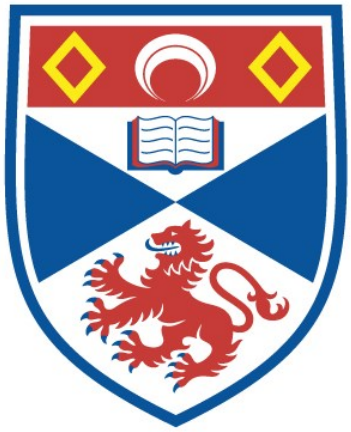
STRUCTURED ARROWS:
A TYPE-BASED FRAMEWORK FOR STRUCTURED PARALLELISM

David Castro

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



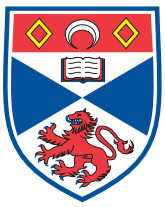
2018

Structured Arrows:

a Type-Based Framework for Structured Parallelism

by

David Castro



University
of
St Andrews

This thesis is submitted in partial fulfilment for the degree of

Doctor of Philosophy (PhD)

at the

University of St Andrews

27 July 2017

Abstract

This thesis deals with the important problem of parallelising sequential code. Despite the importance of parallelism in modern computing, writing parallel software still relies on many low-level and often error-prone approaches. These low-level approaches can lead to serious execution problems such as *deadlocks* and *race conditions*. Due to the non-deterministic behaviour of most parallel programs, testing parallel software can be both tedious and time-consuming. A way of providing guarantees of correctness for parallel programs would therefore provide significant benefit. Moreover, even if we ignore the problem of correctness, achiving good speedups is not straightforward, since this generally involves rewriting a program to consider a (possibly large) number of alternative parallelisations.

This thesis argues that new languages and frameworks are needed. These language and frameworks must not only support high-level parallel programming constructs, but must also provide predictable cost models for these parallel constructs. Moreover, they need to be built around solid, well-understood theories that ensure that: (a) changes to the source code will not change the functional behaviour of a program, and (b) the speedup obtained by doing the necessary changes is predictable. *Algorithmic skeletons* are parametric implementations of common patterns of parallelism that provide good abstractions for creating new high-level languages, and also support frameworks for parallel computing that satisfy the correctness and predictability requirements that we require.

This thesis presents a new type-based framework based on the connection between *structured parallelism* and *structured patterns of recursion*, that provides *parallel structures* as type abstractions that can be used to statically parallelise a program. Specifically, this thesis exploits *hylomorphisms* as a single, unifying construct to represent the functional behaviour of parallel programs, and to perform correct code rewritings between alternative parallel implementations, represented as algorithmic skeletons. This thesis also defines a mechanism for deriving *cost models* for parallel constructs from a queue-based operational semantics. In this way, we can provide strong static guarantees about the correctness of a parallel program, while simultaneously achieving predictable speedups.

Candidate's Declaration

I, David Castro, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 75,000 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree.

I was admitted as a research student at the University of St Andrews in September 2013.

I received funding from an organisation or institution and have acknowledged the funder(s) in the full text of my thesis.

Signature of Candidate: .....

Date: 27 July 2017

Supervisor's Declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Signature of Supervisor: .....

Date: 27 July 2017

Permission for Publication

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, David Castro, confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

Printed copy
No embargo on print copy.

Electronic copy
No embargo on electronic copy.

Signature of Candidate: .....

Date: 27 July 2017

Signature of Supervisor: .....

Date: 27 July 2017

Underpinning Research Data or Digital Outputs

Candidate's declaration

I, David Castro, hereby certify that no requirements to deposit original research data or digital outputs apply to this thesis and that, where appropriate, secondary data used have been referenced in the full text of my thesis.

Signature of Candidate: .....

Date: 27 July 2017

Acknowledgements

I am very grateful to my supervisor Kevin Hammond. We first spoke about the ideas behind this thesis in informal conversations during the DSL 2013 summer school in Cluj-Napoca, and I would have never guessed back then that they would turn into a full PhD thesis! I would have never achieved this if Kevin had not supported me and helped me learn.

I am very grateful to the members of the St Andrews FP group: to Chris B. and Vladimir for our conversations, coffee and lunch breaks; to Edwin, Susmit and all my fellow PhD students, Adam, Chris S., Matus, Franck, Chat, Franta and Yasir, for all the insights and ideas they shared with me.

My special thanks also to the biologists, Pau and Karim, for all the "inter-disciplinary culinary workshops" (i.e. lunch breaks).

A big thank you to Victor Gulías for introducing me into the academic world, and to Laura Castro for helping me and encouraging me when I was searching for PhD positions. I also extend my thanks to Henrique and Macias, for all the conversations and coffees we had.

I want to express my enormous gratitude to my family, specially my grand-mother, my parents, and my brother Jesus and Irina (and Jacobo!), for supporting me in my every single step. A big thank you to Ezequiel and Ana, specially for all the spanish delicacies you sent to Scotland!

Finally, Ana Inés, I am so grateful for everything that you have done for me, that I would need to write another thesis to explain it properly: Moi mal cho fixen pasar! Estoy aqui gracias a ti.

This work was supported by the University of St Andrews (School of Computer Science); by the EU FP7 grant “ParaPhrase:Parallel Patterns for Adaptive Heterogeneous Multicore Systems” (n. 288570); by the EU H2020 grant “RePhrase: Refactoring Parallel Heterogeneous Resource-Aware Applications - a Software Engineering Approach” (ICT-644235), by COST Action IC1202 (TACLe), supported by COST (European Cooperation in Science and Technology); and by EPSRC grant “Discovery: Pattern Discovery and Program Shaping for Manycore Systems” (EP/P020631/1).

David Castro
St Andrews
27 July 2017

Contents

Chapter 1

Introduction

This thesis deals with the important problem of writing *efficient* and *correct* parallel software. Parallel programming has become increasingly important. Parallel hardware is now ubiquitous, and there are major questions that need to be solved to ease the task of developing parallel software. On one hand, there is the problem of *performance*, i.e. writing parallel software that achieves good speedups. On the other hand, there is the problem of *correctness*, i.e. ensuring that a parallel program produces the correct results. The former has been addressed mostly by the high performance computing community, while the latter has been addressed mostly by the static analysis community. However, there has been little research on providing a general framework for achieving both of these simultaneously.

This thesis presents a novel framework, *Structured Arrows* (StA), which allows reasoning simultaneously about both correctness and performance of parallel programs. We developed the StA framework for a purely functional language that can be described as a subset of Haskell [Jon03], and it is based on well-understood theories of patterns of parallelism and patterns of recursion, specifically *algorithmic skeletons* [Col89] and *hylomorphisms* [MFP91]. StA provides a mechanism for: (a) reasoning about the sound introduction of parallelism to sequential functions; and (b) statically predicting the runtime performance of alternative parallelisations of functions.

1.1 The Importance of Parallel Programming

Software systems require ever increasing amounts of computing power. Single-processor systems are no longer able to cope with these growing demands. Less than two decades ago, we witnessed that clock-speeds have stalled, and a trend towards increasingly *parallel* architectures has started. Almost every hardware system nowadays is either a multicore, or incorporates highly parallel processing units such as *Graphic Processing Units* (GPUs). Parallel hardware is now no longer constrained to very specific fields, such as supercomputers or high-performance computing.

The shift in hardware systems has also brought a shift in the way that software systems are developed. Developers can no longer rely on increasing clock speeds to make their programs run faster, and learn new models, languages and frameworks. This requires a different mind-set. Developers have been used to writing highly optimised code for single-processor, single-core architectures. Consequently, the source code that is written is often inherently sequential, and hard to parallelise. In contrast to sequential programming, writing an optimised parallel application requires focusing on different aspects of the *program structure*, from the early stages of the software development process.

1.1.1 Challenges of Parallel Programming

Writing parallel applications is an inherently hard task that requires to focus on many low-level implementation details. *Concurrency* techniques are prevalent. Concurrency theories address the problem of decomposing a program into several components. These components can be run in *parallel* to speed up the program's run-time, provided that these components are *order-independent*. *Concurrency* techniques are, therefore, potential *parallelism* enablers. However, the order-independence of the components is not complete: components need to communicate and synchronise in order to solve the common goal. Communication and synchronisation errors lead to important problems such as *deadlocks* and *race conditions*. Detecting such problems, or debugging them, has become the subject of much re-

search [FKNS14, NY16, LZC+02]. However, even when a parallel program is correct, achieving good speedups is not easy.

Developers must perform a number of steps in order to parallelise a program. As a minimum, they need to

1. *identify* the different components that can be parallelised; and
2. *choose* which components to run in parallel, to achieve good speedups.

This means that developers must think not only about the correctness of their implementations, but also about how to decompose a program into multiple order-independent components, and simultaneously how to run in parallel those components that would improve the application's performance.

1.1.2 Models of Parallelism and Concurrency

In order to ease the task of writing parallel software, new models of parallel computing have been created, as well as programming languages and libraries, some of which have been adopted by industry (see, for example, the survey by Diaz et al. [DMCN12]). The main idea behind most of these models is to *raise the level of abstraction*. This is the same idea that underlies any successful attempt at tackling a problem that is too large, and that contains too many dimensions.

An example of model of parallelism is the *fork-join* model. Fork-join parallelism focuses on the *control flow* of the program, abstracting away the communication details. Examples include the *Cilk* programming language [BJK+96], which included the `spawn` and `sync` primitives[1].

Another example is the concept of *futures and promises*. The key idea is to decouple a value (*future*) from a computation (*promise*). The computation can then be parallelised, and the value later "demanded" where needed. An example of programming with futures in Haskell [Jon03, Hut16] is Glasgow Parallel Haskell, with the `par` and `pseq` constructs [THLJ98].

In contrast, *message passing* models of concurrency are aimed at describing the *interactions* between components as messages exchanged between them. They abstract away the specific low-level details involved in

[1] This was later removed, leaving just `spawn`.

the actual communication. Creating a *component*, and sending/receiving messages are generally considered primitives. Process calculi are examples of this. Some of these process calculi are Hoare's CSP [Hoa78], or Milner's π-calculus [Mil99]. Another example is the *Actor Model*, which originated in 1973 [HBS73] and that has influenced the creation of a number of programming languages and libraries, such as Erlang [CT09, AV90].

