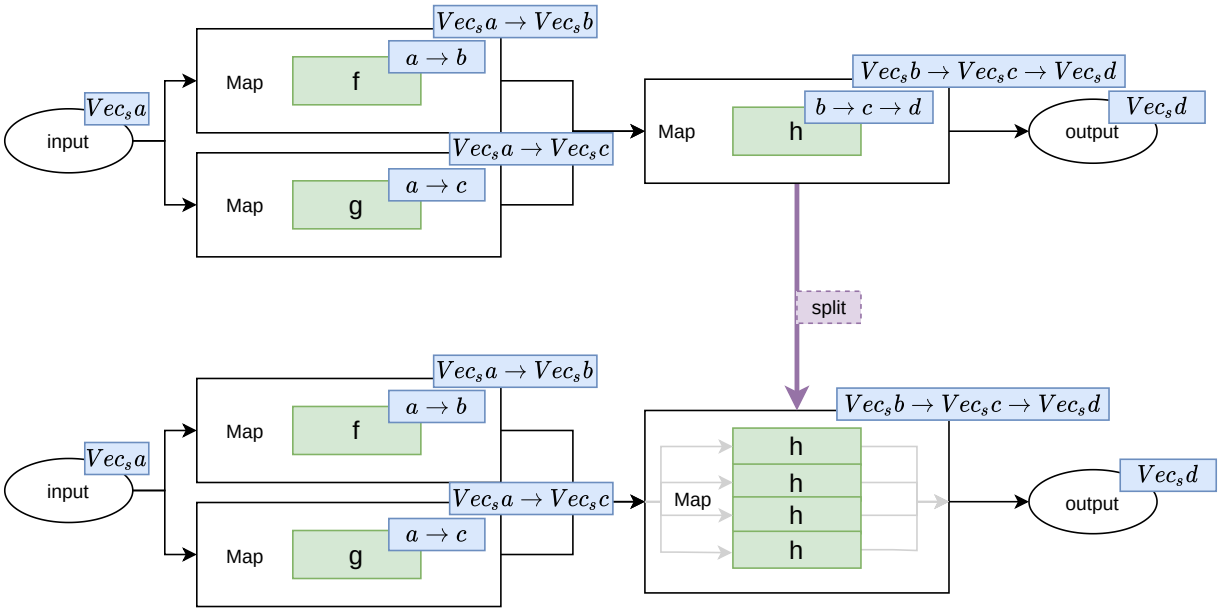


Figure 4.40: Effect of *Split Transform* on computational pipeline.

Where the *h* opaque function would have initially taken *four clock cycles* to produce an item of output, the transformed version is now **capable** of producing an output on *every cycle*.

Splitting computational nodes enough times to deliver peak throughput does not guarantee that said performance is achievable in practice. To understand why, let us consider a simple pipeline with two opaque functions f and g, both of which require a four way splitting.

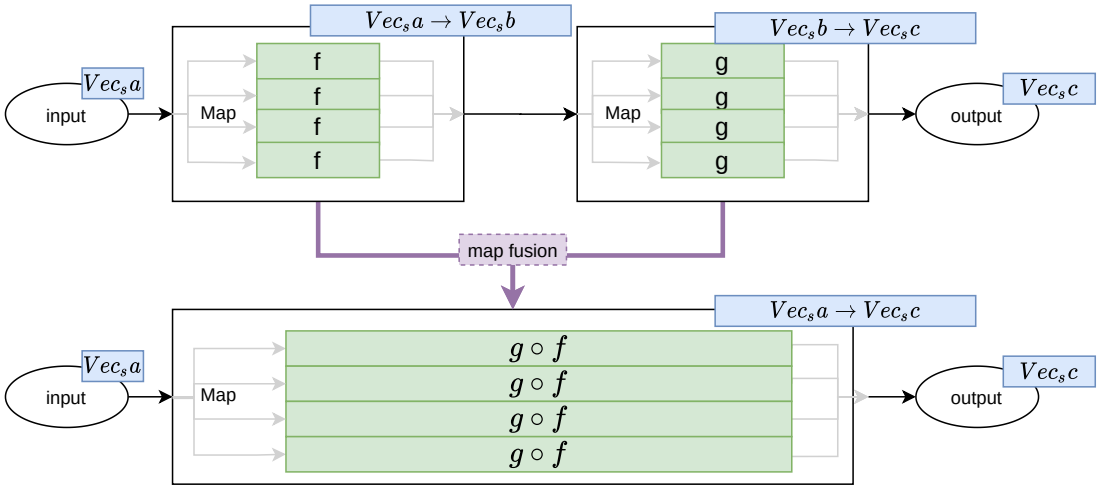


Figure 4.41: Balanced computational pipeline.

The two nodes in Figure 4.41 make up a **balanced pipeline**. As soon as f produces an output, g is ready to receive its input, thus there are no *stalls*. Whenever f and g operate in lock-step, we can make use of *map fusion* to eliminate *one multiplexer* and *one demultiplexer* from the circuit and simplify the overall *design space*.

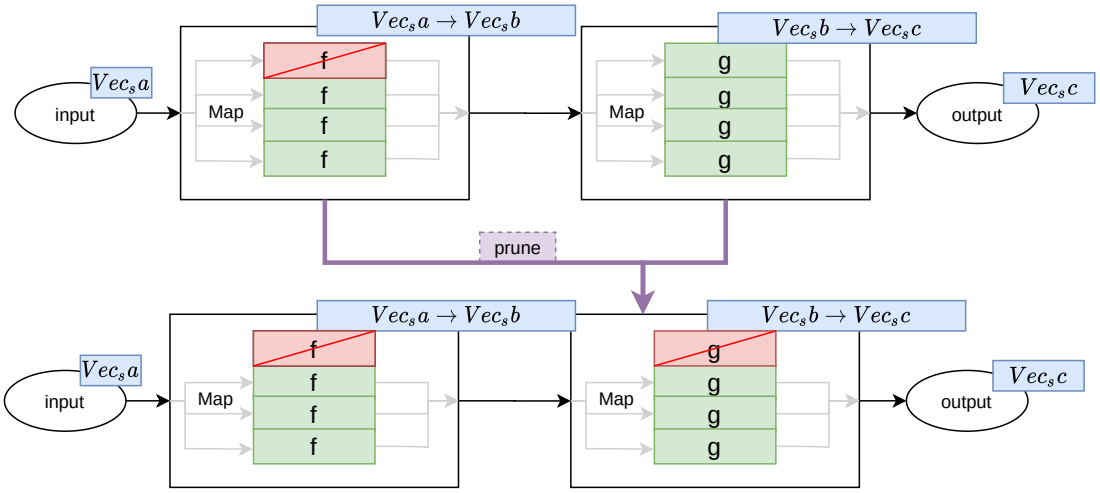


Figure 4.42: *Map Fusion* on balanced computational pipeline.

If for any reason f can only be replicated *three times*, we can **rebalance the pipeline** and save on hardware resources by pruning g, at no detriment to throughput. The more general implication of this heuristic is even more interesting. For certain kinds of terms, the space generated by sub-terms may *functionally depend* on one another.

Variable Terms are treated exactly the same as in the brute-force approach. More complex terms such as **Map Terms** and **Fold Terms** need further attention because they have a recursive constructor argument. In both cases the parameter is a single TyTra CL expression with a *function type* and thus there is no need to merge design spaces. *Map Terms* and *Fold Terms* simply cause their parameter's *design space bounds* to be expressed in the outer expression.

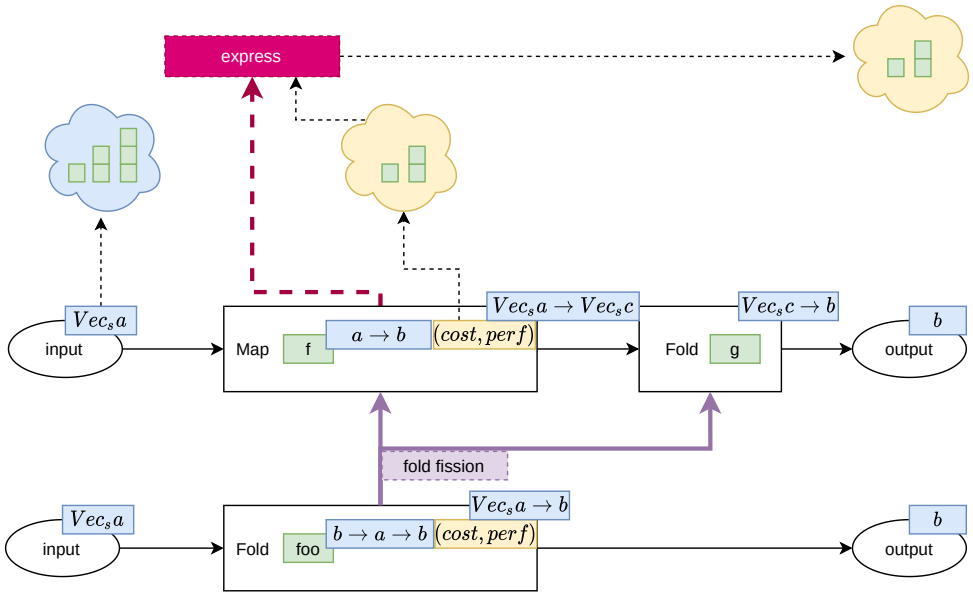


Figure 4.43: *Fold Term* fission transformation.

According to the *equational reasoning* enabled by the *category theoretical semantics* we gave to the TyTra CL language in subsection 4.3.1, *Fold term* can be treated as a special case of *Map Terms*. The category theoretical concept that maps to folds is that of *Catamorphisms* which consume structure, as we have shown in Listing 4.9. Because the output of the fold operation depends on both the structure and the *values contained within that structure* we can split them into a **parametrically polymorphic function** and a **monomorphic** function. This is known more widely as the **fold fission** law and states that a fold can be represented as another, potentially simpler aggregation, *fold g*, and an additional map operation, *map f* which can no be treated in the same way as any other map operation.

$$fold\ foo = fold\ g \circ (map\ f) \tag{4.2}$$

The simpler aggregation $fold\ g$ has no *mapping component* as that part of the computation that can be represented as a map is entirely represented by $map\ f$. Although it is technically possible to further split the primitive aggregation, there would be no point in doing so. The nested fold splitting would yield the equation: $fold\ g = fold\ h \circ map\ id$. In the larger equation this means that $fold\ foo = fold\ h \circ map\ id \circ map\ f$. By applying the map fusion law we can simplify this to $fold\ foo = fold\ h \circ map\ (id \circ f)$. Given that $\forall f.id \circ f = f \circ id = f$ it must be that $g = h$.

Tuple Terms have *Product Object* semantics. The *bounds produced by* a tuple expression are the *product of the bounds of its projections*. Set in the context of *balanced computational pipelines Tuple Terms* are a statement of the fact that: **the property of a pipeline being balance is distributive**. This means that all of a tuple's sub-terms are functionally codependent.

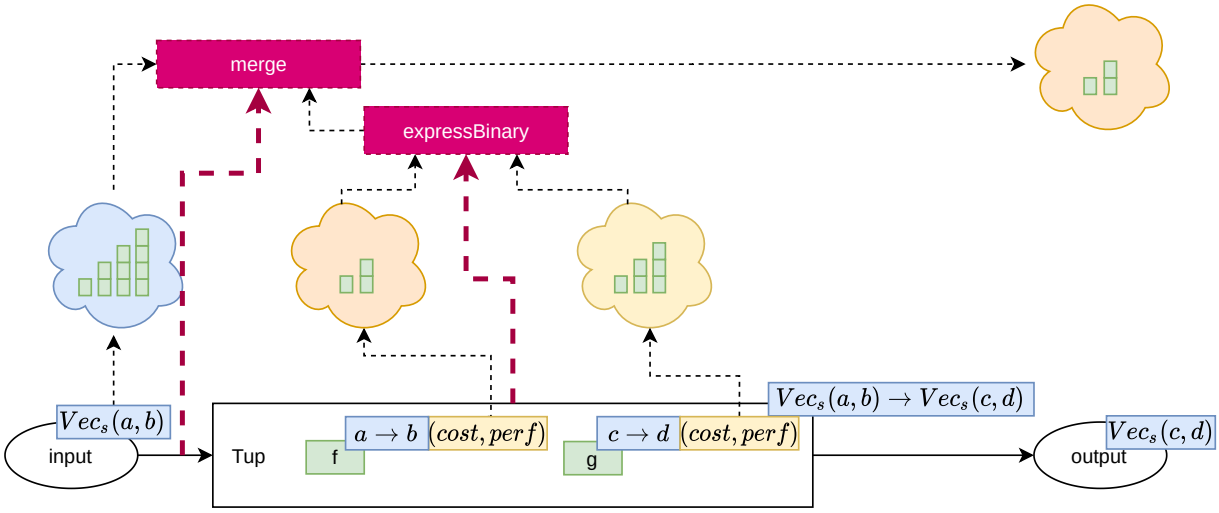


Figure 4.44: *Tuple Term* distributes bounds.

Application Terms, denoted as the simple black arrows, introduce a mutual dependence between the *action expression*, meaning the function to be applied, and the *input expression*.

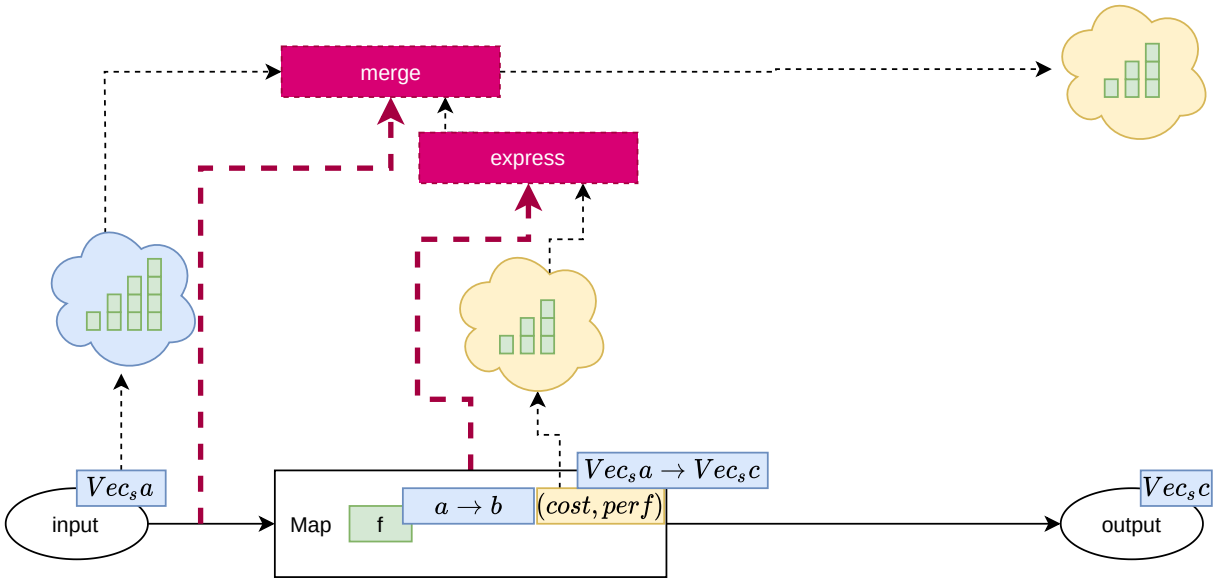


Figure 4.45: *Application Term* merges bounds.

Note the similarity in the effects that *Tuple Terms* and *Application Terms* have on their respective design spaces, they both *merge sub-search spaces* in the same way. This is due to the adjoint relation between *product* and *exponential* objects in a *Closed Cartesian Category*.

.

ZipT and **UnZipT Terms** are simply convenient symbols for a pair of dual higher-order functions that pack a tuple of vectors into a vector of tuples, and vice-versa. Their occurrence within a TyTra CL expression necessarily implies a parent **Application Term** which introduces a mutual performance dependence between the function being applied and *all ZipT input components*, respectively *UnZipT output component*.

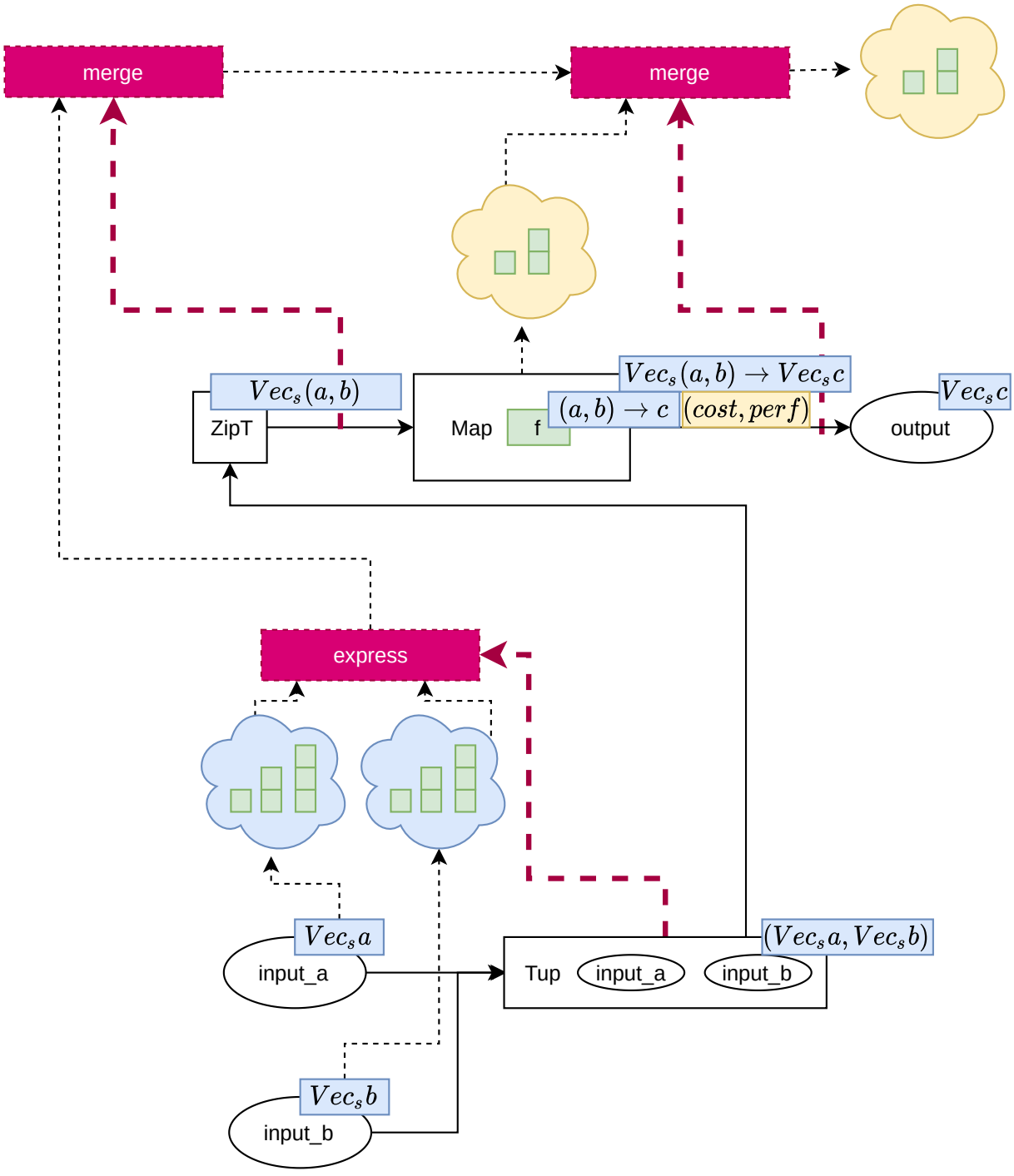


Figure 4.46: *ZipT Term* merges bounds.

Elt Term are parametrized by an *index* that specifies which component of a *finite product* must be extracted.

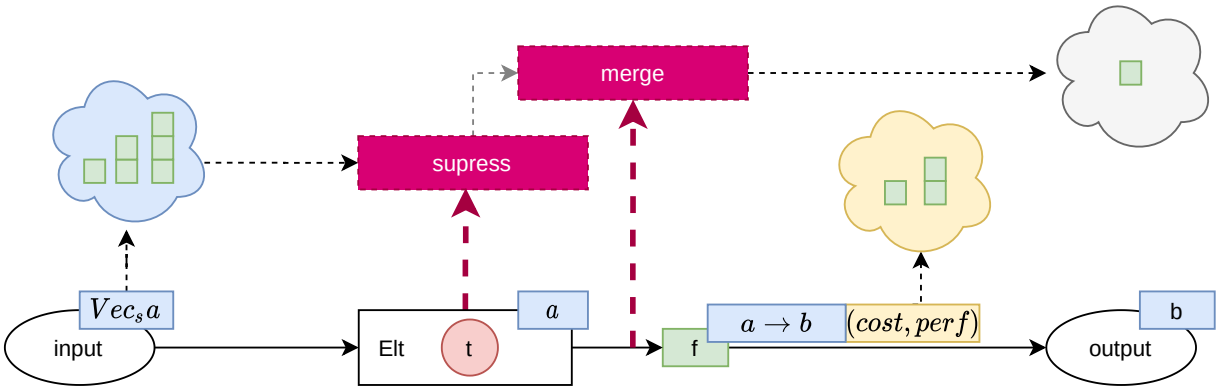


Figure 4.47: *Elt Term*.

Elt Terms influence DSE somewhat differently from all other terms we've covered so far. They alter the *cost-performance estimate* of the expression they are applied to, by increasing the initial delay with an *absolute* number of clock-cycles equal to the index parameter. **Stencil Terms** are parametrized by a collection of *relative* indices and thus introduce a *buffering requirement*. This leads a *significant* but *constant* hardware resource cost growth. *Stencil Terms* increase the *initial delay* similarly to *Elt Terms*, however they do so by a number that is equal to the *largest difference* between the *index parameters*.

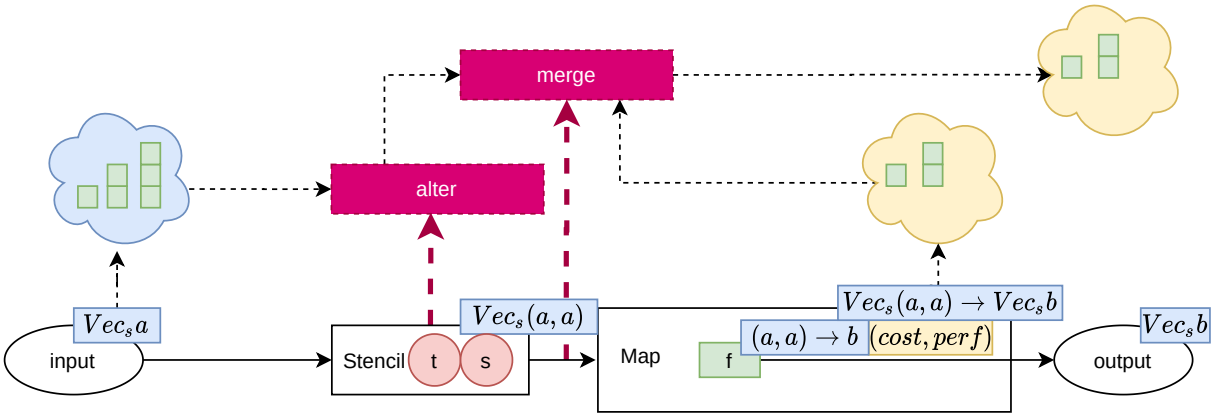


Figure 4.48: *Stencil Term*.

Elt and *Stencil* terms represents actions that will be applied to some input, however they only have an effect on the *latency* and not the *throughput* of the overall expression, meaning that they do not influence the *balancing aspect*.

Having looked at the effects of using the *balanced pipeline heuristic* to remove sub-standard solutions from an overall search-space, we can see that there exists a functional dependency between the search-space generators corresponding to certain sub-terms. This means that the operations which merge design subspaces, which we have shown with fuchsia-coloured boxes, not only represent an opportunity to balance the computational pipeline but also the best place to limit the growth of the overall search space.

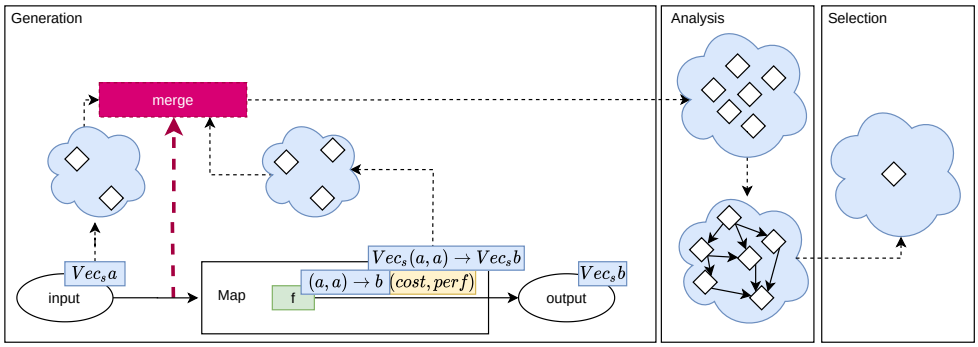


Figure 4.49: DSE on an isolated sub-term.

There are however two issues to be resolved. The first is shown in Figure 4.49. If we want to use every *merge* operation as an opportunity to remove inefficient designs points from the search space, we must perform all three stages of DSE on each sub-term, as shown in Figure 4.50. The first stage generates the full design space, the second produces cost-performance estimates for each design point whilst the third filters the output, retaining only the most valuable solutions.

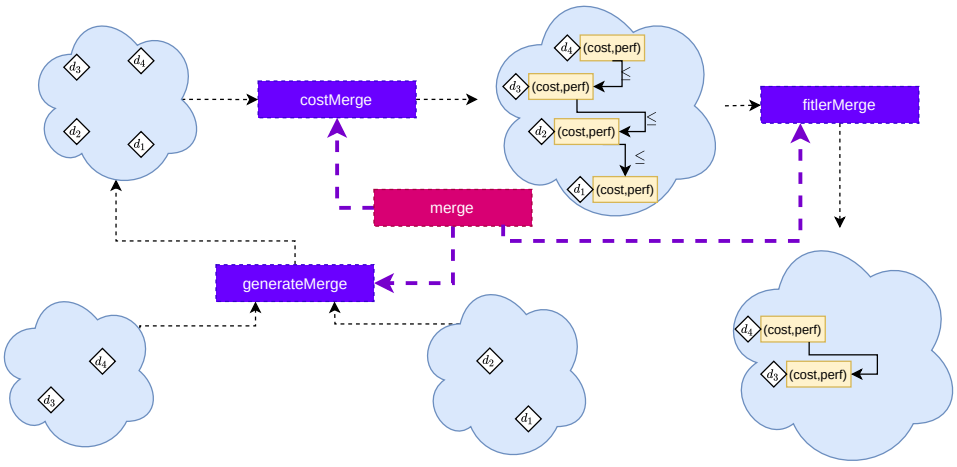


Figure 4.50: DS merge improvement.

The second issue is that our design-space as a simple list of design points makes it difficult to separate the sub-spaces that have a functional dependency on the sub-term currently under consideration. To fix both of these issues, in the next section, we will simply *borrow* a richer design-space structure form the application itself through categorical semantics.

4.4 TyTra Categorical Semantics

Having defined the structure for TyTra CL term, type and cost-performance estimate *categorical data types*, as the fixpoints of several functors including $TermF$ and $TypeF$ we now need to consider the semantics of the language they collectively define. We will do so using the same notions used to construct these categorical data types. This will allow us to relate and transform the evaluation morphisms between these types and thus construct a more efficient *Design Space Exploration* strategy. Let first see what evaluation morphisms are required in the TyTra compiler.

Correct-by-construction transformations on *TyTra CL terms* are derived from *type-level equivalences and transformations*. This implies two evaluation morphisms.

- An evaluation of TyTra CL terms as TyTra CL types:
  $inferType : Fix\ TermF \rightarrow Fix\ TypeF$
- An evaluation of TyTra CL types as *TyTra CL term transformations*:
  $generateSpace : Fix\ TypeF \rightarrow (Fix\ TermF \rightarrow Fix\ TermF)$

For this reason we will see how TyTra CL terms are evaluated by the *type inference and checking* part of the TyTra compiler to produces a *type-level representation* in subsection 4.4.1. At the same time we know that the *selection* phase of DSE removes program variants that have a lower performance and higher resource cost than other solutions already found, requiring two further evaluation morphisms, listed below and presented in detail in subsection 4.4.3.

- An evaluation of terms as *cost-performance estimates*:
  $costPerf : Fix\ TermF \rightarrow CostPerf$
- An evaluation of Design Spaces that selects the *globally optimal* program variant:
  $restrictSpace : (Fix\ TermF \rightarrow Fix\ TermF) \rightarrow (Fix\ TermF \rightarrow Fix\ TermF)$

Once we have given categorical semantics for these evaluators, we will define a category where the program to be optimized is the **initial object**. The **globally optimal transformation** in relation to a given set of resource constraints, is the **terminal object** in this same category. More importantly, we will see that the paths between the *initial* and *terminal* objects in this category are defined in terms of **term**, **type** and **cost-performance estimate transformations**. This category gives us a directed and *minimal* search-space that contains the optimal solution. To define this category however, we must solve the issue of merging the evaluation morphisms. Endofunctors that describe our categorical data types each define their own, distinct categories of *F-Algebras*. We will solve this issue in subsection 4.4.4.

4.4.1 Type Inference and checking

The TyTra CL structure defined by the *TermF* functor is mostly the same as that of the Simply Typed Lambda Calculus (STLC). One choice for type inference is **Algorithm W**, for which there are numerous accounts of implementation including [Gra06]. Another choice [Ste], much to our preferment, is closer in spirit to constraint solving. The type signature for $inferType : Fix\ TermF \rightarrow Fix\ TypeF$ shows that this function constructs a *TypeF* structure while pattern-matching on the *TermF* functor. In truth this evaluation morphism is also dependent on the type-signatures present in the TyTra CL which is not reflected by the *TermF* functor, so we will simply pass in an extra argument of type *TyEnv* with an associated $lookup : TyEnv \rightarrow String \rightarrow Fix\ TypeF$ function.

```
inferType :: Fix TermF -> TyEnv -> Fix TypeF
inferType termF tyenv = case termF of
 Fix (TermVarF name) -> lookup tyenv name
 Fix (TermAppF f i) -> let
   fSubs = generateTypeSubs f tyenv
   iSubs = generateTypeSubs i tyenv
   typeSchema = FunctionSchema
   in unifyTypeSubs typeSchema (fSubs ++ iSubs)
  [...]
generateTypeSubs :: Fix TermF -> TyEnv -> [Substitution]
unifyTypeSubs :: TypeSchema -> [Substitution] -> Fix TypeF
```

Listing 4.36: Sketch of type inference as constraint solving.

At a high-level, *inferType* has the same structure as our DSE process. We traverse the term-level structure and, at every level, build a set of candidate solutions from which we pick the best one. Rather than dwell on the specifics of type inference algorithms, we will focus on what is more important: the *inferType* function identifies an initial object *Fix TermF* in the category of *F-Algebras* over the term we are compiling, with the initial object *Fix TypeF* in the category of *F-Algebras* over the type of the term we are compiling.

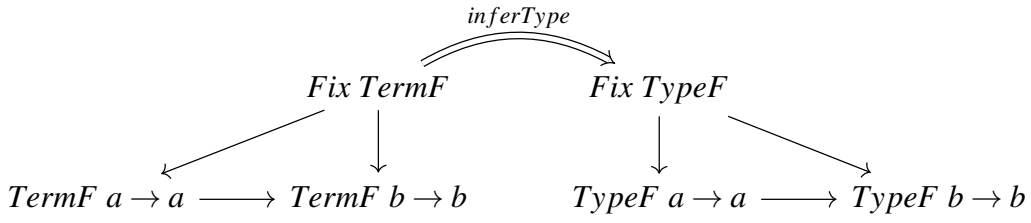


Figure 4.51: Functor from the category of term F-Algebras to the category of type F-Algebras.