All of these approaches provide a simpler way of writing parallel software, but they still require programmers to focus on low-level details, and achieving good speedups can still be hard. The underlying problem is that the two questions that a developer must answer to parallelise a program still need to be answered *for each parallel program*: i.e. the components still need to be identified; and some of them parallelised, ensuring that the resulting program is both *correct* and *efficient*. One potential solution to those questions comes from the field of *structured parallelism*, which consists on programming parallel applications by combining implementations of common patterns of parallel computation [Pel98, RG03]. This thesis uses *structured parallelism* via *algorithmic skeletons* [Col89] as a means to avoid these problems.

1.2 Structured Parallelism

Algorithmic skeletons [Col89] are parametric implementations of common patterns of parallel programming. Using a pattern/skeleton approach, the programmer can design and implement a parallel program in a top-down manner. For example, the programmer could first identify the parallel patterns that occur in a particular piece of software, then select the patterns that potentially lead to the best speedups, and finally select a suitable implementation for those patterns, as a composition of one or more algorithmic skeletons. In other words, *algorithmic skeletons* focus on the *structure* of the computation. Moving from a lower-level *send/receive* approach to a higher-level *structured* approach has been compared to moving away from *goto* statements into more structured forms of recursion [Gor04]. From the functional programming perspective, an algorithmic skeleton is a higher-order function that receives the components of the algorithmic skeleton,

and implements some pattern of parallelism. There are many examples of algorithmic skeleton libraries, frameworks and languages [GzVL10].

Examples of algorithmic skeleton frameworks include:

1. *FastFlow* [ADK+11] is a skeletal parallel programming library for C++ that is targeted to the development of streaming computations.
2. *eSkel* [BCGH05] is a library for structured parallelism for C, and implemented in C+MPI.
3. The *Skel* [EBDH12] library provides an implementation of common algorithmic skeletons in the Erlang programming language.

An example algorithmic skeleton is the *task farm* skeleton, which captures a pattern that is also known as *master-slave* parallelism. The task-farm pattern captures worker replication, where a task farm takes a *worker*, and replicates it a number of times. An input collection of tasks is then distributed between the workers, and each of the workers operates on different tasks independently. An example of a skeletal program using a task farm, in a Haskell-like language is:

```
-- run         :: Skel a b -> a -> b
-- fun         :: (a -> b) -> Skel a b
-- farm        :: Int -> Skel a b -> Skel a b
-- filterImage :: Image -> Image

parallelFilter :: Skel [Image] [Image]
parallelFilter = farm 3 (fun filterImage)

output :: [Image]
output = run skel_fun images
```

In this code, the type `Skel a b` represents a structured parallel program that receives inputs of type `a`, and returns outputs of type `b`. The worker function `filterImage` is replicated 3 times by task `farm 3`. This parallel program is run on the inputs contained in `images`. The construct `fun` is an algorithmic skeleton that lifts a function to the algorithmic skeleton level. The code above is functionally equivalent to a usual **`map`** operation:

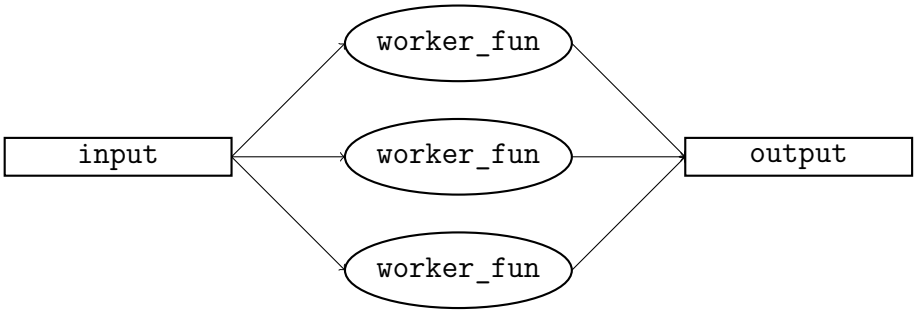


Figure 1.1: Task Farm

```
sequentialFilter :: [Image] -> [Image]
sequentialFilter = map filterImage
```

Algorithmic skeletons can be nested and composed in many ways, each of which represents a different parallelisation. In this thesis, the term *parallel structure* is used to refer to arbitrary nestings of algorithmic skeletons.

1.3 Structured Recursion

The previous section presented structured parallelism using algorithmic skeletons. Algorithmic skeletons provide a good high-level framework for writing parallel software, but they do not provide a mechanism for decomposing a program into components. This decomposition must usually be done manually by the developers. However, algorithmic skeletons have been linked to different *recursion patterns* in a number of different ways [RG03, Ski93b, SFLD15, Col93]. The connection between algorithmic skeletons and recursion patterns provides a mechanism for decomposing a program into components that can then be run in parallel, once a suitable parallel structure has been determined.

Recursion patterns, also known as *Recursion Schemes* [MFP91], capture common ways of writing recursive functions. Moving from `goto` statements towards structured loops is a major step towards more structured forms of iteration. A higher-level approach is the use of explicitly recursive functions instead of iteration, since they allow functions to be implemented as a list of equations. The use of recursion patterns represents the next step towards raising the level of abstraction in programs, by abandoning the idea of

using direct recursive calls in favour of calls to higher-order functions that implement some pattern of recursion. That is, abstraction is improved by moving from *explicit* to *implicit* recursion.

This style of programming recursive functions was pioneered by Richard Bird [Bir88] and Lambert Meertens [Mee86]. It has been promoted by several authors, such as Bird and Wadler [BW88], Gibbons [Gib02a], and Cunha [Cun05]. However, this style of programming is still not widely used. Barwell's forthcoming thesis [Bar17] considers the problem of identifying patterns of recursion in arbitrary Haskell code, in order to introduce parallelism.

Simple examples of recursion patterns include the **`map`** and `fold` functions in functional languages. For example, in Haskell, **`map`** can be defined as follows:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

A parallel **`map`** can be implemented by using a task farm, by replicating a worker function `f`, carefully distributing the "tasks" in the input list, and finally collecting the results. This connection will be exploited in depth in this thesis.

The mathematical properties of recursion patterns provide a way to rewrite, or refactor a program into an *extensionally equal* form. A well-known example of this is the *map-fusion* rule:

```
map f (map g xs) = map (f.g) xs
```

Two functions are extensionally equal if, when given the same input, they both produce the same result. The terms *functionally/extensionally equivalent* are used in this thesis as synonyms of *extensionally equal*. The term *program structure* is then used to refer to the combination of algorithmic skeletons and recursion schemes that are used (or that could be used) to implement a program. In this thesis, the main goal is to simultaneously: (a) explore how to change a program structure, in order to increase or introduce parallelism to it; and (b) statically predict the run-time performance of alternative program structures.

1.4 Structured Recursion and Structured Parallelism

This thesis builds on the connection between *structured parallelism*, in terms of combinations of algorithmic skeletons [Col89], and *structured recursion*, as combinations of recursion schemes [MFP91]. Several authors have proposed a change in programming techniques in favour of more structured approaches [Col04, RG03, Gor04, BW88, CPP06]. They advocate a shift from less structured programming, towards the usage of more structured programming constructs.

In the functional programming community, this "more structured approach" is represented by the *point-free* style of programming [BW88, CPP06]. This style of programming consists of programming without using variables, and creating complex functions by combining simpler functions using a number of primitive combinators. In the point-free style of programming, the data-flow of a program is made explicit. The most basic operation in the point-free style is function composition. For example, the following functions in Haskell are equivalent. One is programmed using a pointed style, the other using the composition operator, '.':

```
-- (f . g) x = f (g x)
pointed x = f (g x)
pointFree = f . g
```

The `pointed` version uses the variable `x` explicitly. In contrast, the function `pointFree` is built using the composition operator. Recursion in the point-free style is generally implemented by using common patterns of recursion, or recursion schemes [MFP91]. Abandoning explicit recursion in favour of using recursion schemes has been compared to moving from *goto* statements to structured iteration in imperative programming.

In the parallel programming community, the definition of *algorithmic skeletons* [Col89] represented a major milestone towards a more structured approach of parallel programming. Algorithmic skeletons are, when described from a functional programming perspective, higher-order functions that implement some common pattern of parallelism. Cole argues that structured parallel programming in terms of algorithmic skeletons will sim-

plify programming, enhance portability, improve performance, and offer scope for optimisations [Col04]. Algorithmic skeletons are not the only form of structured parallelism. *Evaluation strategies* are functions in Haskell that describe *how* the output of a function is computed [THLJ98]. These functions can be combined, using higher-order functions, to describe complex patterns of parallelism.

Gorlatch, in his paper "Send-Receive Considered Harmful" [Gor04], argues against the usage of *send/receive* operations, in favour of *MPI collective operations*, in an analogy to Dijkstra's famous "goto considered harmful". Gorlatch's argument is that the usage of low-level *send/receive* operations in MPI [GLS99] programs leads to code with a more complex, error prone communication structure. In contrast, Gorlatch points out the simplicity and compositionality of collective operations. These MPI *collective operations* are equivalent to algorithmic skeletons.

Several authors have pointed out, and even exploited the correspondence between common patterns of recursion, and common patterns of parallelism, such as [RG03, Ski93b, SFLD15, HTC98, SHMB05]. These authors all build on the results of the functional programming community to (semi-) automate the parallelisation of functions. These approaches show the feasibility of the approach, and represent important steps towards the automatic parallelisation of functions. However, they are either targeted at some specific architecture, and therefore not general, or do not consider the issue of statically reasoning about the performance of their parallel code. Our previous work [CHS16, CHSA17] builds on this connection between patterns of recursion and parallelism. We provide a general framework, in which programmers can reason simultaneously about program rewritings and performance.

1.5 Cost Models for Parallel Constructs

Even using a parallel skeleton library, and having some mechanism for automatically rewriting a program structure into functionally equivalent forms, it is still necessary to *decide* which is the best possible way to parallelise a program. In this thesis, *cost models* are used to achieve this decision.