4.4.2 Correct term transformation

We can derive *term-level transformations* from *type-level transformations* by specifying an evaluation morphism from a special object in the category generated by the $Fix\ TypeF$, namely the initial object. As with the *inferType* function we previously discussed, this evaluation morphism represents a functor. In this case it points in the opposite direction, as shown in Figure 4.4.2.

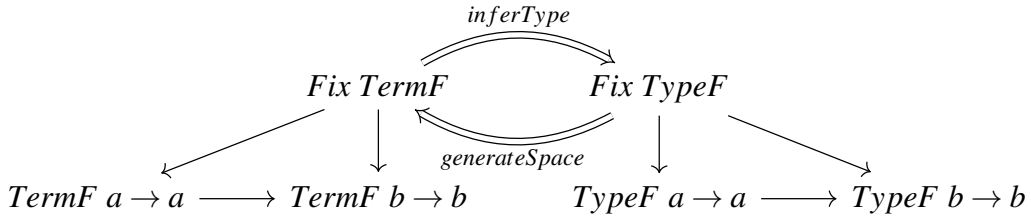


Figure 4.52: Functor from the category of type F-Algebras to the category of term F-Algebras.

The functors defined by *inferType* and *generateSpace* are in fact adjoint. If we evaluate a particular TyTra CL term identified by an object $TermF\ a \rightarrow a$ through *inferType* we identify the $TypeF\ a \rightarrow a$ object in the category of types. Considering every morphism that points away from $TypeF\ a \rightarrow a$ in Figure 4.4.2 to be a valid *type-level transformation*, the adjoint functor defined by *generateSpace* identifies the morphisms in the category of terms that define *functionally correct term transformations*. We know that the composition of functors also defines a functor. The pair of adjoint functors given by *inferType* and *generateSpace* compose, giving us an *Endofunctor* on the category of terms that, for every object in the category, identifies a category of valid transformations.

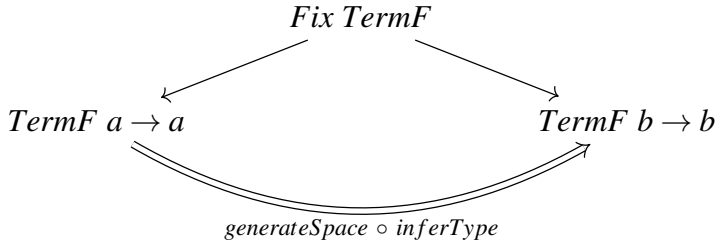


Figure 4.53: Endofunctor on the category of term F-Algebras.

By defining categorical data types for TyTra CL terms and types, as well as by giving the TyTra CL categorical semantics, we have obtained the ability to define a data type for the specific TyTra CL terms that is *sensitive* to the type context of a given TyTra CL application. The Haskell data-type for terms shown in Listing 4.11 can express any TyTra CL term, including ones that are functionally incorrect in a specific type context. By restricting term constructors to those returned by the Endofunctor defined as $generateSpace \circ inferType$ we ensure that only semantically correct terms can be *represented* using our categorical data type.

4.4.3 Cost-performance aware transformation

We will now look at the relationship between the category of TyTra CL terms, and the category of *cost-performance* estimates. This will allow us to further refine our categorical data type such that we can compare transformed terms based on their performance and hardware resource use. We define a pair of adjoint functors, defined by the *costPerf* and *restrictSpace* evaluation shown in Figure 4.54 below.

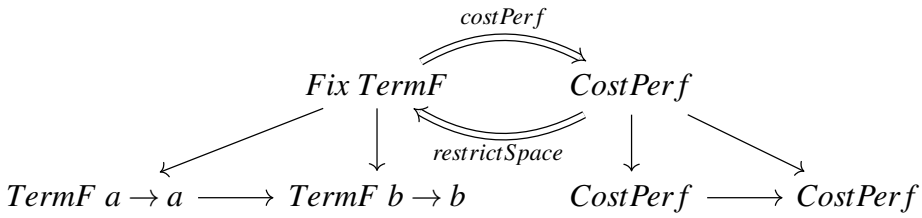


Figure 4.54: Adjoint functors relating term transformations to cost-performance expressions.

In subsection 4.1.4 we defined the type of *cost-performance* estimates as the objects in a category with finite products $CostPerf = (Fix\ PerformanceKIF, Fix\ ResourceKIF)$. This means that the category in which $CostPerf$ is an object can be constructed using a pair of functors from the categories in which $Fix\ PerformanceKIF$ and $Fix\ ResourceKIF$ are the respective initial objects.

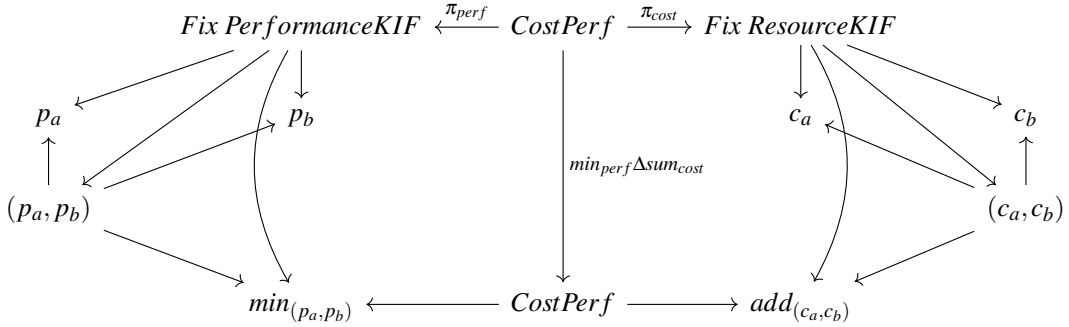


Figure 4.55: Category of performance measures.

Internally, the objects in both the category of performance estimates, and those in the category of resource costs, are nothing more than products of natural numbers for which the usual operations and relations, including addition and $\leq_{\mathbf{Nat}}$ are defined. The functors that relate the category of $CostPerf$ objects to the objects in the categories of performance and resource costs are identified by the projection operations π_{perf} and π_{cost} in Figure 4.55. The *Endofunctor* identified by the $min_{perf}\Delta sum_{cost}$ morphism is defined as the **Algebra pair** of the operations on performance and resource cost objects, which we explain in subsection 4.4.4. We can then define the *restrictSpace* evaluation morphism, which corresponds to an *Endofunctor* in the category of $Fix\ TermF$ objects that is sensitive to the performance and resource-cost.

4.4.4 Borrowing structure

In subsection 4.4.3 we identified the need to define an evaluation morphism for a *product object* in terms of the evaluation morphisms for its components. The solution to this need is not only helpful in describing an overall *cost-performance* evaluation in terms of separate *performance* and *resource-cost* morphisms, but also in relating all the various interpretations of *terms*, *types* and *cost-performance estimates* as the components of an overall *F-Algebra* over the polynomial functor $TermF$.

Polynomial *F-Algebras*

A pair of *F-Algebras* is the *binary product of F-Algebras*, meaning the *product object* in the category of F-Algebras. As with any product, we may recover the component algebras through the projection operations.

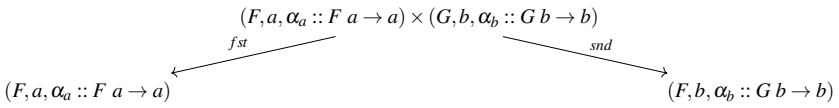


Figure 4.56: Algebra pairs.

The evaluation morphism for an *F-Algebra* pair is thus a higher-order function parametrized by the evaluation morphisms for its component *F-Algebras*.

```
appPair :: (F a -> a, F b -> b) -> (G a, G b) -> (a , b)
applyPolynomial :: ( H c -> c ) -> H c -> c
```

Listing 4.37: Application of F-Algebra Pairs and F-Algebra polynomials

Note that the functor on which the overall evaluation morphisms operates is neither the F nor the G component, but the functor H defined as the product $H = (F,G)$. In a *Cartesian closed category* of *F-Algebras* the idea of *F-Algebra* pairs can be generalized, giving us a way to construct a category of *F-Algebras* for any polynomial functor H. This follows trivially from the duality of sum and product objects.

The fact that we have replaced the two type parameters a and b by a single type variable c need not be a cause for alarm. If we are dealing with *F-Algebra pairs* then the new type parameter is simply understood to be the product object $c = (a,b)$. This can be generalized to any polynomial data constructor, not just pairs.

The Böhm and Berarducci Encoding

Recall that any TyTra CL AST can be represented as a value having the recursive data-type identified by the fixpoint of *TermF*, the polynomial functor shown in Listing 4.12 and that we can represent evaluations of such ASTs as *F-Algebras* over *TermF*. We will now compose the multiple evaluators we have covered in section 4.4 into a single, more efficient, *F-Algebra* using a typed equivalent of the Church Encoding, introduced by Böhm and Berarducci [BB85], and most helpfully explained by Kiselyov [Kis12]. We will from now on refer to this encoding as the **BB encoding**.

```
newtype TermBB = TermBB {
 unTermBB :: forall a.
  (String -> a )              -- varlit
  -> (  a -> a -> a )         -- app
  -> ( [a] -> a )               -- term tup
  -> a                            -- zipbb
  -> a                            -- unzip
  -> ( IndexListBB -> a )         -- stencil
  -> (  a -> a )                  -- map
  -> (  a -> a )                  -- fold
  -> (  NatBB -> a )              -- elt
  -> a
}
```

Listing 4.38: BB encoding for *TermF* from Listing 4.12

Listing 4.12 shows the BB encoding, in Haskell syntax, for the TyTra CL AST data-type. Notice there is a single data-constructor *TermBB* parametrized by a *unTermBB* polymorphic function. We can read this as saying that: given a function having the same type as *unTermBB*, our data-constructor yields a value of type *TermBB*. The *unTermBB* function is parametrized by a number of possibly constant polymorphic functions, all of which have an output type *a* that yields a value of type *a*. Each of *unTermBB*'s parameters are understood to be the TyTra CL AST data-constructors and thus, any function having the same type as *unTermBB* can be understood as one that selects a particular data-constructor and applies it to the required inputs. To make this more concrete, let us look at the data-constructor for TyTra CL variable terms.

```
termLit :: String -> TermBB
termLit name =
 TermBB $ \lit app ttup tzip tuzip tstencil tmap tfold telt -> lit name
```

Listing 4.39: TyTra CL Variable Term constructor (BB encoding).

The *termLit* data-constructor shown in Listing 4.39 parametrizes *TermBB* with an anonymous function, having the same type as *unTermBB*. This anonymous function applies its first parameter to the *String* input provided to *termLit*, yielding what must be a value of type *a*. A *tuple term* data-constructor called *termTup* can be likewise defined, as shown in Listing 4.40.

```
termTup :: [TermBB] -> TermBB
termTup fs =
 TermBB $ \lit app ttup tzip tuzip tstencil tmap tfold telt ->
  ttup (map
   ( \t -> unTermBB t lit app ttup tzip tuzip tstencil tmap tfold telt ) fs )
```

Listing 4.40: TyTra CL Tuple Term constructor (BB encoding).

The *termTup* constructor provides a function having the same type as *unTermBB* that selects and applies its third parameter to a list of sub-terms. The same technique can be used to define all TyTra CL term constructors. Notice that the recursive arguments to *termTup*, our tuple constructor, will be first evaluated through the recursive application of *unTermBB* before the tuple term is constructed. Every term constructor defined in this way can be understood to be a specification of how the constructed term will be evaluated *in the future* when an evaluation morphism for each data constructor is defined. At the moment of term construction, these evaluation morphisms are not yet defined, but are simply referred to by name. The data-constructors and the *TermBB* encoding define a *universal interpretation* of a TyTra CL Terms that all other interpretations must factor through. Recall that this was the exact definition of a data-type as the initial object in a category of *F-Algebras* for a given functor. The morphisms from the initial object to every other evaluation of *TermBB* can all be constructed through an appropriate application of *unTermBB*. Let us look at an example, that of pretty printing TyTra CL terms.

```
viewTerm :: TermBB -> String
viewTerm t = unTermBB t lit app ttup tzip tuzip tstencil tmap tfold telt
 where
  lit name = name
  app f x = f ++ " $( " ++ x ++ " )"
  ttup fs = "( " ++ (intercalate ", " fs) ++ ")"
  tzip = "zip "
  tuzip = "unzip "
  tstencil its = " stencil " ++ show its
  tmap a = "map " ++ a
  tfold a = "fold " ++ a
  telt n = " elt " ++ show n
```

Listing 4.41: TermBB Evaluator producing a String.

Through the *application* of *unTermBB* to a number of functions that determine the interpretation of TyTra CL Terms as Strings, we obtain a new interpreter. This interpreter is simply defined as the composition of *unTermBB* with the various functions that the data-constructors are mapped to.

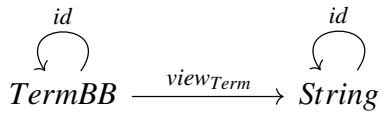


Figure 4.57: Defining an interpreter on TyTra CL terms as transformation on TermBB.

The interpreter defined by *viewTerm* in Listing 4.41 is an *F-Algebra* morphism that maps the *F-Algebra* object denoting the TyTra CL Term, to the *F-Algebra* object on Strings which represents is human-readable form. Let us see how this interpreter could be applied to a simple term: *termApp (termLit "foo") (termLit "bar")*. At the type level, this application is represented as shown in Figure 4.58 below.

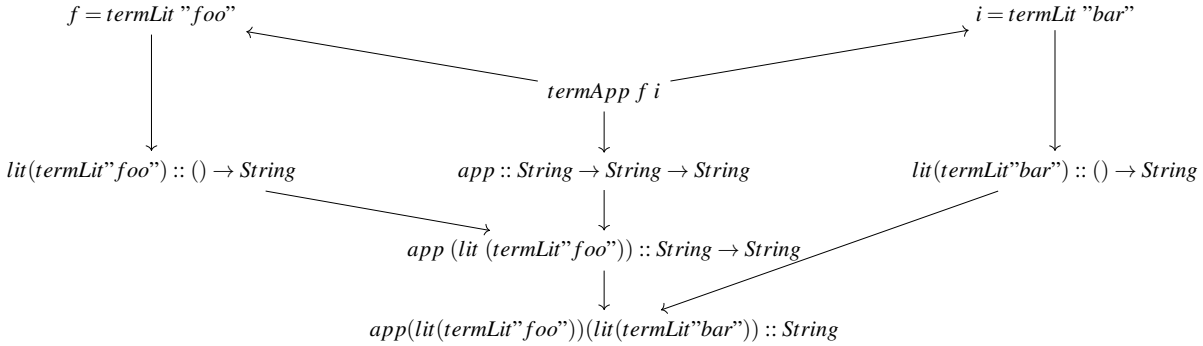


Figure 4.58: Type-level application of *viewTerm* to *termApp (termLit "foo") (termLit "bar")*.

In Figure 4.59 we show the term-level view of the same example. We notice that the composition of *unTermBB* with the individual functions defined in the scope of *viewTerm* is equivalent to an *F-Algebra* over *TermF* with a *String* carrier.

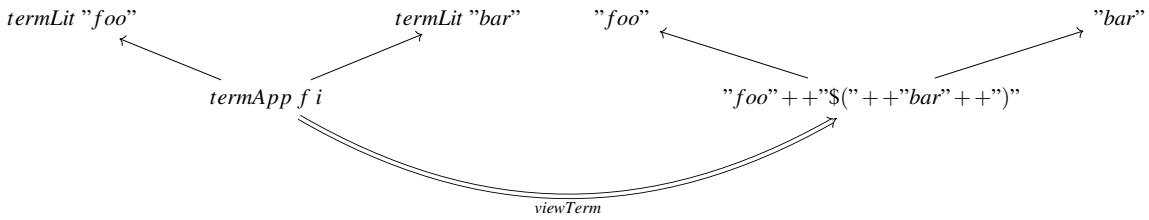


Figure 4.59: Term-level application of *viewTerm* to *termApp (termLit "foo") (termLit "bar")*.

Type inference with the BB Encoding

Having shown the simple example of pretty printing a TyTra CL term, let us now look at a more complex example: the type inference function that determines a term's type. We will first define the data-type of TyTra CL Types using the same BB Encoding as shown in Listing 4.42.

```
newtype TypeBB = TypeBB {unTypeBB :: forall a.
 (TVar -> a )                    -- type variable
 -> (String -> a)                -- literal
 -> ( a -> a -> a )              -- internal products
 -> (Tag -> Size -> a -> a)      -- sized vectors
 -> ([a] -> a)                   -- heterogenous lists / tuples
 -> (a -> a -> a)                -- function type
 -> a
}
```

Listing 4.42: BB encoding for TyTra CL Types.

Data constructors for *TypeBB* values are defined in the same way as *TermBB* data-constructors, by selecting the appropriate evaluation morphism. This is shown below in Listing 4.43.

```
tyvarbb :: TVar -> TypeBB
tyvarbb var = TypeBB $ \ tyvar tycon typrod tyvec tytup tyfun -> tyvar var

tyconbb :: String -> TypeBB
tyconbb name = TypeBB $ \tyvar tycon typrod tyvec tytup tyfun -> tycon name

prodbb :: TypeBB -> TypeBB -> TypeBB
prodbb a b = TypeBB $ \tyvar tycon typrod tyvec tytup tyfun -> typrod
  (unTypeBB a tyvar tycon typrod tyvec tytup tyfun)
  (unTypeBB b tyvar tycon typrod tyvec tytup tyfun)
```

Listing 4.43: Data constructors for TyTra CL Types using the BB Encoding.

The three data-constructors shown in Listing 4.43 correspond to *type variables*, *type literals* and *product types*. Type variables are needed to represent types which have not yet been fully determined. Type literals correspond to the type-level assignments in the TyTra CL representation of an application. The product types are used to as a more granular representation for tuples and vector types. Using *TypeBB* we can now rephrase the constraint-solving approach to type-inference and checking functionality described in subsection 4.4.1 as a morphism defined by composing *unTermBB* with a number of new interpreters for each TyTra CL Term constructor that generate and solve constraint sets.

Constraint generation requires that we apply *unTermBB* to the *TermBB* encoding of a TyTra CL Term and define a number of constraint generators for each data-constructor, as shown in Listing 4.44. The output type of the *genConstraints* evaluator indicates that this is a monadic computation in the *TypeCheck* context. This is needed to generate *fresh* type variable symbols and provide better error reporting in the compiler but can be safely ignored in the context of this presentation. We will instead focus on the output type $(TermBB, TypeBB, Subst)$.

```
genConstraints :: TermBB -> TypeCheck TermBB (TermBB, TypeBB, Subst)
genConstraints term = do
 (termTerm, termTy , termSubst) <- unTermBB term
  gcLit  gcApp gcTup gcZip gcUnZip gcStencil gcMap gcFold gcElt
 return (term, termTy , termSubst )
```

Listing 4.44: Type inference using BB Encoding

The first component is simply the identity morphism on the term structure. The second component is a *TypeBB* expression that represents the inferred type which may contain undetermined *type variables*. The third and final component of type *Subst* is a collection of substitutions that must be applied to fully determine a TyTra CL Type.

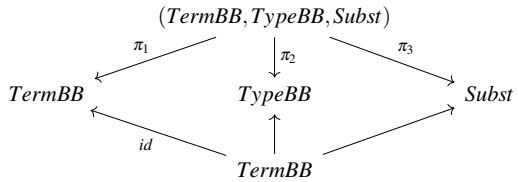


Figure 4.60: Term-level: Application of *viewTerm* to *termApp (termLit "foo") (termLit "bar")*.

The evaluation morphisms provided as parameters to *unTermBB* must all have the same output type. Let us look at *gcLit* which generates the partially resolved *TypeBB* representation for variable terms.

```
gcLit  :: String -> TypeCheck TermBB (TermBB, TypeBB, Subst)
gcLit  name = do
 env <- ask
  case M.lookup name env of
   Nothing -> do
    tv  <- freshTyVar
    return (termLit  name, tyvarbb tv,  mempty)
   Just  ty  ->
  return (termLit  name, ty,  mempty)
```

Listing 4.45: Type inference using BB Encoding.

Inferring the type for a TyTra CL Term variable as described in Listing 4.45 implies a lookup in the type-context for that variable name. If a type definition exists for the variable name in question, then this is simply returned, along with the term expression and an empty set of type-level substitutions. If the variable name is not found in the type context, then it is assigned a fresh variable symbol and likewise returned. The type inference evaluators for the other term-constructors are more complicated, as we can see for *gcMap* in Listing 4.46 below.

```
gcMap action = do
 (aTerm, aType, aSubs ) <- action
 tvi  <- freshTyVar
 tvo  <- freshTyVar
 sv   <- freshSzVar
 let
  actGuess = funbb (tyvarbb  tvi )  (tyvarbb tvo)
  actSubs   = aSubs <> (fromMaybe mempty $ mgu actGuess aType)
  mGuess   = apply actSubs $ funbb
   (vecbb (SizeVar sv) (tyvarbb  tvi ))
   (vecbb (SizeVar sv) (tyvarbb tvo))
 return (termMap aTerm, mGuess , actSubs <> aSubs )
```

Listing 4.46: Type inference using BB Encoding.

In the case of *map* terms we can *guess* that the output type will be that of a function on vector types. In Listing 4.46 we see that we first create two fresh type variables and one fresh size variable. These correspond to the parameters and size-index required by the input and output vector types. We likewise assume that the map term's sub-term will be a function type from the fresh input type variable, to the fresh output type variable. The *mgu* function that appears in Listing 4.46 is an implementation of a *most-general unifier* that simply computes the most general set of substitutes required to match our *guessed type scheme* to the actual type inferred for the map term's parameter.

```
mgu :: TypeBB -> TypeBB -> Maybe Subst
```

Listing 4.47: Most general unifier for type inference (function type).

Any substitutions in the context are then applied to the *guessed type schema*. Concrete types replace variables where these can be determined, yielding a simplified *TypeBB* representation. If the overall *TermBB* value for which the type inference evaluation is being performed is well typed. in the supplied context, then the output *TypeBB* representation will contain no type variables.

Recall that the overall type-inference evaluation defined in Listing 4.44 computes three output values at once. In addition to producing a type for the given input term, it also returns the input term itself and a set of substitutions. From a categorical perspective, this is because the *unTermBB* evaluation is the initial object in a category with product objects. We can use this to work around the issue of having to run all three phases of our DSE strategy to trim off inefficient solutions using the balanced pipeline heuristic. Recall that there exists a dependence between the three DSE stages. From the term representation we infer the *types*. From these *types* we can derive the costs. Lastly, from the *terms, types and costs* we derive a collection of program transformations. In a category of TyTra CL term interpreters with product objects, as we have constructed and now shown in Figure 4.61, there must exist a unique object that is the product of these interpreters.

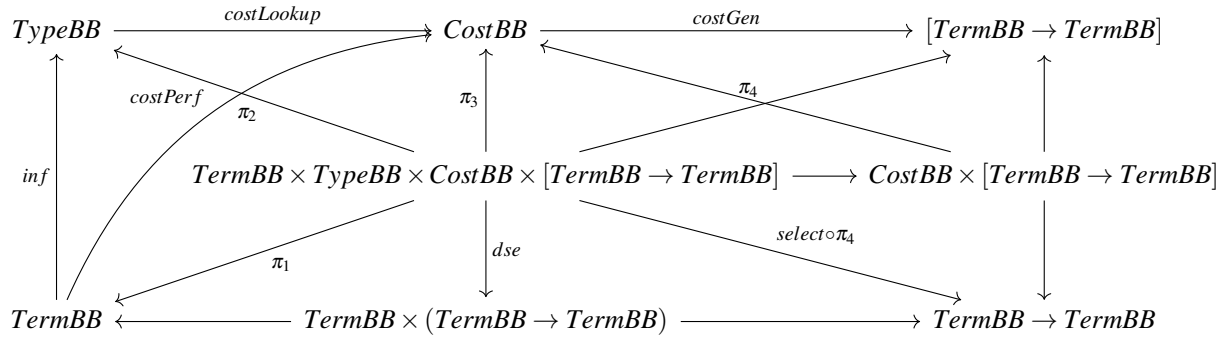


Figure 4.61: The three DSE stages are dependent.

The three DSE stages are not just dependent on one another, but are *fragments* of a larger function that computes all three stages at once. As *non-leaf* terms are constructed from sub-terms, the product object of evaluators is really defined by some operation over the product objects that correspond to the sub-terms. Here, the *CostBB* object is the data-type shown in Listing 4.48 which describes how the overall cost-performance estimate for a term is constructed from the estimates of its sub-terms.

```
newtype CostBB = CostBB {
 unCostBB :: forall a.
   ((PerformanceKI, ResourceKI) -> a)
   -> (a -> a -> a)
   -> a
}
costPerfLit :: (PerformanceKI, ResourceKI) -> CostBB
costPerfLit costPerf = CostBB $ \_lit _merge -> _lit costPerf
costPerfMergeL :: CostBB -> CostBB -> CostBB
costPerfMergeL cpA cpB = CostBB $ \lit_ merge_ -> merge_ a b
```

Listing 4.48: BB-encoded cost-performance model.

Structural Cost-Performance Model

This data-type has two constructors. The *costPerfLit* constructor takes cost-performance estimates for opaque functions directly from the back-end cost-performance model. The *costPerfMergeL* constructor simply states that the cost-performance estimate for a term is computed from the estimates of its sub-terms, but does not specify *how* it is computed. Using the cost-performance data-type and the data-constructors shown in defined in Listing 4.48 a cost-performance evaluation morphism $costPerf :: TermBB \rightarrow CostBB$ that produces *cost-performance estimates* for every term. We know that such a morphism *must* exist because a type inference morphism $inf :: TermBB \rightarrow TypeBB$ and a cost lookup morphism $costLookup :: typeBB \rightarrow CostBB$ exist and compose. Their composition gives us $costPerf = costLookup \circ inf$, shown in Listing 4.49 below.

```
costPerf :: TermBB -> CostEnv -> CostBB
costPerf term costEnv = unTermBB term lit app ttup  [..]
where
  lit  name = case (Map.lookup name costEnv) of
   Just cst -> costPerfLit cst
 app f x = costPerfMergeL f x
 [..]
```

Listing 4.49: Abstract cost-performance evaluator.

The *costPerf* evaluation morphism can be understood as an *abstract interpreter* it replaces, or rather, **composes** a *TermBB* constructor with a *CostBB* constructor. Both of these are higher-order functions that must be parametrized by first-order functions before they can be fully evaluated. Their composition is then also a higher-order function that performs the work of both. In Figure 4.62 below, we show how an application term having two sub-terms: a function foo and a variable *input* onto which foo is applied; would be abstractly interpreted using *costPerf*.

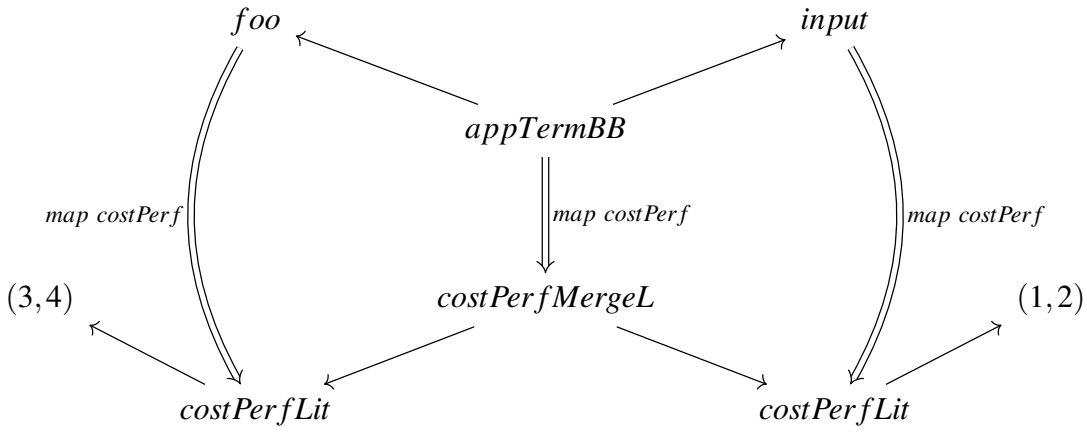


Figure 4.62: Homomorphism from *TermBB* to *CostBB*.

Notice that both *viewTerm* and *costPerf* are Catamorphisms over the term structure. Recall that *Catamorphisms* are generalized folds and that certain folds can be fissioned. If we view the definition of the *costPerf* function shown in Listing 4.49 through the lens of Equation 4.2 that describes fold fissioning, we can that *costPerf* must be composed of a primitive fold operation, and a *map* operation. The *map operation* is represented by the entire **where block** in Listing 4.49 which substitutes *TermBB* constructors for *CostBB* constructors as shown in Figure 4.62. The primitive fold side of the fissioned *costPerf* function, we will have to perform the actual merge operation to produce the total cost-performance estimate.

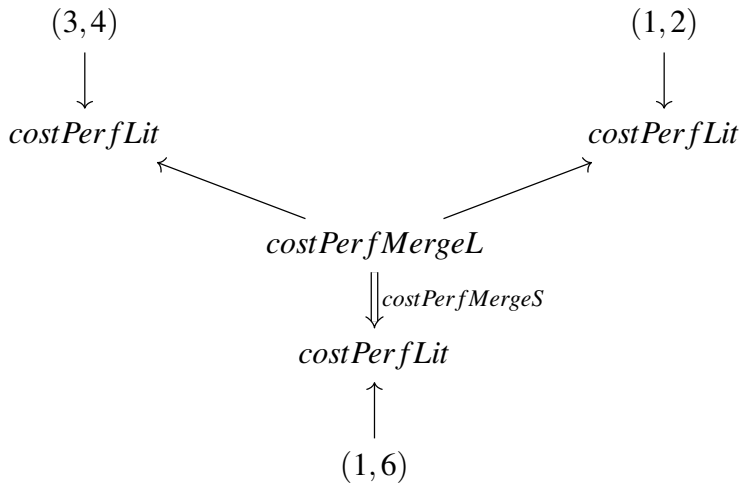


Figure 4.63: Strict CostBB merge operation.

To compute the concrete *cost-performance estimate* we will need a **strict** implementation of the merge operation, shown in Listing 4.48 below. We can see that this is also a homomorphism as it simply maps one *CostBB* constructor to another.

```
costPerfMergeS :: CostBB -> CostBB -> CostBB
costPerfMergeS ac bc = costPerfLit  $ foldCostBB $ costPerfMergeL ac bc
```

Listing 4.50: BB-encoded cost-performance model.

The strict *merge* implementation is only needed in the later stages of DSE. Recall that in Figure 4.61 we had shown a morphism $costGen :: CostBB \rightarrow [TermBB \rightarrow TermBB]$. This morphism generates a design space corresponding to a term based on its concrete *cost-performance estimates*. What is not shown in Figure 4.61 is that this is morphism is one part of a larger morphism. In fact, the complete picture is quite a bit more nuanced. We can however see how the BB encoding, together with the category-theoretical semantics we defined for the TyTra compiler allow us to easily compose/transform *interpreters* and thus the DSE process. What follows in subsection 4.4.5 is the result of putting these categorical semantics together.

4.4.5 Fused DSE

Our DSE strategy is a function that traverses the TyTra CL term structure from the inner-most sub-terms to the root, or overall term that defines the application. As it pattern-matches on the structure of the term being optimized, **a function that computes the *cost-performance* estimate** for every sub-term is generated. This requires access to a *cost-performance* estimate environment with the type *CostEnv*. As program variants that exceed the resources available on the target FPGAs must be opportunistically removed, a measure of the resource bounds is given by a value of type *Board*. The type signature for our DSE strategy, accounting for all of these input parameters is shown in Listing 4.51.

```
fusedFilterGenVariants :: TermBB -> CostEnv -> Board
 -> (TermBB, [(Ratio Integer, ResourceKI)])
```

Listing 4.51: Fused/optimized DSE implementation (signature).