A *cost model* is a static estimation of the run-time performance of a program, based on the characteristics of the program, the architecture where it is going to be run, and possibly some dynamic information, that may be obtained via profiling. The regular structure of algorithmic skeletons makes it possible to define *predictable* cost models. An early example of this is Skillicorn's cost calculus [SC95]. Brown et al. [BDH+13] define a number of cost models to direct refactorings of parallel Erlang programs. Hammond et al. [HBS16] derive a number of cost models for common algorithmic skeletons from their implementation on *x86-64* architectures that include micro-architectural details that model the relaxed memory consistency approach taken by modern CPU architectures.

The focus of this thesis is on *statically predicting* how to parallelise a program. Therefore, the cost models that are used in this thesis need to be derived from a *predictable* implementation of the parallel constructs. The definition of *predictable* is, however, quite vague. In this thesis, our notion of predictability is that the operational semantics of the algorithmic skeletons must allow the *automatic* derivation of cost equations. A number of examples are then run to gain evidence that the operational semantics accurately captures sufficient information to make realistic predictions of the overall performance of the program.

1.6 Aim of the Thesis

Automatically parallelising sequential functions has many challenges, since achieving good speedups depends on many factors: architecture-dependent parameters, task-sizes, communication and synchronisation times, memory accesses, etc. In order to automatically parallelise a program, all these factors need to be taken into account. However, considering them all might be computationally more expensive than simply profiling alternative parallelisations of a program. The problem is that the goal of "achieving the best possible speedups" is simply too large to be tackled in general. A more reasonable goal is "achieving provably optimal speedups *using some parallel programming model*". The problem is then moved to using or defining a suitable model of parallel programming and demonstrating that the least

cost solution is chosen using some well-defined cost model. Within that model, one should be able to define new parallel structures, and derive cost equations from them. If the model is reasonably accurate, a program should achieve real speedups that are close to the predicted ones. Therefore, a program that is proved to be optimal in the cost model is likely to be close to the real optimal program.

The second problem is rewriting the structure of a program. Predicting speedups is powerful, but only when coupled with a mechanism for systematically exploring the space of functionally equivalent program structures. Without such a mechanism, a programmer needs to manually change the structure of a program in order to generate the alternative parallelisations. In our opinion, the problem of systematically exploring the functionally equivalent structures should be restricted to structured parallel programs, which use a known set of algorithmic skeletons. Even in this case, it is impossible to explore *all* of the alternative implementations for a complex program, since this would lead to a very large search space. In order to solve this problem, a characterisation of *what* kind of automatic parallelisations can be explored, and a mechanism for doing so are needed.

Finally, any mechanism for automatically rewriting a program must ensure that the resulting program is functionally equivalent to the original one. Without strong static guarantees that the parallel program is functionally equivalent to the original program, programs that either deadlock or that give inconsistent results might be generated or an incorrect parallelisation could be produced.

To summarise, the task of parallelising programs would be greatly simplified by having a common framework that combines: (a) a predictable model of parallel structures from which cost models can be derived; and (b) a sound mechanism for systematically exploring the space of functionally equivalent parallel programs, up to a certain point. State-of-the-art approaches either tackle the former problem, or the latter, but not both. Current techniques either: a) require the programmer to guide each step of the tool, b) do not consider the extensionality of the parallel structures, or c) are targeted at some specific architecture, and so are not fully general. These techniques are reviewed in detail in Chapter 2

This thesis takes a type-based approach, and is the first known attempt at representing the potential parallelisation of a program as a type. Type-systems offer many advantages for statically enforcing properties on programs. For example, in the field of *behavioural types* [ABB+16], types are used to enforce that components of communication-centric software communicate according to given protocol specifications. Statically enforcing properties of communicating systems has also been studied by Brady in the context of the dependently-typed language Idris [Bra17, Bra13]. Type systems are not just useful for enforcing functional properties of programs, but also of non-functional properties, such as cost [Vas08].

The aim of this thesis is to develop novel state-of-the-art mechanisms for reasoning simultaneously about run-time performance of parallel programs, and correctness of program transformations transformations. Additionally, a mechanism for systematically exploring (a subset of) the space of functionally equivalent parallel implementations must be provided. This thesis represents the first attempt to develop a general type-based framework for parallel programming in which a programmer would write a program *once*, and then parallelise their program by providing a minimal amount of type annotations to instantiating different parameters in the framework.

A good general framework needs to address the following questions:

- *How can a program structure be extracted, i.e. how can a program be split into the different components that can be parallelised?*
- *What are all the different ways in which a program structure can be rewritten into functionally equivalent parallel forms?*
- *How can a program's run-time behaviour be statically predicted, based on its parallel structure?*

In order to answer the first and the second questions, this thesis presents a novel type-based framework, *Structure-annotated Arrows*[2], StA or *Structured Arrows*, which annotate function types with an abstraction of the underlying function structure. The type-system that is presented in this thesis describes how to extract a program structure, and annotates function types

[2]Arrows in this thesis are 'function types'. They should not be confused with Hughes' arrows [Hug00].

with the corresponding structure. A novelty of this type-based framework is that it uses a common recursion pattern, *hylomorphisms*, as a single unifying construct to guide the rewritings that can be applied to a program. Any attempt to rewrite a program starts by converting its structure into a canonical form as a composition of hylomorphisms. This type-system is then extended to deal not only with patterns of recursion, but also with arbitrary recursive functions. This extension is built on top of combinatory logic, exploiting the close correspondence between λ-calculus and combinatory logic [Bar84]. By tying the combinators to hylomorphisms, a full framework that can extract a program structure from explicitly recursive function has been developed.

These program structures can be used to guide any possible rewritings to the underlying function, thus answering the second question: *What are all the different ways in which a program structure can be rewritten into functionally equivalent parallel forms?*. The system is proven sound, and the space of functionally equivalent programs is characterised by a set of properties that was used to prove the system complete.

Finally, to answer the third question, *How can a program's run-time behaviour be statically predicted, based on its parallel structure?*, a mechanism for specifying the operational semantics of parallel structures has been developed. This mechanism consists of a simple but predictable queue-based language that comprises a number of processes that communicate by writing elements to shared queues. The operational semantics of a number of representative algorithmic skeletons can be specified in this language, and cost models can be automatically derived from these skeletons. These cost models can be shown to be accurate with respect to the underlying operational model, and therefore can be used to generate *provably optimal* programs, with respect to this model.

A theoretical framework for structured parallel programming that allows reasoning simultaneously about correct program transformations and performance has been developed. Its expressive power has been illustrated by a number of examples. The examples are used as common benchmarks of parallel programming, and the framework shows that it is able to capture many of their common good parallelisations. A prototype type-checking al-

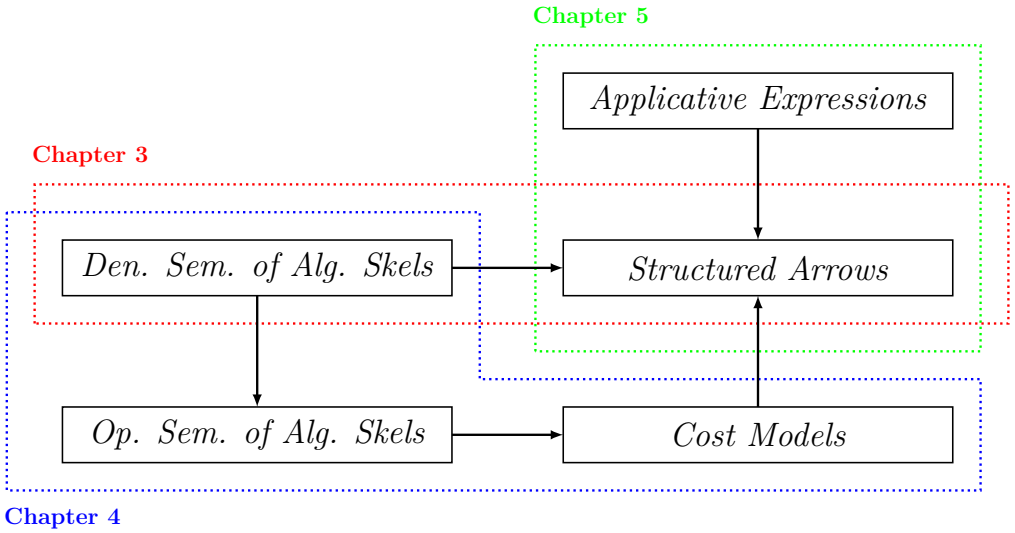


Figure 1.2: Contributions of the thesis

gorithm has been developed for the *Structured Arrows* framework, which is available at `https://bitbucket.org/david_castro/skel`. A number of examples have been compiled using the queue-based operational semantics. The cost models were compared to the real speedups on a 24-core and a 64-core system, both of which use a x86-64 architecture. Our results show not only a theoretical improvement to the state-of-the-art, but additionally provide preliminary results on how to turn this approach into a practical toolset.

1.7 Contributions

1.7.1 Main Contributions

The main novel contributions of this thesis are illustrated in Figure 1.2 on page 14. Specifically:

• *Structured Arrows (StA) (Chapter 3).* We have developed a novel type and effect system that annotates function types of a point-free programming language with the underlying *program structure*, which can be used to reason simultaneously about equivalent, alternative implementations and their cost on different architectures [CHS16]. Inspired by the notion of behavioural types [ABB+16], i.e. using types to represent *how* is a compu-

tation performed, this type system is the first known attempt at capturing the parallel structure program as a type. We prove soundness and completeness against a set of program equivalences derived from the laws of hylomorphisms, and the framework is illustrated by a number of common benchmarks for parallel programming systems. Finally, this thesis discusses the generality of the StA framework, which is not necessarily constrained to parallelising functions, and which has the potential of being used for studying general program optimisation.

• *A denotational semantics for common algorithmic skeletons in terms of hylomorphisms (Chapter 3).* By using the hylomorphism recursion pattern as a general unifying construct, the functional behaviour of different parallel programs can be compared by simply reducing the problem to comparing that they have the same *canonical representation*. Moreover, the alternative parallelisations of a program can be explored by systematically applying any possible rewriting, starting from its canonical representation. Although the relation between hylomorphisms and algorithmic skeletons has been previously noted, this is the first attempt at using it systematically.