The output type of this function is a pair, containing the input structure, as well as an ordered list of *performance ratios* and estimated resource uses for each of the program variants that can be generated. In the compiler implementation, there is a third component that contains the transformation function, which would generate a program variant with the performance and resource use indicated by the other two components. To keep this presentation manageable, we elide this third component, but note that it is simply defined as a lazy composition of the transformations on the sub-terms. The term-level pattern-matching functionality is given directly by the *unTermBB* which defines the TyTra CL term destructor in the *BB* encoding. The *unTermBB* destructor is a higher-order function, parametrized by a number of functions which are used to replace the data-constructors which make up the structure of the term being compiled, as shown in Listing 4.52.

```
fusedFilterGenVariants term costEnv board =
 unTermBB term lit app ttup leaf  leaf  leaf  tmap tfold  leaf
 where  [...]
```

Listing 4.52: Fused/optimized DSE implementation calls unTermBB.

TyTra CL Term constructors that are not recursively defined, aside from the variable terms that could represent either input variables or opaque functions, are simply replaced with the *leaf* constant. It is assumed that such expressions have no impact on performance, and that they require a constant amount of resources. In the compiler implementation these costs are different between say *Stencil* terms and *Elt* terms, however this does not affect the soundness of our result.

```
 [...]
leaf  = (term ,  [(1,  nullRKI )])
```

Listing 4.53: Fused / Optimised DSE Implementation

Input variable terms are likewise assumed to have no effect on performance and a negligible resource use. In the compiler we would model the size of the required data transfers from main memory and used that as an additional component of the hardware resource use estimates. We can tell input variables apart from opaque functions simply by checking the cost-performance estimate context, which only contains entries for opaque functions.

```
lit name =
 case (Map.lookup name costEnv) of
  Nothing -> (term , [(1, nullRKI )])
  Just cst ->
   let
    efiVal = efi . fst $ cst
    rkiVal = snd cst
    factors = [1..( efi . fst $ cst )]
    res = (term , filter (\ t -> snd t <= bound)
     $ zip
      ( map (\i -> (fromInteger i) % fromIntegral efiVal ) factors )
      ( map (\f -> smulRKI f rkiVal) factors )
     )
   in res
```

Listing 4.54: Design-space generating anamorphism replaces variable constructors.

A successful lookup for opaque functions returns the $(PerformanceKI, ResourceKI)$ tuple. The first component is use to generate a bounded number of program variants that parallelise map operations up to a degree equal to the function's input-to-output latency denoted by the efi component of the $PerfromanceKI$ value. Program variants that take up more resources than available are immediately filtered out. The *merge* operation discussed in section 4.3 is implemented as *mixDesigns* shown in Listing 4.55. Note that we make use of the fact that sub-term generated search-spaces are ordered in ascending order of performance.

```
mixDesigns b varF varX =
 let
  fMap = Map.fromAscListWith min varF
  xMap = Map.fromAscListWith min varX
  goMix f x =
   map (\ t -> (t , head $ filter (\s -> fst s >= fst t) (Map.toList x))) $ Map.toList f
 in
  map ( \(( i ,iC ),( j , jC)) -> ( min i j , addRKI iC jC ) )
  $
   if length varF >= length varX then goMix xMap fMap else goMix fMap xMap
```

Listing 4.55: The *merge* operation on sub-spaces.

The *lit* function that replaces variable terms shown in Listing 4.54 is an Anamorphism, as it generates an ordered list of *cost-performance* estimates from each variable term. The *app* and *ttup* functions in Listing 4.56 are Catamorphisms that generate an overall search-space from a pair of sub-term generated spaces, in the case of *app* which replaces application terms, and a $k-ary$ tuple of sub-spaces in the case of *ttup* which replaces tuple constructors.

```
app (_, varF) (_, varX) = (term ,  filter (\t -> snd t <= bound)
 $ mixDesigns bound varF varX )

ttup xs = (term, filter (\t -> snd t <= bound)
 $ foldr1 (mixDesigns bound) $ map snd xs)
```

Listing 4.56: Applicaion and Tuple terms mix their sub-term generates search-spaces.

Notice that the *mixDesigns* function in Listing 4.55 inspects each of the sub-term generated search-spaces and merges them in order, from the smallest to the largest (last line). Finally, *tmap* and *tfold* simply pass along the sub-space generated by their respective recursive arguments, and are thus simply search-space homomorphisms.

```
tmap a = (term, snd a)
tfold a = (term, snd a)
```

Listing 4.57: Map and fold search-spaces are generated entirely by their recursive argument.

Once the entire *fusedFilterGenVariants* function is applied to a *BB-encoded* TyTra CL term value of type *TermBB*, its first argument, the *unTermBB* higher-order function is parametrized by the functions we have just presented. This yields a new, first-order function that takes the second argument, the cost-performance estimate environment of type *CostEnv* and returns another function which, when given a description of the hardware target, would generate the search space. Because we have not yet supplied all required arguments, none of *fusedFilterGenVariants*'s output can yet be determined, and so no computation is executed. Once the third and final argument of type *Board* is supplied, the composition of Anamorphisms, Catamorphisms and Homomorphisms that make up *fusedFilterGenVariants* is optimized by the language run-time. We know that the composition of an Anamorphism (the *lit* function), with a Catamorphisms (the *app* and *ttup* functions) yields a hylomorphism which is optimized in the sense that it does not produce intermediary values that are not required by the output. Each of the instances where we see a call to the *filter* function in these Catamorphisms can be seen as being propagated to the closest place where it can be fused with the *lit* Anamorphism, meaning that *fusedFilterGenVariants* generates the minimum size of search-space required. In section 4.5 that follows next, we'll present this result as a formal theorem and proof of efficiency for the DSE strategy we have just defined.

4.5 Efficient DSE Theorem

Having given *categorical semantics* to the TyTra CL, its dependent type system, and the *design-space exploration process*, we can now state and prove our main theorem.

Theorem 1. *Given an arbitrary term t in the TyTra Coordination language, a type context* Γ *under which t is well typed, a set of* accurate cost-performance estimates *for every* opaque function *occurring in t and a* bounding measure *of* hardware resources *for a target device, the* globally optimum parallel implementation *of t that fits the target device can be found in* polynomial time.

4.5.1 Proof

Recall the presentation of our DSE strategy in Listing 4.52 which shows that search-space construction involves applying the specific *unTermBB* function defined by the outer term of the application being compiled, parametrized by a number of search-space generating functions, each corresponding to every possible term-level constructor. The *TermBB* encoding of the application will have been constructed by nesting functions such as *termLit* from Listing 4.39 and *termTup* from Listing 4.40.

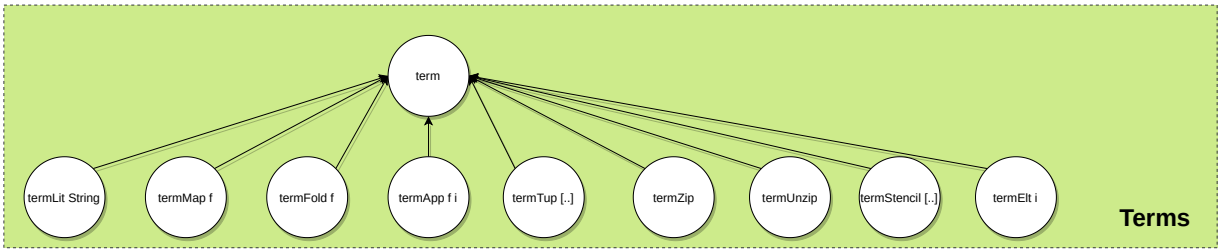


Figure 4.64: Case-splitting on the *TermBB* encoding of the application.

The *TermBB* constructors are all shown in Figure 4.64 and can be categorized as either recursive or non-recursive functions. For example, the *termLit* constructor shown in Figure 4.64 is non-recursive because all of its arguments have a type different from *TermBB*. In Listing 4.38 we can see that the same is true for *termZip*, *termUnzip*, *termStencil* and *termElt*. The remaining constructors: *termMap*, *termFold*, *termApp* and *termTup*; are all recursive constructors meaning that calling *unTermBB* on the terms produced by these constructors will involve further nested calls to *unTermBB* on their nested sub-terms, as we have shown in Listing 4.40 for tuple terms.

We can give a proof by structural induction on our *TermBB* representation. The non-recursive term-constructors represent the base case whilst the recursive constructors make up the inductive case.

Proof of Theorem 1 . By induction on the structure of the *TermBB* representation.

Base case. In Listing 4.54 we can see that the *termLit* constructor is replaced by a design space generating function. If the term in question corresponds to an input variable, the generator will produce a single design point. The same is true for all non-recursive constructors. If on the other hand the term corresponds to a function type, then the space generating function can produce as many design points as would be required to overcome the *effective firing rate* of the opaque function in question.

```
factors = [1..( efi . fst $ cst )]
```

This design space generating function is pre-composed with a filter that ensures only program variants that would fit on the target device will be generated. It must be stressed that at this point, no program variants or cost-performance estimates are actually generated. We've only created the function that *would* produce them.

```
res = (term , filter (\t -> snd t <= bound) $ zip
  ( map (\i -> (fromInteger i) % fromIntegral efiVal ) factors )
  ( map (\f -> smulRKI f rkiVal) factors ) )
```

Induction case. Recursive constructors are those for map, fold, tuple and application terms which all contain *TermBB* sub-terms. Map and fold terms include a single sub-term each. The design space they generate has the same number of design points as their respective sub-terms. Tuple terms constructed with *termTup*, as well as application terms built using *termApp* contain multiple *TermBB* sub-terms. Calling *untermBB* on these sub-term constructors yields a design-space generating function for each. These functions are composed using the *mixDesigns* function as exemplified in Listing 4.56 and shown in the orange section of Figure 4.65.

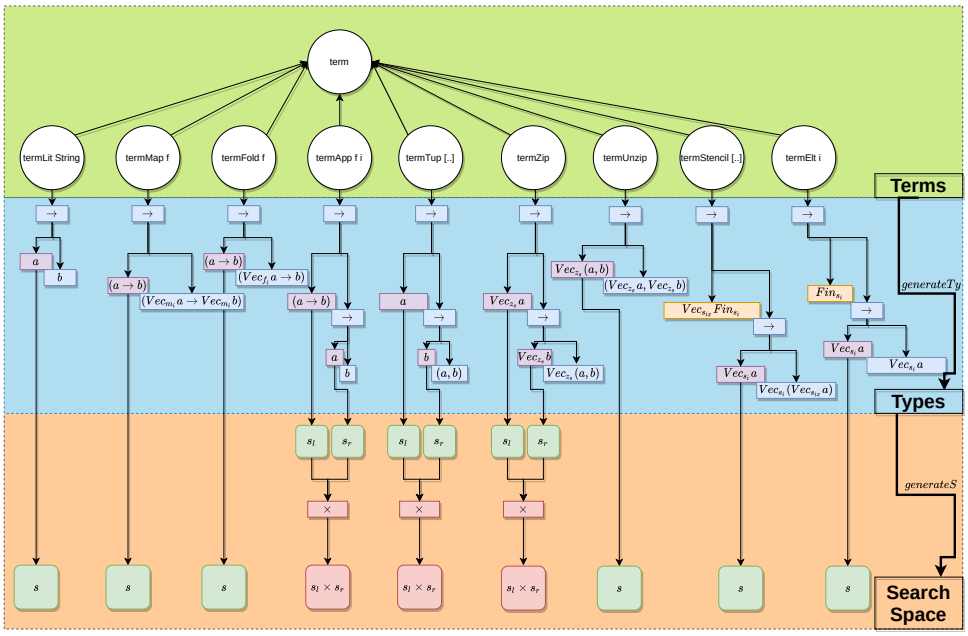


Figure 4.65: Graphical summary for the proof of Theorem 1.

In the naive implementation this would have been implemented as taking the *cross-product* of sub-spaces, meaning that there would have been an exponential blow-up in the number of solutions. While the search space generated by the *naive* strategy is indeed complete, it is also particularly large. If every $TermAppF$, $TermTupF$ or $TermZipF$ sub-term produces a search-space by taking the *cross-product* of their respective sub-terms' search spaces, then the overall search-space is clearly exponential due to the simple algebraic identity that relates products to exponentials: $s_1 \times s_2 \ldots \times s_k = s^k$. In our optimized DSE strategy, the cross-product operation is replaced with the mixing function shown in Listing 4.55. The *mixDesigns* function that replaces the cross-product generates a design space having a $s_1 + s2$ cardinality where s_1 and s_2 are the cardinalities of the sub-term search-spaces by pre-composing the space-generating functions with a filter that ensures program variants are only generated if their throughput can be achieved in practice, in a balanced pipeline. This filter is the *goMix* helper function below.

```
goMix f x =
 map (\t -> (t , head $ filter (\s -> fst s >= fst t) (Map.toList x))) $ Map.toList f
```

The *goMix* helper traverses the smallest of the two sub-spaces to determine the quantized levels of achievable throughput and selects a program variant that archives an equal or higher throughput for the minimum cost, from the larger sub-space as shown below.

```
map ( \(( i ,iC ),( j , jC)) -> ( min i j , addRKI iC jC ) ) $
 if length varF >= length varX then goMix xMap fMap else goMix fMap xMap
```

In Figure 4.66 we can see that the output of the *mixDesigns* function has a size that is at most the sum of the input design-space sizes.

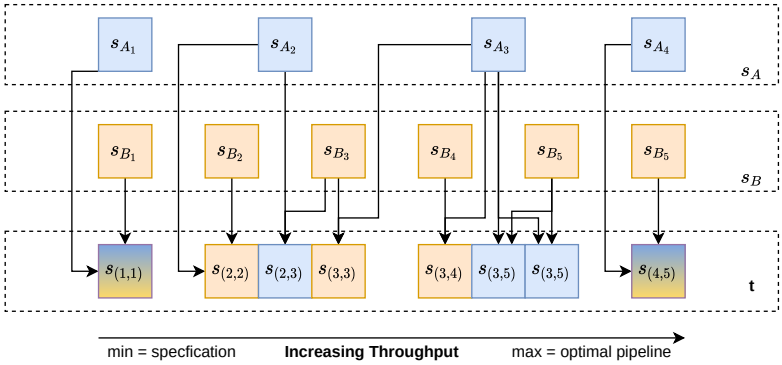


Figure 4.66: Additive sub-space mixing.

Given that the base-case generates a linear design-space for non-recursive terms, and that the inductive case shows all such spaces produce additive compounds spaces, we have now shown that we can *efficiently* generate a *complete* design-space using a *polynomial-time* function by opportunistically pruning design points that can not be implemented within the resource constraints as well as those that consume more resources to deliver the same performance as other candidate solutions. □

Chapter 5

Experimental Evaluation, Conclusion & Future Work

Through this work we have sought to show that *design space exploration* can be an *effective and efficient* way to derive *application optimization schedules*. This we have indeed achieved, as witnessed by the proof in section 4.5, however, this theoretical result may be rightfully attacked with the observation that: *Given sufficiently large constants, a polynomial-time algorithm may be of little practical use*. To dispel this possibility, we will present experimental results that show our contributed DSE strategy to be of practical utility, meaning that the constant factors which define the polynomial-time complexity are suitably small.

The experimental results shown in section 5.1 are related to the optimization of a number of representative examples of applications from the domain of scientific high-performance computing. Part of the data used in this section is derived from work produced in collaboration, as part of the following publication: Cristian Urlea, Wim Vanderbauwhede, and Syed Waqar Nabi. Efficient FPGA cost-performance space exploration using type-driven program transformations. In David Andrews, René Cumplido, Claudia Feregrino, and Marco Platzner, editors, *2019 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019*, pages 1–2. IEEE, 2019. Figure 5.2 shows experimental results produced in joint unpublished work with Syed Waqar Nabi.

In section 5.2, we will give our concluding remarks regarding design space exploration in the TyTra compiler. Finally, in section 5.3, we speculate on a number of possible future directions for expanding this work.