• *An operational semantics of common algorithmic skeletons, in terms of a simple but predictable queue-based language, with a precise and well-known operational semantics (Chapter 4).* This queue-based language is designed so that synchronisation and communication between different components of algorithmic skeletons can be specified and reasoned about. The operational semantics of the algorithmic skeletons is shown to be sound with respect to the denotational semantics. This operational semantics is used to compile several examples, which are run on 24 and 64-core systems with x86-64 architectures, achieving speedups of up to 20 and 56 respectively.

• *A systematic mechanism for deriving cost models from the operational semantics of algorithmic skeletons (Chapter 4).* Essentially, this means that whenever a specification of the operational semantics of a parallel structure is available, the following are derived *for free*: (a) a translation scheme that can be used to compile a structured parallel program to low-level predictable code, and (b) cost models that can be used to statically predict the runtime behaviour of a program. The usage of these cost models is illustrated by comparing real vs. predicted speedups for a number of examples. This

shows that the cost models derived from the operational semantics predict a tight lower bound on the speedups, and therefore that a *provably optimal* program is likely to represent a good parallelisation.

• *An extension of the Structured Arrows framework to deal not just with point-free programs, but also with pointed programs that are written using explicit recursion (Chapter 5).* The extension of this framework consists on a type and effect system that annotates the types of `let` definitions with the underlying structure. This is done by building on well-known translations between λ-terms and combinatory logic, or *applicative expressions*, and the translation between applicative expressions and hylomorphisms. The former can be described in a compositional way, which is a requirement for using a type-based approach; and the latter can be done by applying a number of rewritings to the applicative expression in a systematic way, until it is in a "hylomorphism form". As usual, the soundness of the approach is proven, and a number of examples are provided to illustrate the usefulness of the transformations.

1.7.2 Minor Contributions

In addition, this thesis makes the following minor contributions:

• *A decision procedure for the functional equivalence of alternative parallel implementation (Chapter 3).* The decision procedure is developed and illustrated within the Structured Arrow framework. It boils down to rewriting any two parallel programs into some canonical representation by: (a) removing any parallel construct by replacing it with a sequential equivalent; (b) extracting a program structure from the resulting program in terms of hylomorphisms; and finally (c) rewriting the hylomorphism structure into a canonical representation. For obvious reasons, this decision procedure cannot be complete, but by extending the normalisation procedure, a larger class of programs can be compared. The decision procedure is illustrated by showing several fully worked examples.

• *A prototype implementation of the Structured Arrows framework (Chapter 4 and Chapter 5).* The StA framework is prototyped, both the type-and-effect system for the point-free language of hylomorphisms, but also the extension that incorporates pointed programs. This implementation

illustrates how these approaches can be applied to real programming languages, and also show that formalising this framework as a type-and-effect system has, as a side benefit, that most of the techniques described can be implemented as simple extensions of standard type-checking, inference and unification techniques. The prototype implementation is available at `https://bitbucket.org/david_castro/skel/`.

1.8 Thesis Outline

• **Chapter 2** presents an overview of structured approaches to both parallelism and recursion. The necessary background is provided to understand these theories, and a survey is given of a number of relevant algorithmic skeleton frameworks and recursion scheme libraries and frameworks. A survey of cost models for structured parallelism, or different models of parallelism is also provided. Finally, state-of-the-art approaches on using recursion patterns for parallelism are also surveyed, e.g. the Bird-Meertens formalism as a parallel programming framework, Skillicorn's cost calculus and automatic parallelism based on the Third List Homomorphism theorem. Finally, the work presented in this thesis is situated in the larger context.

• **Chapter 3** presents the Structured Arrow type-based framework for structured parallelism. The type-based framework consists of a type-and-effect system for a functional point-free language, Hylo. A Hylo expression represents the functionality of a program, while a structured arrow is used to introduce a *parallel structure*. The benefits of this separation are explored. The main benefits are: (a) a programmer can focus first on an initial sequential implementation, and later use the types to reason about how to parallelise it; (b) parallel structure can be changed by swapping type annotations. Proofs of soundness and completeness are presented, and a number of examples are shown that illustrate the technique with common benchmarks. The discussion of a prototype implementation is presented.

• **Chapter 4** contains a possible back-end for this system, based on a queue-based operational semantics of algorithmic skeletons. Cost equations for these skeletons are derived systematically from their operational semantics. These cost models are then passed as input to the StA framework, and

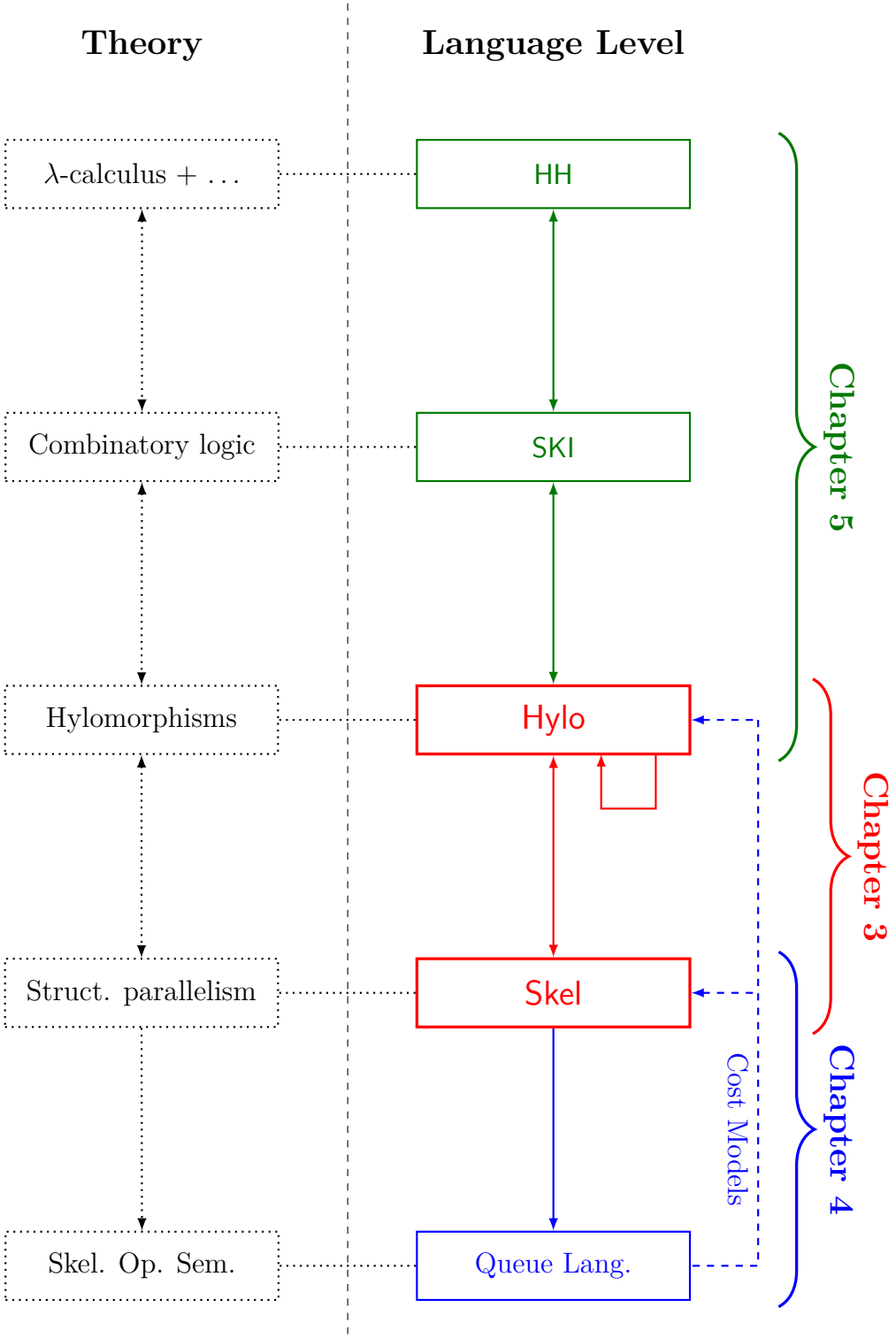


Figure 1.3: Structure of the Thesis

used to parallelise a number of programs. This illustrates the generality of the approach, since these parallel structures are simply parameters of the type-based framework. Together, Chapters 3 and 4 describe a full framework that provides a mechanism for deriving parallel implementations from high-level specifications of a program's parallel structure and functionality. The approach is evaluated by comparing real and predicted speedups for several examples, showing that we can predict tight lower bounds on speedups.

• **Chapter 5** describes an extension of the StA framework that is built on top of a relation that extracts program structures from functions defined in a subset of Haskell, HH. This is based on the close connection between combinatory logic and λ-calculus, which provides a mechanism for annotating expressions in HH with a "structure" in terms of *applicative expressions*, which are expressions that are built on top of combinatory logic. A set of simple transformations derives hylomorphisms from these applicative expressions. This is a novel usage of combinatory logic, which is aimed at performing source-to-source transformations, instead of using it for compilation. The expected soundness proof is provided for the novel relation, and it is illustrated by a number of common parallel programming examples.

• Finally, **Chapter 6** concludes the thesis, summarises the key results, points out achievements and limitations, and suggests how it could be improved thereafter.

1.9 Publications

The work that is described in this thesis has formed the subject of a number of papers.

- David Castro and Kevin Hammond. Skeletor: A DSL for Describing Type-based Specifications of Parallel Skeletons. In *Proc. Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU 2014)*, 2014.

  The main idea behind this thesis was first explored in this paper, where a formalisation in a dependently typed language was presented. In this

paper, the work is presented as an embedded DSL in a dependently typed language. The dependently typed approach, although useful and interesting, was quickly abandoned due to the lack of flexibility of the approach.

- David Castro, Kevin Hammond, and Susmit Sarkar. Farms, Pipes, Streams and Reforestation: Reasoning About Structured Parallel Processes using Types and Hylomorphisms. In *Proceedings of ICFP 2016: ACM International Conference on Functional Programming*, pages 4–17, Nara, Japan, September 2016.

  Most of the work in Chapter 3 is based on this paper. The paper omits some standard technical development which is provided in this thesis in full detail. Additionally, the results presented in that paper are discussed in more depth in Chapter 3. The term *Structured Arrows* for the type-based framework that is described in this paper is coined in this paper.

- David Castro, Kevin Hammond, Susmit Sarkar, and Yasir Alguwaifli. Automatically deriving Cost Models for Structured Parallel Processes using Hylomorphisms. *Future Generation Computer Systems*, 2017.

  Most of the work in Chapter 4 is based on this paper. In the paper, the mechanism for deriving automatically cost models from the operational semantics is sketched superficially. More details are provided in Chapter 4 of this thesis. The results are also described in more depth in this chapter.

- Adam D Barwell, Christopher Brown, David Castro, and Kevin Hammond. Towards Semi-Automatic Data-Type Translation for Parallelism in Erlang. In *Proceedings of the 15th International Workshop on Erlang*, pages 60–61, Nara,Japan, 2016. ACM.

  This extended abstract outlines a novel set of refactorings for datatype translation in the Erlang programming language. The collaboration on Barwell's work led to some of the discussion for future work and extending the Structured Arrow framework with further rewritings.

Chapter 2

Structured Approaches to Parallelism and Recursion

This thesis builds on the connection between structured parallelism and structured recursion, as was stated in Chapter 1. This Chapter provides an overview of the common techniques in structured parallelism (Section 2.2) and recursion (Section 2.3), contains a survey of state-of-the-art techniques in structured parallelism frameworks (Section 2.4), and presents a comparison of the characteristics of current state-of-the-art approaches, and situates the work developed for this thesis in the larger context.

2.1 Classes of Architectures

Unstructured techniques of parallelism consist of implementing parallel programs by using some *suitable model of concurrency*, in which programmers need to specify the individual components of their parallel program, and have a direct control over how communication between components happens. Traditionally, a suitable model of concurrency is chosen based on the characteristics of the target architecture.

In 1966, Michael Flynn proposed a taxonomy of computer architectures [Fly72], which was used to explain the different kinds of parallelism available depending on the characteristics of the underlying architecture. The original taxonomy comprises four classifications that are shown in in Figure 2.1a on page 23. In Flynn's original taxonomy, a sequential computer

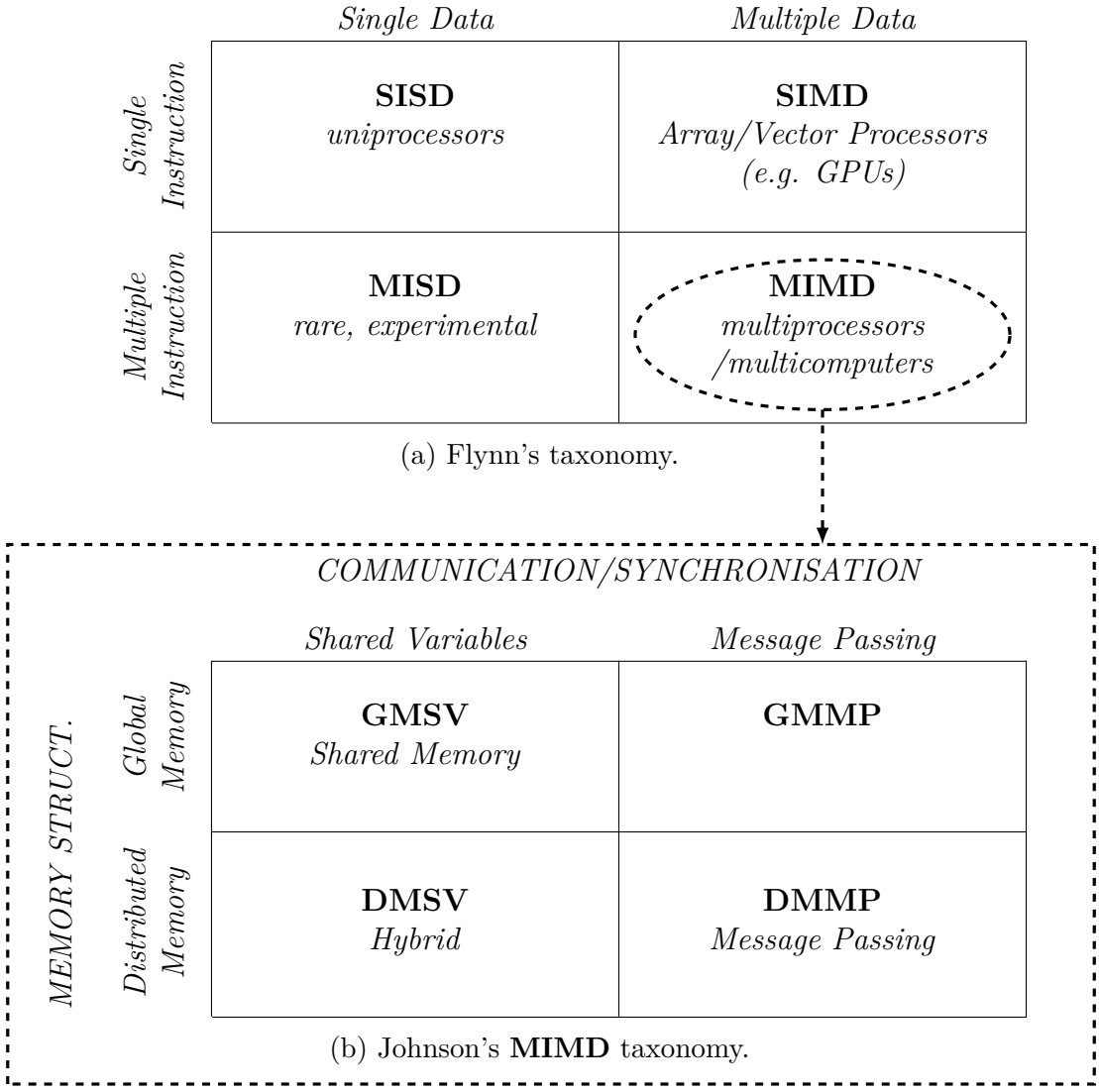
architecture is called SISD, i.e. a computer architecture in which a single instruction stream operates on a single data stream. A SIMD architecture exploits multiple data streams against a single instruction stream. *Graphics Processing Units* (GPUs) are examples of SIMD architectures. They exploit the natural parallelism that arises from performing an operation on different, independent data points. The MISD class of architectures exploits the parallelism of performing multiple, independent instruction streams on a single data stream. However, this architecture is not very common. Finally, MIMD architectures can operate multiple instruction streams, possibly on different data streams. This is the class of architectures of modern (possible heterogeneous) multi-processor and multi-core machines.

The MIMD class of architectures is too general. Johnson completes Flynn's taxonomy with a further classification of MIMD architectures, based on the memory structure of the architecture and how communication/synchronisation is done in such architectures [Joh88]. Figure 2.1b on page 23 presents Johnson's taxonomy, which consists of four different classes of MIMD architectures. The class of GMSV systems are classical shared-memory multiprocessors. There are not many examples of GMMP systems, and the few that exist are experimental. The class of DMSV architectures correspond to distributed shared memory systems. Finally, the class of DMMP architectures correspond to distributed-memory systems.

Classes of Parallelism and Concurrency

The Flynn-Johnson taxonomy illustrates one of the main difficulties of parallel programming. Generally, different parallel programming models and techniques are better suited for programming different classes of architectures. A common classification of different parallel programming models differentiates *data* parallelism from *task* parallelism.

Data parallelism In this form of parallelism, functions are applied in parallel to different, independent parts of the input data. In data parallelism, the input to parallel programs must be distributed across different components that operate on the data in parallel. Data parallelism is available in both MIMD and SIMD architectures. SIMD architectures naturally



(a) Flynn's taxonomy.

(b) Johnson's **MIMD** taxonomy.

Figure 2.1: Flynn-Johnson Taxonomy

support data parallelism, by applying the same instruction stream on multiple data streams in parallel. An example of data parallel programming environments is the CUDA programming model [Coo12]. From a functional programming perspective, data parallelism arises, for example, from instances of `map` functions. A representative example of data parallelism in the functional paradigm is Data Parallel Haskell [CLPJ+07].

Task parallelism This is a form of parallelism in which *tasks* are distributed across multiple processing units of some hardware architecture. Task parallelism is therefore better suited for MIMD architectures. Tra-

ditional approaches to task parallelism involve using different concurrency mechanisms, again depending on the underlying architecture. For shared-memory parallel programming, for example, a common approach is the *OpenMP* API extension for C/C++ and Fortran languages [DM98]. In OpenMP, a parallel program is implemented by using a number of directives and *pragmas* for thread creation, specifying parallel regions, barriers for synchronisation, etc. Communication between the different threads or processes of a parallel application is implicit, and relies on the usage of variables that point to shared memory locations. By default, variables are shared, and programmers must declare which variables are private.

A common approach for programming distributed-memory systems is the *Message Passing Interface (MPI)*, a parallel programming library available for C/C++ and Fortran [GLS99]. Parallel programs in MPI, in contrast to OpenMP, are programmed by explicitly defining how the communication between components happens. This communication is implemented by using a number of functions, which range from point-to-point communication functions, such as `MPI_Send`, to collective functions, `MPI_Reduce`.

For hybrid systems that combine shared-memory with distributed-memory architectures, such as clusters of SMP nodes, a combination of MPI+OpenMP can be used [RHJ09]. This approach would use message passing to explicitly specify the communication between SMP nodes, and rely on shared variables for the parallelisation of the specific tasks performed at each SMP node.

The OpenMP and MPI libraries correspond to the *fork-join* and *message-passing* models of concurrency. In general, in the *fork-join* model of concurrency, a programmer focuses on specifying the *control flow* of a parallel program. On the contrary, in the *message-passing* model of concurrency, a programmer focuses on specifying not only the different components of a parallel program, but also their specific interactions in terms of the messages exchanged between them. Examples of the *fork-join* model of parallelism also include the Cilk language [BJK+96], or the .NET Task Parallel Library [LSB09]. Other examples of the *message passing* model include the Erlang programming language [AV90, CT09] or the Go programming language [Tea].