5.1 Experimental validation

Compiler developers often give empirical evidence of *effectiveness* and *efficiency* by running their compilers on **benchmark applications**. This approach is problematic because *what exactly constitutes a representative application* is a matter of debate. Optimizing transformations imply a *trade-off* between certain classes of properties. When applications gain in *execution speed* they may also require an equivalent increase in the *resource utilization*. Structurally different applications may imply different trade-offs in performance and resource use. Although *benchmarking* may not conclusively prove such properties as *effectiveness* or *efficiency* by itself, *benchmarking* can be used validate a formal proof that such properties hold, as we have given in section 4.5. To this end, we ran our design space exploration strategies on a number of examples, derived from real-world scientific and numerical applications. Note that the **TyTra CL representation** for these examples only denotes the application's *data-flow* structure. The implementation details pertaining to the *opaque functions* are only represented in the TyTra IR which is not shown. For layout reasons, the example in Listing 5.1 also elides the the type declarations within the **TyTra CL**. This means that the *opaque function signatures*, such as *shapiro :: Float → SVec 5 (Float, Float) → Float*, an the *input*, *intermediate* or *output* variable type annotations, such as structures *hzero :: Vec 500 Float*, are not shown. The concrete point of this exercise is to show:

1. That our polynomial-time design-space exploration strategy runs in a practically small amount of time, for a set of representative applications. If this is indeed the case, then the *constant time factors* in this polynomial-time strategy will have been shown to be small. The practicality of the search strategy is shown for *any* TyTra CL application.

2. The ratio between the *explored search-space* and the *exhaustive search-space* validates the theoretical result that the optimized search space for a term t is an additive function of the search spaces for all sub-terms $s \in t$, rather than a multiplicative one.

The example applications chosen are the computationally intensive sections from: a large eddy weather simulator [NTN12]; an ocean model [Käm09]; a synthetic example derived from the latter. The TyTra CL representation for the second example is shown in Listing 5.1. At first glance, the structure and size of this example may appear to indicate that it is a trivial, but this is not the case. The examples used to validate our result are structurally complete. They showcase every language feature currently available to TyTra CL applications. In addition to the basic language constructs such as *let* expressions and the appearance of input and opaque function variable terms, the examples also make use of: higher-order operations on *opaque functions*: *map* and *fold*; data-container restructuring operations: *zip* and *unzip*; *stencil*.

Another reason why these example seem trivial is that the TyTra CL language, which at heart is effectively a dependently-typed sub-set of Haskell. It has a concrete syntax with a much higher density of computational constructs than the imperative languages commonly supported by vendor-supported HDL and HLS solutions.

```
vout =
let
eta_stencil = stencil [0,500] eta
wet_stencil = stencil [0,1,500] wet
un_vn = map (create_un_vn g) (zipt (eta_stencil, wet_stencil, u, v))
(un,vn) = unzipt un_vn
un_stencil = stencil [−1,0] un
vn_stencil = stencil [−500,0] vn
un_vn_stencil = zipt (un_stencil, vn_stencil)
hs1 = [−500,−1,0,1,500]
h_stencil = stencil hs1 h
etan = map (sea_level_predictor dx dy dt ) ( zipt (un_vn_stencil, h_stencil))
wet_etan = zipt (wet,etan)
wet_etan_stencil = stencil [−500,−1,0,1,500] wet_etan
eta' = map (shapiro eps) wet_etan_stencil
h_u_v_wet1 = map (upd1 dt dx dy eps g hmin) (zipt (eta', hzero, un, vn))
h_u_v_wet2 = map (upd2 dt dx dy eps g hmin) h_u_v_wet_1
h_u_v_wet3 = map (upd3 dt dx dy eps g hmin) h_u_v_wet_2
(h', u', v', wet') = unzipt h_u_v_wet3
in
zipt (h', u', v', wet')
```

Listing 5.1: Example TyTra CL Application (No Types)

Having broadly described the examples, we now look at the target hardware devices. In Table 5.1 we show the target FPGA model names and their hardware resource counts.

FPGA	***Slices***	***Logic Cells***	***Block Ram***	***DSPs***
XC6SLX4	600	3840	12	8
XC6SLX9	1430	9152	32	16
XC6SLX16	2278	14579	38	32
XC6SLX25	3758	24051	52	38
XC6SLX45	6822	43661	116	58
XC6SLX150T	23038	147443	268	180

Table 5.1: FPGA Resource Bounds.

The target FPGA devices listed in Table 5.1 showcase a wide spectrum of available hardware computational resources. Each hardware target represents a particular *Xillinx FPGA chip* from the vendor's commercial offering at the time of evaluation. All devices considered belong to the same *product line*, meaning that the internal structure of these devices is identical thus estimated performance can only be influenced by computational resource availability, rather than other architectural considerations. We performed one *design-space exploration* run for each example-target pair, yielding the results shown in Table 5.2 below.

Example	**FPGA**	**Perf**	**reg**	bram	dsp	lut	vars	dse ratio	speedup
SOR	*baseline*	0.017	2432	3	40	4631	0	0	1x
SOR	XC6SLX4	0.031	3136	6	44	5473	8	0.001	1.8x
SOR	XC6SLX9	0.218	21248	39	304	37469	19	0.01	12x
SOR	XC6SLX16	0.375	36224	66	520	63992	33	0.02	22x
SOR	XC6SLX25	0.625	59904	108	864	106092	55	0.03	36.7x
SOR	**XC6SLX45**	1.0	95424	171	1380	169242	88	0.05	58.0x
1DS	*baseline*	0.043	64	0	4	0	0	0	1x
1DS	**XC6SLX4**	1.0	1472	0	92	0	23	1.0	23x
Synth	*baseline*	0.037	1388	7	13	3380	9	0	1x
Synth	XC6SLX4	0.26	8520	42	82	21122	13	0.02	7x
Synth	XC6SLX9	0.69	22496	112	214	55343	34	0.05	18x
Synth	**XC6SLX16**	1.0	32308	161	307	79424	49	0.07	27x

Table 5.2: Best performing (example, board) pairs showing achieved/theoretical maximum throughput, hardware resource use and ratio of explored design space.

The *first column*, labelled **Example** indicates which of the three applications these results belong to. These are separated by the horizontal border. The *second column* indicates the target hardware device which sets the hardware resource limits. Notice that the target devices are ordered from top to bottom in order of increasing hardware resource availability. We only show those FPGA targets that yield a performance that is less than or equal to the theoretical maximum. Targeting a larger device would have no impact on the application performance nor the efficiency of the DSE process, so there is no point in showing these. As a pair, the *first and second columns* define a complete yet bounded search-space. The limits are set by the structure and types of the application, the hardware resource limits pertaining to the selected device and the cost-performance estimates for the application's opaque functions. The *third column*, labelled **Perf**, indicates the ratio between the throughput achieved by the best performing program variant found, and the theoretical maximum throughput for the application in question, as could be achieved on the targeted FPGA device. *Columns four through seven*, labelled **reg, bram, dsp and lut** represent the hardware resource usage of the best performing program variant in the search-space. The *eighth column*, labelled **dse ratio** shows what proportion of the *exhaustive search-space* our solution searched until the *best performing* program variant, that fits within the device's resource limits, was found.

The *ninth and final column*, labelled **speed-up**, indicates the ration between the forecasted throughput of the best performing program variant found, and the performance of the initial program variant from which the search had started.

In Figure 5.1 we show the overall and the pruned design spaces produced for the *Synth* example when targeting the largest device: *XC6SLX150T*. As we previously mentioned, Table 5.2 shows only those example-device pairs that exhibit a throughput that is less than or equal to the theoretical maximum. A close inspection of Figure 5.1 will reveal that the search-space produced by the *(Synth,XC6SLX150T)* example-target pair is identical to the largest space shown in Table 5.2 that corresponds to the *(Synth,XC6SLX16)* pair.

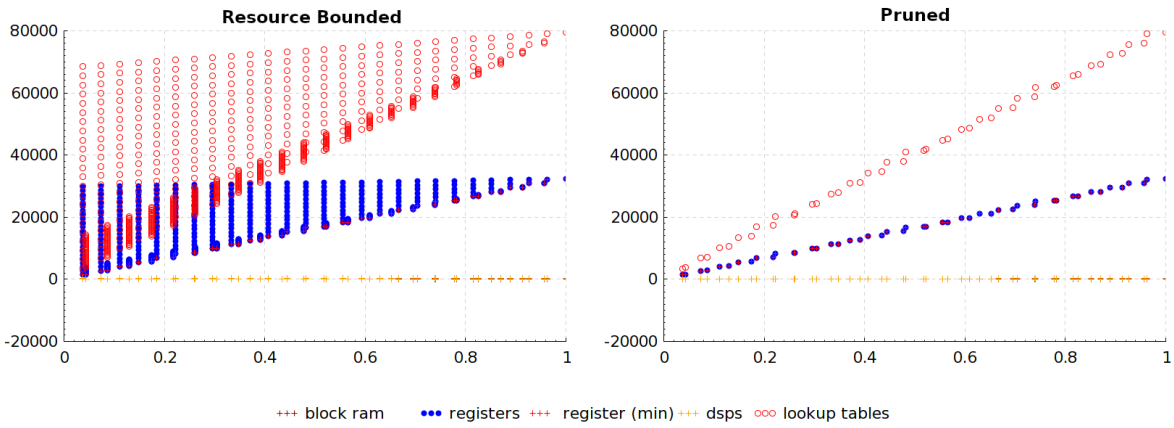


Figure 5.1: Hardware resource limit bounded Design Space (left) vs Filtered Design Space (right) for Synth Kernel on XC6SLX150T.

In both plots, the **X-Axis** corresponds to the *performance ratio* between individual design points and the *optimal program variant* that corresponds to a single instance of a *fully balanced pipeline* implementation. At each performance point on th X-Axis in the left-hand plot, we find an entire set of design points. Each design point is represented by multiple symbols, each of which corresponds to one of the hardware resource types. The left-hand side of Figure 5.1 shows that section of the *exhaustive search-space* that fits within the *hardware resource bounds* given by the target device, XC6SLX150T. Note that this is already significantly smaller than the space produced by the *naïve approach*. Design points that exceed the resources available on a *XC6SLX150T* FPGA were pruned at the earliest opportunity to generate the plot in a reasonable time-frame. The right-hand plot shows the search-space generated by our most efficient DSE strategy. It can be readily observed that the search-space contains a single solution for each discrete performance value: the hardware resource cost solution. The **Y-Axis** shows the **amount** of a hardware resource, that used by each design point.

The shape of the plots correspond to our expectations. The right-most plot marks out the *pareto-optimal frontier of the total design space*, generated by this particular example on the largest device *XC6SLX150T*. As these plots show *design points* through multiple symbols, it may be difficult to observe that the optimized search space is now defined by an additive function. Let us take a further example and verify that the number of design points is in line with our theorem. This last example is an application from the realm of high-performance computing: the 2D Shallow-Water Model derived from an ocean model [Käm09] and focused on a particular part of this example application that can be roughly described as a sequence of three *map* operations, each with its own *opaque function* implementation, collectively acting upon a *vector* of input values. We denote the opaque functions as f, g and h respectively. For each of these, we show the *effective firing interval*, the expressed parallelism as well as a count of total clock-cycles required to produce the final output, the total cycles per item of data, and the total resource use estimate in the number of lookup tables used.

Firing Intervals (f, g, h)	**Map Factor**	**Simulated Cycles**	**CPI** *(cycles/ item)*	**Resource Estimate** *(LUTs)*
(9, 15, 28)	(1,1,1)	204971	50.04	9588
(9, 15, 28)	(1,2,4)	37076	9.05	10988
(9, 15, 28)	(4,10,23)	9431	2.30	23688
(9, 15, 28)	(9,15,28)	4316	1.05	32688

Table 5.3: Experimental results for an extended 2D Shallow-Water Model, having three stalling nodes with varying *Firing Interval*, each parallelized to varying degrees as to generate design variants. The cycle count is from a cycle-accurate simulation based on Verilog-HDL generated automatically by the TyTra backend. The resource cost estimates are generated by the backend as well.

Looking at this particular slice of the DSE problem we can observe that even a simple application such as this generates a *very large search-space*, when using the naïve approach. The total space generated by *loop unrolling* the three *map* operations contains a number of products equal to the product of the *opaque function* firing interval. That is **3780** total program variants to be considered. Synthesizing and checking the performance of each variant is clearly unfeasible: synthesis alone can take on the order of tens of hours to perform.

Device	**CLBs**	**LUTs**
XC6SLX25	3758	15032
XC6SLX45	6822	27288
XC6SLX75	11662	46648

Table 5.4: Three Xilinx Spartan-6 FPGA devices targeted in the experiments, showing the maximum *CLB* and *LUT* resources available on each (Each CLB provides 4 LUTs).

In Figure 5.2 we show experimental results produced in joint unpublished work with Syed Waqar Nabi [1]. These correspond to the design-space exploration for a 2D Shallow-Water Model. Each of the three stalling nodes is *replicated* to varying degrees. Assuming lookup tables (LUTs) are the limiting resource, the figure shows which of the design variants can fit inside the three devices considered.

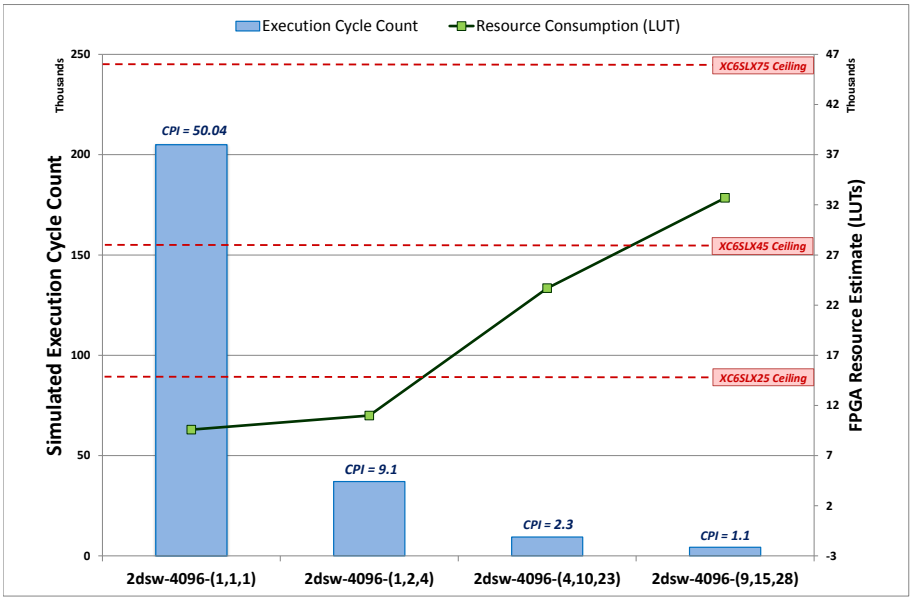


Figure 5.2: Hardware simulation showing program variants generated through DSE present the expected cost-performance characteristics.

Having confirmed through simulation that the solutions found by our DSE strategy lead to the expected performance gains, and consume the expected amount of hardware resources we moved on to confirm that the *asymptotic complexity* improvement shown *logically* by the reduction of the search space translates into physical speed-ups of the DSE process.

Firing Intervals (f, g, h)	**real time**	**user time**	**sys** *time*	**Variants**
(9, 15, 28)	0m0.644s	0m0.570s	0m0.072s	850500
(90, 15, 28)	0m4.113s	0m3.545s	0m0.568s	8505000
(90, 150, 28)	**crash**	**crash**	**crash**	**crash**

Table 5.5: Naive DSE strategy results.

For this purpose, we ran the three versions of our DSE strategy against the first example application, **SOR**, with modified costs, whilst gathering *tracing data* to see the amount of time spent in each. Unsurprisingly, as we raised the firing interval, the brute-force tactic quickly succumbed to the larger and larger design space, crashing early on.

[1] Syed Waqar Nabi produced the HDL representation of the program variants selected through DSE and the simulation results shown in Figure 5.2

With the expert tactic, but no hardware resource limits and filtering performed at the end, the results are more promising, but we could only scale up the firing interval so far before the process ran out of memory and was killed.

Firing Intervals (f, g, h)	**real time**	**user time**	**sys** *time*	**Variants**
(90, 150, 280)	0m0.219s	0m0.196s	0m0.024s	480
(250, 190, 280)	0m0.263s	0m0.241s	0m0.023s	700
(1250, 1190, 1280)	0m3.421s	0m3.060s	0m0.356s	3700
(9250, 1190, 1280)	0m19.795s	0m18.179s	0m1.616s	11700
(9250, 9190, 1280)	**crash**	**crash**	**crash**	**crash**

Table 5.6: Filtered DSE strategy results.

With the **fused strategy** the size of the resulting design space and the exploration time are further reduced. An unexpected "issue" occurred, where we realised that by scaling up the firing interval but leaving the opaque function costs intact, eventually none of the solutions were viable.