2.2 Structured Parallel Programming

This section presents the notion of *structured parallel programming*, focusing on *structured parallelism* in terms of *algorithmic skeletons*. An approach to parallel programming can be considered *structured* if it provides abstractions to the programmer that correspond to *common patterns* of parallelism.

Structured parallel approaches, such as *algorithmic skeletons* [Col89], offer many advantages in terms of built-in safety and parallelism by construction. They can eliminate *by design* common but hard-to-debug problems including *deadlock* and *race conditions*. As discussed in the previous section, such problems are prevalent in typical low-level *concurrency based* designs for parallel systems. A survey of different key algorithmic skeleton approaches is provided later in Section 2.4.

2.2.1 Functional Parallel Programming

Higher-Order Functions are important to functional programming languages. The ability to use and define new HOFs, when coupled with a model of parallel programming, provides the necessary components for a structured approach to parallelism. An example of this is GpH and Haskell's *evaluation strategies* [THLJ98]. The idea of evaluation strategies is to fully separate the implementation of an algorithm from *behavioural code*, i.e. code that represents *how* the output is computed. The key ingredients of evaluation strategies are the `par` and `pseq` constructs[1]:

```
1 par  :: a → b → b
2 pseq :: a → b → b
3
4 type Strategy a = a → ()
5
6 using :: a → Strategy a → a
7 x `using` s = s x `pseq` x
```

[1] Originally, Trinder et al. [THLJ98] use `seq` instead of `pseq`. Later, a `seq` function was introduced in the Glasgow Haskell Compiler that does not guarantee that the arguments are evaluated in sequence, which is a requirement for evaluation strategies. To solve this problem, an additional `pseq` function was introduced to GHC [MML+10, MPJS09].

Briefly, `par` and `pseq` are used to control how evaluation happens. The expression `par x y` "marks" the first argument `x` as potentially parallelisable by adding it to a *spark queue*, and returns the second argument `y`. GHC's run-time system will select some of the sparks, and run them in parallel. The expression `pseq x y` evaluates the first argument `x` to weak head normal form, and *then* returns the second argument `y`. Note that not every call to `par` will result in a computation being performed in parallel, since this is decided by GHC's runtime system. Combinations of `par` and `pseq` can be used to define complex patterns of parallelism, known as *strategies*. Below we show an example of evaluation strategy:

```
1 parList :: Strategy a -> Strategy [a]
2 parList strat [] = ()
3 parList strat (x:xs) = strat x `par` parList strat xs
```

The strategy `parList` applied to a strategy `strat` takes a list as input, and triggers the evaluation of each of the elements in parallel, using the strategy `strat`.

The `Par` monad [MNPJ11] is a monad for *deterministic parallelism* that provides more low-level control to programmers on how parallelism happens. The `Par` monad is built on top of the *fork-join* model of parallelism, and provides control over the necessary side effects that are required to parallelise a program. In this monad, computations of type `Par a` are computations that return a value of type `a`. The word *deterministic* means that any computation in the `Par` monad will return the same value, independently of the evaluation order. Therefore, a function `runPar` can be safely provided:

```
1 runPar :: Par a → a
```

The `Par` monad is implemented on top of the `IO` monad. Therefore, `runPar` must be implemented using the function `unsafePerformIO`. Despite the low-level structure exposed by the basic operations of the `Par` monad, higher-level structures can be defined thanks to the usage of Haskell's type classes and HOFs [MNPJ11].

The *Eden* programming language [BLOMPM96] is a dialect of Haskell that is aimed at writing parallel and concurrent software. It provides a

construct, `process` for explicit process creation, and the operator # for process instantiations.

```
1 process :: (Trans a, Trans b) ⇒ (a → b) → Process a b
2 ( # )   :: (Trans a, Trans b) ⇒ Process a b → a → b
```

Processes communicate using unidirectional communication channels, so the values used by processes must be *transmissible*. This is handled by the `Trans` type class.

The advantages of the purely functional programming paradigm for parallel computing are very important for finding a model of parallel computation that scales to larger problems [HM00]. Purity ensures that computations that are run in parallel do not interfere with each others results. Therefore, pure computations can be run in parallel. Although the `Par` monad does expose side-effecting computations to the programmer, they do so in a controlled way. This controlled way ensures that computations are *deterministic*, i.e. side effects do not affect the result of the parallel computations. The existence of *higher-order functions* provides a tool for defining higher-level structures that can be used to simplify the development of parallel programs.

However, these solutions rely on particular low-level models of parallel computing: e.g. *futures* with `par` and `pseq` in GpH, `process` in Eden, or `fork` in the `Par` monad. They all rely on built-in support from the run-time system.

2.2.2 Dataflow Languages

In dataflow languages, programs are specified in terms of how data *flows* through the instructions. In LUSTRE [CPHP87], for example, a program comprises a series of node definitions. Nodes are implemented as sets of equations, and are connected to form larger programs. Lucid [AW77] is another example of synchronous dataflow language, where variables represent streams of values that are modified using transformers and filters.

In the functional programming community, the language *pH* [NAA+95] is an extension of Haskell for parallel computing, that is highly related to dataflow languages. The key characteristic of *pH* is that its expressions are

reduced using *lenient* evaluation. Lenient evaluation is a non-strict evaluation order that lies somewhere between lazy and eager evaluation [Tra88], in which the evaluation of the expressions is delayed until their input data is available, but not necessarily any longer. Roughly, lenient evaluation exposes a large amount of fine-grained parallelism, since as soon as some expression has its inputs available, it can be evaluated in parallel to other independent expressions.

The main problem with dataflow languages is precisely that they expose too fine-grained parallelism to achieve good speedups.

2.2.3 Coordination Languages

Coordination languages [GC92] follow a model of computation in which the *computation* model is separated from the *coordination* model. The computation model is used to represent sequential components of a program, and the coordination model is used to *create* computation activities, and provide a mechanism for them to communicate. Most of the coordination languages offer low-level primitives for the coordination model. These low-level primitives can be combined into higher-level constructs. However, providing this low-level control over how the computation units communicate makes it possible to write incorrect programs that deadlock or have race conditions.

An example of coordination language is the SCL language [DGTY95]. In SCL, coordination is implemented using HOFs that are split into three categories: *configuration*, *elementary* and *computational* skeletons. Configuration skeletons provide a mechanism for specifying data alignment, elementary skeletons are basic data-parallel operations, and computational skeletons abstract common parallel control-flow patterns.

However, the concept of coordination languages is highly related to the algorithmic skeleton approach to structured parallelism. In fact, many of the coordination structures in SCL can be found in other skeletal approaches. In a sense, coordination languages use, or provide mechanisms for defining algorithmic skeletons [Col89, Col04].

2.2.4 Type-Based Parallel Programming

A type system comprises a set of rules that assign a property, a *type*, to the syntactic constructs of a languages. These properties, types, can be checked statically, dynamically or a combination of both [Pie02]. Types are generally related to correctness or consistency properties, "well-typed programs do not go wrong", but they can also be used to check non-functional properties (see, e.g. Vasconcelos thesis on statically checking the space cost of programs using sized types. [Vas08]).

Type systems and type-based approaches have been used for parallel computing. By using types to ensure correctness properties or guide the parallelisation process, types add some form of structure on top of parallelisation techniques. For example, *sized-types* have been used to reason about termination and productivity of parallel programs [PnS05, PnS01] written in Eden. In this line of work, types are used to ensure two important properties of parallel programs, termination and productivity, but the question of *how* to parallelise sequential implementations, or statically predicting the run-time performance of parallel programs was not addressed.

Few type-based approaches have been applied to parallelise programs. Xu et al. [XKCH03] developed and formalised the PType system, the first ever type system that is used to guide the parallelisation of sequential implementations. However, PType is just used to determine the parallelisability of recursive functions. The parallelisation is done in terms of parallel skeletons that correspond to list/tree homomorphisms. The PType system offers little control to the programmer on how parallelisation is done. The PType system does not provide abstractions for selecting alternative parallelisations of list/tree homomorphisms, or reasoning about the cost of the resulting parallel programs.

Brown [Bro13, Bro14, Bro17] follows a type-oriented approach to parallelism in the Partitioned Global Address Space memory model of parallelism. The PGAS model assumes a global memory address space that is partitioned into different portions. The different portions are local to different processing element, thus exploiting locality of reference. Brown's approach is to rely exclusively on type annotations to determine how the data is allocated, partitioned and distributed. Brown's approach uses type

annotations to *tune* parallel programs in the PGAS model, but does not use types to explore alternative parallel structures. In Brown's approach, whenever type annotations are not provided, the type system selects default values and proceeds as usual, and does not provide cost models for estimating the run-time performance of the alternative parallelisations.

2.2.5 Algorithmic Skeletons

Algorithmic skeletons offer many advantages, such as correctness and predictability, since the communication and computation structure of algorithmic skeletons is generally known statically. Good structural cost models that predict the run-time performance of skeletal programs have been previously studied [HM00, HC02, LL10]. A number of common skeletal approaches are surveyed later in Section 2.4. In this section, a brief background on algorithmic skeletons is provided. The notation for algorithmic skeletons that is used in this chapter is roughly the one used in the rest of this thesis. The following distinct terminology is used:

1. *Pattern* or *parallel pattern*: a common form of parallelism, or a common way of implementing the computation and communication structure of a number of parallel programs.

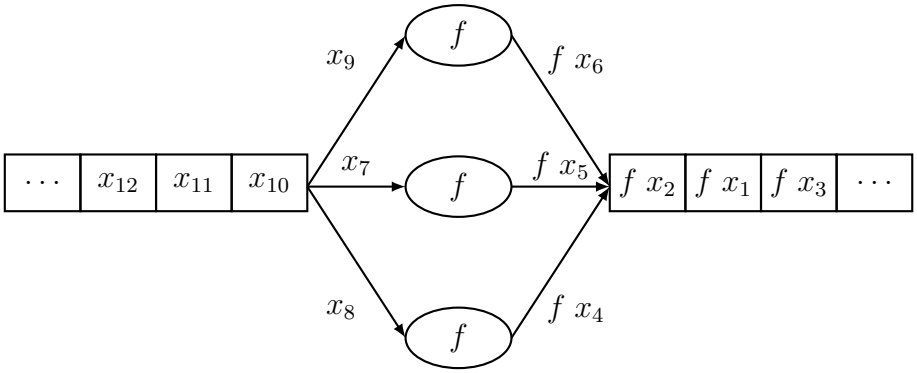
2. *Skeleton*, *algorithmic skeleton* or *parallel skeleton*: a particular implementation of a common pattern of parallelism.