Firing Intervals (f, g, h)	**real time**	**user time**	**sys** *time*	**Variants**
(9, 15, 28)	0m0.209s	0m0.183s	0m0.028s	28
(90, 15, 28)	0m0.179s	0m0.168s	0m0.012s	90
(90, 150, 28)	0m0.179s	0m0.169s	0m0.011s	**none**
(9250, 9190, 128)	0m0.181s	0m0.157s	0m0.024s	**none**

Table 5.7: Fused DSE strategy results, with hardware limits

Because we also filter for hardware resource limits bounds, there is no solution that fits on the selected device, *XC6SLX150T*, once we increase the firing intervals past *(90,150,28)*. To test the limits of our solution, we lowered the cost of the opaque functions such that the hardware resource limits would not intervene until much later.

Firing Intervals (f, g, h)	**real time**	**user time**	**sys** *time*	**Variants**
(9250, 9190, 1280)	0m11.746s	0m11.660s	0m0.087s	**9250**
(9250, 9190, 121280)	1m46.542s	1m46.115s	0m0.416s	**121280**

Table 5.8: Fused DSE strategy results, with hardware limits, lowered kernel cost.

The runtime measurements conclusively prove that our improved DSE strategy is orders of magnitude quicker, in line with the design space exploration ratio shown in Table 5.2 indicating that the runtime is proportional to the design space size, and thus that our reasoning is sound. □

5.2 Conclusion

Throughout this work we have seen how modern HLS solutions bring considerable improvements to FPGA application development, particularly in the context of Heterogeneous HPC, where the HDL workflow is ill-suited for the task. It is true that certain applications can still benefit from a hand-crafted approach to optimization that revolves around the circuit-view of computation. This is primarily true for applications where the complexity of computation is low and timing requirements are strict. In general however, the ability to specify intended application semantics at a higher level of abstraction enables brings numerous benefits in terms of porting flexibility, reducing development cost and minimizing software errors. Where HLS solutions fall short, is in their ability to deliver optimal performance without significant input from an expert programmer.

We have also seen how taking the general idea of HLS further, as **TyTra Compiler Framework** does, can help deliver on the promise of performance portability. The TyTra *front-end compiler* recovers a *functional description of the application structure* that can be safely transformed into a large number of alternative implementations. The resulting program variants come with different performance and hardware resource costs, and thus expose a searchable design-space from which a DSE strategy can choose the optimal implementation. The TyTra Back-end compiler is responsible with Verilog code-generation as well as producing accurate cost-performance estimates.

The efficient Design Space Exploration strategy we have contributed through this work extends the TyTra Compiler framework by enabling it to automatically find the best set of optimizing transformations within a very small amount of time, orders of magnitude quicker than a brute force approach.

Our third chapter served to contrast our approach to those used in a number of practical and theoretical related works. On the practical side, we have seen that other HLS solutions can also benefit greatly from the use of DSE to maximise performance and reduce the amount of computational resources required to parallelize applications. On the theoretical side, we have seen that there is a strong connection between the imperative and functional programming language approaches to parallel computation. The link between parallel skeletons and recursion schemes on the one hand, and the link between recursion schemes, categorical data types and the Böhm-Berarducci encoding on the other seem to indicate that Design Space Exploration can be used to gain a tighter coupling between the semantics of the application being compiled and the optimizing compiler. In section 5.3, we speculate on the subject of future work that may benefit from this tighter coupling.

5.3 Future work

Design space exploration is a process that is widely used outside the specific area of optimizing parallel applications for FPGAs that we have tackled through this work. We expect that the search-space reduction method we have successfully applied to the TyTra compiler could be applied in other areas of computing science, in some cases with minimal translation effort. In this section we will highlight some of the areas and key problems we believe may benefit from our approach.

Cryptography

Cryptographic protocols are an abstract specification for cryptographic functions that can be described as the *composition* of *cryptographic primitives*. These cryptographic primitives usually belong one of the following types of functions.

- *Cryptographic hash functions*. Functions that *summarize* the input in a cryptographically safe way, allowing them to be compared, usually for equality, without having to reveal the exact input used.
- *Symmetric encryption routines*. *Bijective functions* that can obscure or reveal the meaning of a message, the so-called *clear-text*. Symmetric encryption routines use a single *cryptographic key* for both the *encryption* and *decryption* operations.
- *Asymmetric encryption routines*. Cryptographic functions that use different keys for the *encryption* and *decryption* operations. These enable secure communication amongst multiple parties using a shared medium.

Although numerous cryptographic primitives have been shown to provide adequate levels of security against direct attacks, either through formal methods or simply the test of time, proving that specific cryptographic protocols, which makes use of such primitives, are also secure remains difficult. In a survey on the formal verification of cryptographic protocols [Mea94], Meadows highlights several issues, including that:

- Methods based on state machines are insufficiently powerful, leading to situations where a human expert must assist in guiding the search for a secure cryptographic protocol.
- The granularity used to assess the quality of a cryptographic model can make the issue of searching for a cryptographic protocol design intractable.

Drawing on the parallels to the correct-by-construction approach in the TyTra compiler, we speculate that the DSE strategy optimization methods we have presented through this work may be used to solve the issue of cryptographic protocol design. This poses an immediate research question: what constitutes an appropriate *cost-performance model* in this context?

Artificial Neural Networks

Artificial Neural Network (ANN) design is a hot topic in the field of *machine learning* and *artificial intelligence*. The structure of of ANNs is remarkably similar to that of the dataflow computations the TyTra compiler framework was designed to handle, as we will now show.

- Artificial neurons, sometimes called *perceptrons* can be represented as pure, functional terms. A simple perceptron implementation is a function that computes the weighted sum of its inputs, adds a bias term and then squashes the result using a *sigmoid function*.
- Perceptrons are arranged into layers. The connections between these layers *compose* sub-solutions, sometimes called *features* into larger scale and more complex decisions.
- Information flow between perceptrons is completely defined by the connections between them and implying that a fully parallel execution model is appropriate. This makes FPGAs a particularly appealing target for the implementation of ANN inference passes.

Others have attempted to optimize such payloads for FPGAs using their own formulations of DSE which produce good results, but must exhaustively search the entire search space [ZLS+15]. We believe that a quicker exploration strategy could be derived from our methodology by simply defining a cost-performance model that also accounts for the *precision and accuracy* of ANNs.

Edge/Fog Computing

Edge computing represents a current that both opposes and augments that of *cloud computing*. The central idea is that of lowering latency by moving some or all computation away from the cloud and closer to the user. Where a particular computation serves a single user, a lower latency translates into better performance. If the computation to be performed serves multiple, perhaps competing users, moving the computation to an equidistant location means that service delivery is fair to everyone. Moving computation closer to the user can also help deliver better *privacy grantees* by keeping data within regional borders where specific legislation is in force. Optimizing such movement of computation is a use-case that resembles that of optimizing dataflow computations on FPGAs in two ways. Firstly, services can be distributed to a network of machines, each of which can be seen as the equivalent to a functional unit in a dataflow architecture. Secondly, the network that connects these machines is in some ways analogous to the tiered memory architecture found on FPGAs and GPUs. The key challenge here is more complicated than simply providing an adequate cost-performance model, however. The dynamic nature of public and private networks makes this problem potentially more difficult to solve. FPGAs provide a static target for the compiler, whereas data networks are subject to real-time changes.

Bibliography

[App] Apple. Opencl programming guide for mac. `https://developer.apple.com/library/archive/documentation/Performance/Conceptual/OpenCL_MacProgGuide/TuningPerformanceOntheGPU/TuningPerformanceOntheGPU.html/`. [Online; accessed 17-Mar-2020].

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[Bas04] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*, pages 7–16. IEEE Computer Society, 2004.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.

[BBH18] Adam D Barwell, Christopher Brown, and Kevin Hammond. Finding parallel functional pearls: Automatic parallel recursion scheme detection in haskell functions via anti-unification. *Future Generation Computer Systems*, 79:669–686, 2018.

[BBL+16] Thomas Bridi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. A constraint programming scheduler for heterogeneous high-performance computing machines. *IEEE Trans. Parallel Distributed Syst.*, 27(10):2781–2794, 2016.

[BDPV99] Bruno Bacci, Marco Danelutto, Susanna Pelagatti, and Marco Vanneschi. Skie: A heterogeneous environment for HPC applications. *Parallel Comput.*, 25(13-14):1827–1852, 1999.

[BDW16] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and opencl. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pages 634–648. ACM, 2016.

[Ben12] Shajulin Benedict. Energy-aware performance analysis methodologies for HPC architectures - an exploratory study. *J. Netw. Comput. Appl.*, 35(6):1709–1719, 2012.

[BG91] Stephen Brookes and Shai Geva. *Computational comonads and intensional semantics*. Citeseer, 1991.

[Bir87] Richard S Bird. An introduction to the theory of lists. In *Logic of programming and calculi of discrete design*, pages 5–42. Springer, 1987.

[Bol08] Thomas Bollaert. Catapult synthesis: a practical introduction to interactive c synthesis. In *High-Level Synthesis*, pages 29–52. Springer, 2008.

[Bot12] Mirela-Madalina Botezatu. A study on compiler flags and performance events. *Conseil Européen pour la Recherche Nucléaire*, 2012.

[BR96] Stephen Dean Brown and Jonathan Rose. FPGA and CPLD architectures: A tutorial. *IEEE Des. Test Comput.*, 13(2):42–57, 1996.

[BRS13] David F. Bacon, Rodric M. Rabbah, and Sunil Shukla. FPGA programming for the masses. *Commun. ACM*, 56(4):56–63, 2013.

[BS91] Thomas B. Berg and Howard Jay Siegel. Instruction execution trade-offs for SIMD vs. MIMD vs. mixed mode parallelism. In V. K. Prasanna Kumar, editor, *The Fifth International Parallel Processing Symposium, Proceedings, Anaheim, California, USA, April 30 - May 2, 1991*, pages 301–308. IEEE Computer Society, 1991.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, 1989.

[BTL10] Brahim Betkaoui, David B. Thomas, and Wayne Luk. Comparing performance and energy efficiency of fpgas and gpus for high productivity computing. In Jinian Bian, Qiang Zhou, Peter Athanas, Yajun Ha, and Kang Zhao, editors, *Proceedings of the International Conference on Field-Programmable Technology, FPT 2010, 8-10 December 2010, Tsinghua University, Beijing, China*, pages 94–101. IEEE, 2010.

[CAD+12] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P.

Singh. From opencl to high-performance hardware on FPGAS. In Dirk Koch, Satnam Singh, and Jim Tørresen, editors, *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31, 2012*, pages 531–534. IEEE, 2012.

[Cas18a] Stephen Cass. The 2017 top programming languages. *IEEE Spectrum*, 31, 2018.

[Cas18b] David Castro. *Structured arrows: a type-based framework for structured parallelism*. PhD thesis, University of St Andrews, UK, 2018.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[CCA+11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In John Wawrzynek and Katherine Compton, editors, *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 33–36. ACM, 2011.

[CH05] S Chtourou and O Hammami. Systemc space exploration of behavioral synthesis options on area, performance and power consumption. In *2005 International Conference on Microelectronics*, pages 5–pp. IEEE, 2005.

[Cha01] Rohit Chandra. *Parallel programming in openMP*. Morgan Kaufmann, 2001.

[CLS+08] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Proceedings of the IEEE Symposium on Application Specific Processors, SASP 2008, held in conjunction with the DAC 2008, June 8-9, 2008, Anaheim, California, USA*, pages 101–107. IEEE Computer Society, 2008.

[Col04] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

[Cur98] Matt Curtin. Write once, run anywhere: Why it matters. *Technical Article. http://java.sun.com/features/1998/01/wo*, 1998.

[DAF11] Mayank Daga, Ashwin M Aji, and Wu-chun Feng. On the efficacy of a fused cpu+ gpu processor (or apu) for parallel computing. In *2011 Symposium on Application Accelerators in High-Performance Computing*, pages 141–149. IEEE, 2011.

[DKK09] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating gpu binaries to tiered simd architectures with ocelot. Technical report, Georgia Institute of Technology, 2009.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[dMRVP09] Frédéric de Mesmay, Arpad Rimmel, Yevgen Voronenko, and Markus Püschel. Bandit-based optimization on graphs with application to library performance tuning. In Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman, editors, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 729–736. ACM, 2009.

[Doe03] Osvaldo Pinali Doederlein. The tale of java performance. *J. Object Technol.*, 2(5):17–40, 2003.

[Dow06] Jack Doweck. White paper inside intel® core™ microarchitecture and smart memory access. *Intel Corporation*, 52:72–87, 2006.

[DSW+13] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal ir: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[EM45] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2):231–294, 1945.

[ESFC14] Pacôme Eberhart, Issam Said, Pierre Fortin, and Henri Calandra. Hybrid strategy for stencil computations on the apu. In *Proceedings of the 1st international workshop on high-performance stencil computations, Vienna*, pages 43–49, 2014.

[Fau82] Antony A. Faustini. *The equivalence of an operational and a denotational semantics for pure dataflow*. PhD thesis, University of Warwick, Coventry, UK, 1982.

[FLP+18] Franz Franchetti, Tze Meng Low, Doru-Thom Popovici, Richard Michael Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. SPIRAL: extreme performance portability. *Proc. IEEE*, 106(11):1935–1968, 2018.

[FSH04] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In Michael D. McCool and Tomas Akenine-Möller, editors, *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware 2004, Grenoble, France, August 29-30, 2004*, pages 133–137. Eurographics Association, 2004.

[FT10] Grigori Fursin and Olivier Temam. Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim.*, 7(4):20:1–20:29, 2010.

[Fut99] Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *High. Order Symb. Comput.*, 12(4):381–391, 1999.

[GBL10] Cristian Grozea, Zorana Bankovic, and Pavel Laskov. FPGA vs. multi-core cpus vs. gpus: Hands-on experience with a sorting application. In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore-Challenge - Aspects of New Paradigms and Technologies in Parallel Computing [Proceedings of a conference held at the Heidelberger Akademie der Wissenschaften, March 17-19, 2010]*, volume 6310 of *Lecture Notes in Computer Science*, pages 105–117. Springer, 2010.

[GdSO09] Jeremy Gibbons and Bruno C. d. S. Oliveira. The essence of the iterator pattern. *J. Funct. Program.*, 19(3-4):377–402, 2009.

[Gee05] David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[GFG+16] Abhishek Gupta, Paolo Faraboschi, Filippo Gioachin, Laxmikant V. Kalé, Richard Kaufmann, Bu-Sung Lee, Verdi March, Dejan S. Milojicic, and Chun Hui Suen. Evaluating and improving the performance and scheduling of HPC applications in cloud. *IEEE Trans. Cloud Comput.*, 4(3):307–321, 2016.

[Gib06] Jeremy Gibbons. Datatype-generic programming. In Roland Carl Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming - International Spring School, SSDGP 2006, Nottingham, UK, April 24-27, 2006, Revised Lectures*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–71. Springer, 2006.

[GKSC13] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. Accelerating financial applications on the GPU. In John Cavazos, Xiang Gong, and David R. Kaeli, editors, *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6, Houston, Texas, USA, March 16, 2013*, pages 127–136. ACM, 2013.

[GLJ93] Andrew John Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In John Williams, editor, *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, pages 223–232. ACM, 1993.

[Gor96] Sergei Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume II*, volume 1124 of *Lecture Notes in Computer Science*, pages 401–408. Springer, 1996.

[Gra06] Martin Grabmüller. Algorithm w step by step, 2006.

[Gro09] Khronos OpenCL Working Group. The opencl specification. `https://www.khronos.org/registry/OpenCL/specs/opencl-1.0.pdf`, 2009. [Online; accessed 22-July-2019].

[GS08] David J. Greaves and Satnam Singh. Kiwi: Synthesis of FPGA circuits from parallel programs. In Kenneth L. Pocek and Duncan A. Buell, editors, *16th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2008, 14-15 April 2008, Stanford, Palo Alto, California, USA*, pages 3–12. IEEE Computer Society, 2008.

[GSAK00] Maya B. Gokhale, Janice M. Stone, Jeffrey M. Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the streams-c high level language. In *8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2000), 17-19 April 2000, Napa Valley, CA, USA, Proceedings*, pages 49–58. IEEE Computer Society, 2000.