3. *Skeletal program*: a parallel program implemented as a composition of algorithmic skeletons.

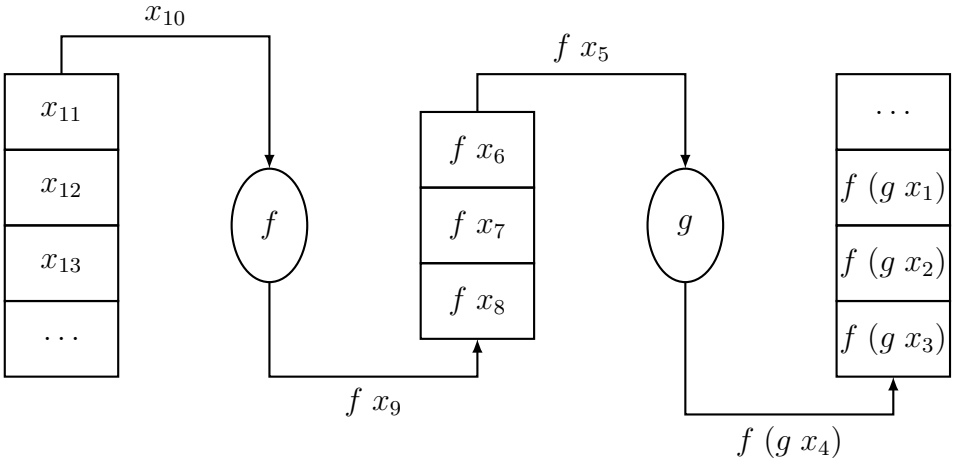
4. *Worker*: the components of skeletal programs. Workers are small skeletal programs, used in the implementation of a larger skeletal program.

5. *Task*: an independent unit of computation. Tasks are distributed to the different workers of a skeletal program, according to its parallel structure.

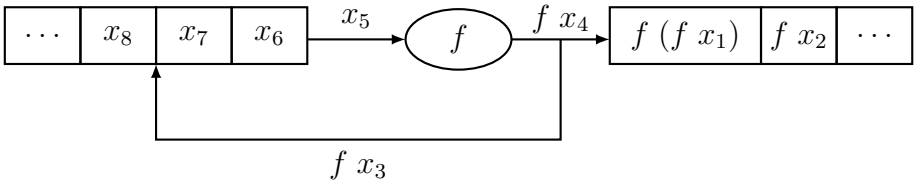
In this section, a brief overview of a small but representative [DT13] set of *task-parallel* skeletons is provided. The syntax used to represent algorithmic



(a) Task-Farm Skeleton



(b) Pipeline Skeleton



(c) Feedback Skeleton

Figure 2.2: Common Algorithmic Skeletons

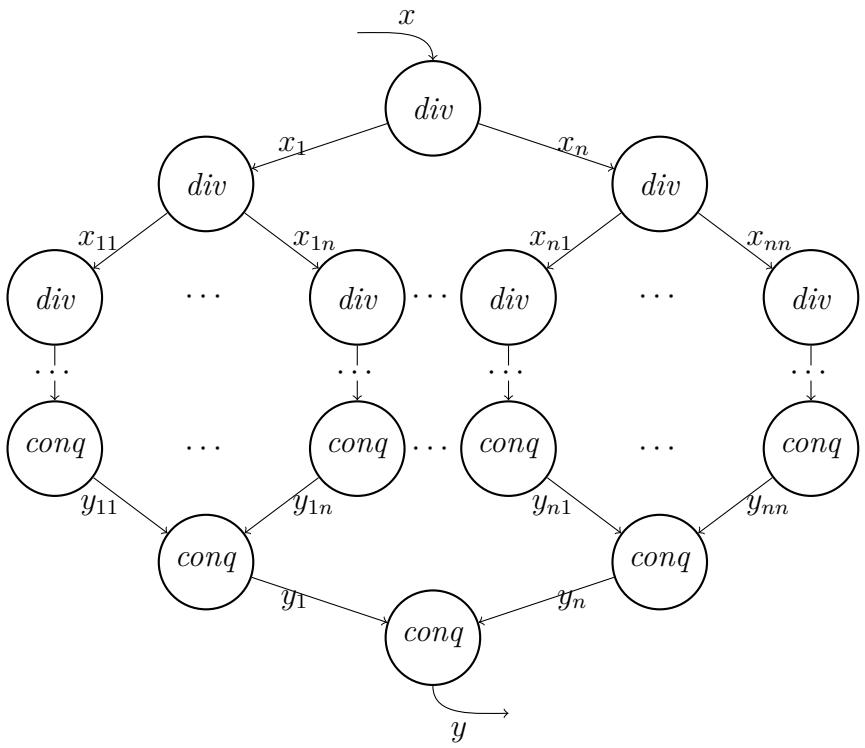


Figure 2.3: Divide and Conquer Skeleton

skeletons and skeletal programs is the one that will be used throughout this thesis.

Task Farm Skeleton

A common pattern of parallel computation is the *task farm* pattern, also known as *master-slave* or *master-worker*. In this pattern, a *worker* is replicated n times, each of which operating in parallel over independent tasks. Task independence is crucial; if tasks are not independent, the result of a worker might affect the result of another worker, and therefore leading to unpredictability and inconsistencies in the result.

The implementation details of task-farms vary between different skeleton implementations. A common example of skeleton implementation relies on a *master* process that is in charge of: (a) creating the worker processes, *slave*; (b) distributing the tasks between the workers; and (c) collecting the results. Another example, in FastFlow [ADK+11], uses two different pro-

cesses for task distribution and collection. In FastFlow, a task farm skeleton is a *streaming* skeleton that receives a stream of independent input tasks, and returns a stream of output tasks. Roughly, a task farm in Fastflow comprises: a) an *emitter* process that receives the input tasks, and distributes them to the different workers; b) a number of *workers* that perform some computation on the tasks, and send the results to a collector process; and, c) a *collector* process returns a stream of output tasks, possibly performing some computation, e.g. reordering them to match the order in which they were received by the emitter.

In this thesis, we use the queue-based task-farm implementation illustrated by Figure 2.2a on page 31, which is based on [HBS16]. In this implementation, the *master* worker enqueues tasks into a shared queue, from which all the workers read the tasks. The workers then enqueue the results in the output queue. The resulting tasks do not necessarily respect the ordering of the input, and this must be handled by the master worker.

An example of usage of the task-farm skeleton is an image stream processing program, that applies a series of filters to all the elements of an input stream of images, followed by an edge detection algorithm. In a Haskell-like functional language, this program would be:

```
process :: Stream Img -> Stream Img
process (img : imgs)
  = edge (filter2 (filter1 img)) : process imgs
```

This program can be parallelised using a task-farm skeleton, for example, as follows:

```
-- fun :: (a -> b) -> Stream a -> Stream b
-- farm :: Int -> (Stream a -> Stream b) -> Stream a -> Stream b

process' :: Img -> Img
process' img = edge (filter2 (filter1 img))

process :: Stream Img -> Stream Img
process = farm 5 (fun process')
```

The `fun` construct lifts functions into workers: `process'` is a function, `fun process'` is a worker that computes `process'`. The construct `farm 5` replicates 5 times the worker `fun process'`. More examples of task farms can be found in a number of papers on algorithmic skeletons, e.g. [ADK+11, BDH+13].

Pipeline Skeleton

The notion of pipelines is common from low-level hardware architectures, to high-level parallel programming. A pipeline is series of processing elements, where the output of one element is the input of the next one. In the parallel programming community, each of these processing elements is a worker of a parallel process.

The synchronisation details between the different stages of a pipeline depend again on the particular implementation of a pipeline skeleton. A common approach (e.g. Fastflow [ADK+11]) uses an intermediate buffer between the stages of a parallel pipeline. In [HBS16], an intermediate queue is used, used concurrently by the first and second workers. This is illustrated in Figure 2.2b on page 31. For each input, the stages in a pipeline are activated sequentially, as the tasks flow through the pipeline. However, both stages of the pipeline can perform their operations in parallel to the different inputs.

An example of parallel pipeline is an image stream processing application that applies a series of filters, followed by an edge detection algorithm, to each element of an input stream of images:

```
process :: Stream Img -> Stream Img
process = fun filter1 || fun filter2 || fun edge
```

In this example, `process` proceeds as follows:

1. The first worker, `fun filter1` takes an image from the input stream of images, applies `filter1` to it, and puts the result into an intermediate queue.

2. The second worker, `fun filter2` takes an image from the intermediate buffer, and applies `filter2` to it, placing it in another intermediate

queue. The first worker, `fun filter1`, can take in parallel the next image from the input stream and process it.

3. The third worker, `fun edge` takes the output of `fun filter2` and applies function `edge` to it. In parallel, `fun filter1` and `fun filter2` can proceed with their respective inputs.

Note that the definition of the intermediate queues is crucial to the parallel application. The usage of bounded queues would require for the first stages of the parallel application to wait until there is free space on it, while the usage of unbounded queues could result on large amounts of tasks being stored in the intermediate buffers. However, these details can be dealt with by the programmer by selecting a suitable implementation of parallel pipelines, but the low-level details of this are delegated to the skeleton implementation.

Algorithmic skeletons can be nested. Suppose the `fun edge` worker runtime dominates. In order to speed up the parallel application, a task farm could be applied to this stage, resulting in the following skeletal program:

```
process :: Stream Img -> Stream Img
process = fun filter1 || fun filter2 || farm 4 (fun edge)
```

Feedback Skeleton

Some applications perform the same operation iteratively, until some dynamic condition is met. Iteration is captured by the feedback pattern. Essentially, a feedback loop allows the output of a parallel program to be passed to an earlier stage as an input. This pattern allows decoupling the body of an iterative computation from the iteration condition. As a result, each iteration can be performed in parallel for different inputs.