[HCG+07] Martin C. Herbordt, Tom Van Court, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Douglas DiSabello. Achieving high performance with fpga-based computing. *Computer*, 40(3):50–57, 2007.

[HGMS+12] JA Herdman, WP Gaudin, Simon McIntosh-Smith, Michael Boulton, David A Beckingsale, AC Mallinson, and Stephen A Jarvis. Accelerating hydrocodes with openacc, opencl and cuda. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 465–471. IEEE, 2012.

[HHV15] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. A model-driven framework for guided design space exploration. *Autom. Softw. Eng.*, 22(3):399–436, 2015.

[HTWB10] Hans Hacker, Carsten Trinitis, Josef Weidendorfer, and Matthias Brehm. Considering GPGPU for HPC centers: Is it worth the effort? In Rainer Keller, David

Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore-Challenge - Aspects of New Paradigms and Technologies in Parallel Computing [Proceedings of a conference held at the Heidelberger Akademie der Wissenschaften, March 17-19, 2010]*, volume 6310 of *Lecture Notes in Computer Science*, pages 118–130. Springer, 2010.

[Hud97] Paul Hudak. Domain-specific languages. *Handbook of programming languages*, 3(39-60):21, 1997.

[Hug89] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[JHH+93] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

[JK13] Michael R. Jantz and Prasad A. Kulkarni. Exploring single and multilevel JIT compilation policy for modern machines. *ACM Trans. Archit. Code Optim.*, 10(4):22:1–22:29, 2013.

[JLBF10] Yang Jiao, Heshan Lin, Pavan Balaji, and Wu-chun Feng. Power and performance characterization of computational kernels on the GPU. In Peidong Zhu, Lizhe Wang, Feng Xia, Huajun Chen, Ian McLoughlin, Shiao-Li Tsao, Mitsuhisa Sato, Sun-Ki Chai, and Irwin King, editors, *2010 IEEE/ACM Int'l Conference on Green Computing and Communications, GreenCom 2010, & Int'l Conference on Cyber, Physical and Social Computing, CPSCom 2010, Hangzhou, China, December 18-20, 2010*, pages 221–228. IEEE Computer Society, 2010.

[JQR06] P Joly, A Quarteroni, and J Rappaz. Scientific computation. 2006.

[Käm09] Jochen Kämpf. *Ocean modelling for beginners: using open-source software*. Springer Science & Business Media, 2009.

[KF19] Michael Kruse and Hal Finkel. Design and use of loop-transformation pragmas. In Xing Fan, Bronis R. de Supinski, Oliver Sinnen, and Nasser Giacaman, editors, *OpenMP: Conquering the Full Hardware Spectrum - 15th International Workshop on OpenMP, IWOMP 2019, Auckland, New Zealand, September 11-13, 2019, Proceedings*, volume 11718 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2019.

[Kis10] Oleg Kiselyov. Typed tagless final interpreters. In Jeremy Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010,*

Oxford, UK, March 22-26, 2010, Revised Lectures, volume 7470 of *Lecture Notes in Computer Science*, pages 130–174. Springer, 2010.

[Kis12] Oleg Kiselyov. Beyond church encoding: Boehm-berarducci isomorphism of algebraic data types and polymorphic lambda-terms. `http://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html`, 2012. [Online; accessed 22-July-2019].

[KJS10] Eunsuk Kang, Ethan K. Jackson, and Wolfram Schulte. An approach for effective design space exploration. In Radu Calinescu and Ethan K. Jackson, editors, *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems - 16th Monterey Workshop 2010, Redmond, WA, USA, March 31- April 2, 2010, Revised Selected Papers*, volume 6662 of *Lecture Notes in Computer Science*, pages 33–54. Springer, 2010.

[KMG08] Nupur Kothari, Todd D. Millstein, and Ramesh Govindan. Deriving state machines from tinyos programs using symbolic execution. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks, IPSN 2008, St. Louis, Missouri, USA, April 22-24, 2008*, pages 271–282. IEEE Computer Society, 2008.

[KS97] Christoph W. Keßler and Helmut Seidl. The fork95 parallel programming language: Design, implementation, application. *Int. J. Parallel Program.*, 25(1):17–50, 1997.

[KSA+10] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating performance and portability of opencl programs. In *The fifth international workshop on automatic performance tuning*, volume 66, page 1, 2010.

[KSS+09] Joachim Keinert, Martin Streubühr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith. Systemcodesigner - an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. Design Autom. Electr. Syst.*, 14(1):1:1–1:23, 2009.

[LA04] Chris Lattner and Vikram S. Adve. The LLVM compiler framework and infrastructure tutorial. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *Languages and Compilers for High Performance Computing, 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers*, volume 3602 of *Lecture Notes in Computer Science*, pages 15–16. Springer, 2004.

[Lam88] Monica S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 318–328. ACM, 1988.

[LCP+11] Olav Lindtjorn, Robert G. Clapp, Oliver Pell, Haohuan Fu, Michael J. Flynn, and Oskar Mencer. Beyond traditional microprocessors for geoscience high-performance computing applications. *IEEE Micro*, 31(2):41–49, 2011.

[Leo08] Philip Heng Wai Leong. Recent trends in FPGA architectures and applications. In *4th IEEE International Symposium on Electronic Design, Test and Applications, DELTA 2008, Hong Kong, January 23-25, 2008*, pages 137–141. IEEE Computer Society, 2008.

[LNLG20] Joshua Lant, Javier Navaridas, Mikel Luján, and John Goodacre. Toward fpga-based HPC: advancing interconnect technologies. *IEEE Micro*, 40(1):25–34, 2020.

[LPS15] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python JIT compiler. In Hal Finkel, editor, *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, pages 7:1–7:6. ACM, 2015.

[Mal89] Grant Malcolm. Homomorphisms and promotability. In Jan L. A. van de Snepscheut, editor, *Mathematics of Program Construction, 375th Anniversary of the Groningen University, International Conference, Groningen, The Netherlands, June 26-30, 1989, Proceedings*, volume 375 of *Lecture Notes in Computer Science*, pages 335–347. Springer, 1989.

[Mal90] Grant Malcolm. Data structures and program transformation. *Sci. Comput. Program.*, 14(2-3):255–279, 1990.

[Mea94] Catherine A. Meadows. Formal verification of cryptographic protocols: A survey. In Josef Pieprzyk and Reihaneh Safavi-Naini, editors, *Advances in Cryptology - ASIACRYPT '94, 4th International Conference on the Theory and Applications of Cryptology, Wollongong, Australia, November 28 - December 1, 1994, Proceedings*, volume 917 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1994.

[Mee86] LGLT Meertens. Algorithmics: Towards programming as a mathematical activity. 1986.

[ML13] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[MR13] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 1–18. ACM, 2013.

[MRRP11] Pablo D. Mininni, Duane Rosenberg, Raghu Reddy, and Annick Pouquet. A hybrid mpi-openmp scheme for scalable parallel pseudospectral computations for fluid turbulence. *Parallel Comput.*, 37(6-7):316–326, 2011.

[NSP+16] Razvan Nane, Vlad Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Dean Brown, Fabrizio Ferrandi, Jason Helge Anderson, and Koen Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 35(10):1591–1604, 2016.

[NTN12] Hiromasa Nakayama, Tetsuya Takemi, and Haruyasu Nagai. Large-eddy simulation of urban boundary-layer flows by generating turbulent inflows from mesoscale meteorological simulations. *Atmospheric Science Letters*, 13(3):180–186, 2012.

[NV15a] Syed Waqar Nabi and Wim Vanderbauwhede. An intermediate language and estimator for automated design space exploration on fpgas. *CoRR*, abs/1504.04579, 2015.

[NV15b] Syed Waqar Nabi and Wim Vanderbauwhede. Using type transformations to generate program variants for FPGA design space exploration. In Michael Hübner, Maya B. Gokhale, and René Cumplido, editors, *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015, Riviera Maya, Mexico, December 7-9, 2015*, pages 1–6. IEEE, 2015.

[NV17] Syed Waqar Nabi and Wim Vanderbauwhede. FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis. *Journal of Parallel and Distributed Computing*, 2017.

[NV19] Syed Waqar Nabi and Wim Vanderbauwhede. Automatic pipelining and vectorization of scientific code for fpgas. *Int. J. Reconfigurable Comput.*, 2019:7348013:1–7348013:12, 2019.

[Ols95] Roland Olsson. Inductive functional programming using incremental program transformation. *Artif. Intell.*, 74(1):55–81, 1995.

[PA12] Oliver Pell and Vitali Averbukh. Maximum performance computing with dataflow engines. *Comput. Sci. Eng.*, 14(4):98–103, 2012.

[PKB14] Nikolay Pydiura, Pavel Karpov, and Yaroslav Blume. On the efficiency of cpu and hybrid cpu-gpu systems in computational biology tasks. *Comput. Sci. Applicat*, 1(1):48–59, 2014.

[RLFdS12] Ruymán Reyes, Iván López-Rodríguez, Juan J. Fumero, and Francisco de Sande. accull: An openacc implementation with CUDA and opencl support. In Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*, volume 7484 of *Lecture Notes in Computer Science*, pages 871–882. Springer, 2012.

[RVDDB10] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An experimental study on performance portability of opencl kernels. In *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, 2010.

[SC19] Lars Schütze and Jerónimo Castrillón. Efficient late binding of dynamic function compositions. In Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, pages 141–151. ACM, 2019.

[SCP02] Ronald Scrofano, Seonil Choi, and Viktor K. Prasanna. Energy efficiency of fpgas and programmable processors for matrix multiplication. In *Proceedings of the 2002 IEEE International Conference on Field-Programmable Technology, FPT 2002, Hong Kong, China, December 16-18, 2002*, pages 422–425. IEEE, 2002.

[SEP+09] Michael Showerman, Jeremy Enos, Avneesh Pant, Volodymyr Kindratenko, Craig Steffen, Robert Pennington, Wen-mei Hwu, et al. Qp: A heterogeneous multi-accelerator cluster. In *Proc. 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[SFLD15] 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 Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference*

on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015, pages 205–217. ACM, 2015.

[SG79] SHIVA SG. Computer hardware description languages. a tutorial. *PROC. I.E.E.E.; USA; DA. 1979; VOL. 67; NO 12; PP. 1605-1615; BIBL. 88 REF.*, 1979.

[SGO+98] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[SGPJ12] Jerôme Schalkwijk, Eric J Griffith, Frits H Post, and Harm JJ Jonker. High-performance simulations of turbulent clouds on a desktop pc: Exploiting the gpu. *Bulletin of the American Meteorological Society*, 93(3):307–314, 2012.

[SKH+99] Masakazu Suzuoki, Ken Kutaragi, Toshiyuki Hiroi, Hidetaka Magoshi, Shin'ichi Okamoto, Masaaki Oka, Akio Ohba, Yasuyuki Yamamoto, Makoto Furuhashi, Masayoshi Tanaka, et al. A microprocessor with a 128-bit cpu, ten floating-point mac's, four floating-point dividers, and an mpeg-2 decoder. *IEEE Journal of Solid-State Circuits*, 34(11):1608–1618, 1999.

[Ski05] David B Skillicorn. *Foundations of parallel programming*. Number 6. Cambridge University Press, 2005.

[Smi96] Douglas J. Smith. VHDL & verilog compared & contrasted - plus modeled example written in vhdl, verilog and C. In Thomas Pennino and Ellen J. Yoffa, editors, *Proceedings of the 33st Conference on Design Automation, Las Vegas, Nevada, USA, Las Vegas Convention Center, June 3-7, 1996*, pages 771–776. ACM Press, 1996.

[SN05] James E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[SS71] Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group Oxford, 1971.

[SSJ19] V Venkatesh Shenoi, Vaishali Shah, and Sandeep K Joshi. Hpc education for domain scientists: An indian experience and perspective. In *2019 26th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*, pages 64–70. IEEE, 2019.

[Ste] Diehl Stephen. Hindley-milner inference. `http://dev.stephendiehl.com/fun/006_hindley_milner.html#constraint-generation`.

[Ste15] Michel Steuwer. *Improving programmability and performance portability on many-core processors*. PhD thesis, Universität Münster, 2015.

[Tho10] David B Thomas. Acceleration of financial monte-carlo simulations using fpgas. In *2010 IEEE Workshop on High Performance Computational Finance*, pages 1–6. IEEE, 2010.

[UVN19] Cristian Urlea, Wim Vanderbauwhede, and Syed Waqar Nabi. Efficient FPGA cost-performance space exploration using type-driven program transformations. In David Andrews, René Cumplido, Claudia Feregrino, and Marco Platzner, editors, *2019 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019*, pages 1–2. IEEE, 2019.

[vAVSvN09] Alexander S. van Amesfoort, Ana Lucia Varbanescu, Henk J. Sips, and Rob van Nieuwpoort. Evaluating multi-core platforms for HPC data-intensive kernels. In Gearold Johnson, Carsten Trinitis, Georgi Gaydadjiev, and Alexander V. Veidenbaum, editors, *Proceedings of the 6th Conference on Computing Frontiers, 2009, Ischia, Italy, May 18-20, 2009*, pages 207–216. ACM, 2009.

[VD17] Wim Vanderbauwhede and Gavin Davidson. Domain-specific acceleration and auto-parallelization of legacy scientific code in FORTRAN 77 using source-to-source compilation. *CoRR*, abs/1711.04471, 2017.

[VN14] Mário P. Véstias and Horácio C. Neto. Trends of cpu, GPU and FPGA for high-performance computing. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–6. IEEE, 2014.

[VN19] Wim Vanderbauwhede and Syed Waqar Nabi. Towards automatic transformation of legacy scientific code into opencl for optimal performance on fpgas. *CoRR*, abs/1901.00416, 2019.

[VNU19] Wim Vanderbauwhede, Syed Waqar Nabi, and Cristian Urlea. Type-driven automated program transformations and cost modelling for optimising streaming programs on fpgas. *International Journal of Parallel Programming*, 47(1):114–136, 2019.

[VT13] Wim Vanderbauwhede and Tetsuya Takemi. An investigation into the feasibility and benefits of gpu/multicore acceleration of the weather research and forecasting model. In *International Conference on High Performance Computing & Simulation, HPCS 2013, Helsinki, Finland, July 1-5, 2013*, pages 482–489. IEEE, 2013.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In Harald Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 1988.

[WBC13] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. High-level synthesis of dynamic data structures: A case study using vivado HLS. In *2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013*, pages 362–365. IEEE, 2013.

[Weg72] Peter Wegner. Operational semantics of programming languages. *ACM SIGACT News*, (14):128–141, 1972.

[WL08] Enhua Wu and Youquan Liu. Emerging technology about GPGPU. In *IEEE Asia Pacific Conference on Circuits and Systems, APCCAS 2008, Macao, China, November 30 2008 - December 3, 2008*, pages 618–622. IEEE, 2008.

[WOL+17] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Jürgen Becker, and Mehdi Baradaran Tahoori. Energy efficient scientific computing on fpgas using opencl. In Jonathan W. Greene and Jason Helge Anderson, editors, *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, pages 247–256. ACM, 2017.

[WOPW13] Michal Witkowski, Ariel Oleksiak, Tomasz Piontek, and Jan Weglarz. Practical power consumption estimation for real life HPC applications. *Future Gener. Comput. Syst.*, 29(1):208–217, 2013.

[WVH04] Kurt Wall and William Von Hagen. Basic gcc usage. In *The Definitive Guide to GCC*, pages 59–99. Springer, 2004.

[Xil14] Xilinx. The xilinx sdaccel development environment, 2014.

[ZLS+15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In George A. Constantinides and Deming Chen, editors, *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*, pages 161–170. ACM, 2015.

[ZMS+16] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In John West and Cherri M. Pancake, editors, *Proceedings of the International Conference for High Performance Computing,*

Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016, pages 409–420. IEEE Computer Society, 2016.

[ZVL+14] Guanwen Zhong, Vanchinathan Venkataramani, Yun Liang, Tulika Mitra, and Smaïl Niar. Design space exploration of multiple loops on fpgas using high level synthesis. In *32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, South Korea, October 19-22, 2014*, pages 456–463. IEEE Computer Society, 2014.