In a queue-based model, an implementation of the feedback pattern as an algorithmic skeleton is realised as a connection from the output queue of a parallel structure to the input queue of the stage where the iteration happens. Consider the earlier image stream processing skeletal program:

```
process :: Stream Img -> Stream Img
process = fun filter1 || fun filter2 || farm 4 (fun edge)
```

Suppose that the function applied by the second stage, `filter2`, iteratively applies a filter, until the image meets some dynamic condition. This is a common case, for example, in many iterative methods for image deblurring [NPP04, NCT99, BLM90]. The program could be further parallelised by applying a farm with a feedback loop. Suppose that `filter2'` represents a single iteration of `filter2`, together with a tag that indicates if the dynamic condition is met by the output or not. The program could be parallelised as follows:

```
process :: Stream Img -> Stream Img
process = fb (fun filter2')
```

In this example, the construct `fb` inspects the tag of the output of `filter2'`, and enqueues the element back to the input, or to the output queue, depending on whether the need of further processing. To parallelise this function, a task farm can be used:

```
process :: Stream Img -> Stream Img
process = fb (farm 3 (fun filter2'))
```

Note that even if the `farm` returns the results in order, since each image in the input stream will require a different number of iterations, the output of `fb` does not necessarily respect the order of the input. This, as in the task farm case, needs to be handled by the master process/thread.

Parallel Divide and Conquer Skeleton

A common pattern of computation is the *divide and conquer* pattern. In this pattern, a program takes an input, divides it into different sub-input, and proceeds recursively in each sub-input until a base case is reached. When the base case is reached, an operation is performed to it. When the computation is performed to the sub-inputs, the outputs are then combined into a single output. There are many ways of implementing divide and conquer algorithmic skeletons. Danelutto and Torquati [DT13] show how it can be implemented in terms of pipelines and feedback loops. In his thesis, Christoph Herrmann presents a hierarchy of divide and conquer functional skeletons that have efficient C+MPI implementations [Her00]. An implementation used in [CHSA17] uses intermediate queues to communicate be-

tween the different *divide* and *combine* stages, as shown in Figure 2.3 on page 32.

2.2.6 Advantages of Structured Parallel Programming

Message passing and shared variable concurrency methods contain many potential sources of errors, some of which are non-obvious. A comprehensive and systematic exploration of common errors in OpenMP was done by Süß and Leopold [SL08]. Süß and Leopold classified programming mistakes into two categories: *correctness mistakes*, such as accessing a shared variable that is not protected, and *performance mistakes*, such as unnecessarily protecting a region that does not need to be protected. Correctness mistakes may lead to *race conditions*, where a parallel program will return inconsistent results, depending on the execution order, or *deadlocks*, where a program will block indefinitely.

In the message passing approach, similar kinds of problems may appear. Consider the following fragment of C+MPI program:

```
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
   MPI_Send (sendbuf, count, MPI_INT
             , 1, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT
             , 1, tag, comm, &status);
} else if (my_rank == 1) {
   MPI_Send (sendbuf, count, MPI_INT
             , 0, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT
             , 0, tag, comm, &status);
}
```

In this code, there are two threads running in parallel, one running the fragment of code under `my_rank == 0`, and another thread running the fragment under `my_rank == 1`. Both threads will send a message to each other, and then receive a message from the other thread. A naïve program-

mer might think that `MPI_Send` is a non-blocking operation. This is the case in many MPI implementations, where the code above will work with no problems. However, if this code is ported to an architecture where the implementation of the `MPI_Send` operation is blocks until a matching `MPI_Recv` is executed, then the above code will deadlock. Detecting deadlocks in message-passing approaches is a major and challenging problem that has been subject of much research, e.g. [LZC+02, NY16, FKNS14, LMM+15].

The source of most of those problems is the too fine-grained control that is provided by concurrency approaches over how a program is parallelised. This similar to Gorlatch's argument in his paper *Send-receive Considered Harmful* [Gor04]. A programmer must always take low-level decisions on how/when are the threads created, how do they communicate, what is protected by a lock, etc.

Algorithmic skeletons [Col89], as implementations of patterns of parallelism, already handle implicitly low-level thread/process creation, synchronisation and communication details. Algorithmic skeletons allow the programmer to focus on the high-level structure of the parallel computation. Algorithmic skeleton implementations handle. Provided that their implementation is correct, the usage of algorithmic skeletons remove *by construction* deadlocks and race conditions.

2.3 Structured Recursion

This section presents a brief overview on recursion patterns, or recursion schemes, from a functional programming perspective. The presentation in this section is based on [MFP91].

2.3.1 A Categorical Interpretation

The basic types and primitive operations used throughout this thesis are derived from a standard categorical interpretation. A brief overview of this is provided in this section. A more in-depth introduction to the category theory background can be found in any introductory book, such as [Awo10].

A *category* $\mathcal{C}$ consists of:

- a collection of *objects*, $\mathsf{ob}(\mathcal{C})$;
- a collection of *morphisms*, $\mathsf{hom}(\mathcal{C})$, also known as *arrows* or *maps*; and,
- an operation, $\circ$, called the *composition of morphisms*.

Objects are generally written $A, B, \ldots$. If an object A is in a category $\mathcal{C}$, the notation used is $A \in \mathsf{ob}(\mathcal{C})$, or $A \in \mathcal{C}$. Morphisms are generally written $f, g, \ldots$. Each morphism f has a *source object* A, and a target object B. The notation used for stating that "f is a morphism from A to B" is $f : A \to B$. The collection of morphisms from A to B in some category $\mathcal{C}$ is written $\mathsf{hom}_{\mathcal{C}}(A, B)$, or $\mathsf{hom}(A, B)$ whenever there is no ambiguity about the category of objects A and B. The collection of morphisms from A to B, $\mathsf{hom}(A, B)$ is called the *hom-class* of all morphisms from A to B. The composition operator, $\circ$, is a binary operation $\mathsf{hom}(A, B) \times \mathsf{hom}(B, C) \to \mathsf{hom}(A, C)$, that takes a morphism from A to B and a morphism from B to C to another morphism from A to C. The composition operator in $\mathcal{C}$ must satisfy some properties for $\mathcal{C}$ to be a category:

- *associativity*, if $f : A \to B$, $g : B \to C$ and $h : A \to C$, then $f \circ (g \circ h) = (f \circ g) \circ h$; and
- *identity*, for every object A, there is a morphism $\mathtt{id}_A : A \to A$ such that for all $f : A \to B$ and $g : C \to A$, $f \circ \mathtt{id}_A = f$ and $\mathtt{id}_A \circ g = g$.

A common category for studying the semantics of non-strict functional programming languages is the category of *pointed complete partial orders*, and *continuous functions* (*CPO*) (see e.g. [Mit96]). A *complete partial order* is a set, D, together with a relation $\sqsubseteq \subseteq D \times D$ that is reflexive, transitive and antisymmetric, such that each chain has a least upper bound in D. In the category *CPO*, objects are CPOs, and morphisms are *continuous functions* [Sto77]. The categorical semantics of a programming language interprets types as objects, and functions as morphisms. In this categorical interpretation, the partial order $x \sqsubseteq y$ denotes that x is less defined than y, or that x is an approximation of y. The basic types and combinators that

$$A, B, \ldots ::= \mathsf{T} \quad | \quad A \to B \quad | \quad A \times \cdots \times B \quad | \quad A + \cdots + B \quad | \quad F\ A \cdots B \quad | \quad \mu F$$

Figure 2.4: Basic categorical types.

are going to be used throughout this thesis are derived from this categorical interpretation, and will be introduced later in this section.

Functors Functors are used to model type constructors. In the categorical interpretation, a functor F is a mapping between categories $\mathcal{C}$ and $\mathcal{D}$ that

- associates objects from one category, $X \in \mathcal{C}$, to objects of the other category $F\ X \in \mathcal{D}$; and
- associates each morphism $f : X \to Y$ in $\mathcal{C}$ to a morphism $F\ f : F\ X \to F\ Y$ in $\mathcal{D}$, such that:
  - $F\ \mathtt{id}_X = \mathtt{id}_{F\,X}$ for all object X, i.e. F preserves the identities; and,
  - $F\ (f \circ g) = F\ f \circ F\ g$ for all morphisms f and g, i.e. F preserves compositions.

A functor $F : \mathcal{C} \to \mathcal{C}$ between a category $\mathcal{C}$ and itself is called an *endofunctor*. In functional programming languages, the morphism $F\ f$ is generally written $\mathtt{map}\ f$. The notions of functors can be generalised to multiple arguments. For example, a bi-functor is a mapping that takes two categories $\mathcal{C}_1$ and $\mathcal{C}_2$ into a category $\mathcal{D}$. In general, a multi-functor is a mapping from n categories.

2.3.2 Basic Types and Combinators

Types Figure 2.4 on page 40 shows our basic types and combinators. They have a standard categorical interpretation. The type T denotes a primitive type. The type $A \to B$ is interpreted as the domain of continuous functions from A to B, the type $A \times B$ is a tuple of objects A and B, and is interpreted as the cartesian product of A and B. The type $A + B$ is

interpreted as the separated sum of A and B. The type $A{+}B$ is similar to the usual `Either` A B type in Haskell. Here, product and sum types are generalised to n arguments. The type $F\ A \cdots B$ is the n-functor F applied to types $A \cdots B$. Finally, the type μF is the fixpoint of functor F, and is used to model recursive datatypes. Note that we make no distinction between *greatest* or *least* fixpoint, since in the category *CPO* data and co-data coincide.

n-functors Functors of n arguments are defined using a pointed notation, or by *sectioning* another functor. The expression $\Lambda V_1 \cdots V_n.\underline{A}$ denotes a functor that takes $V_1 \cdots V_n$ parameters, and returns a new type $\underline{A}$. Here, the notation $\underline{A}$ denotes a subset of the types A, where there are no arrows of the form $B \to C$ for any B and C, thus only allowing *regular functors* [Mee96].

$$F ::= \Lambda V_1 \cdots V_n.\ \underline{A} \quad | \quad F\ A_1 \cdots A_n$$

$$\underline{A}, \underline{B}, \ldots ::= \mathsf{T} \quad | \quad V \quad | \quad \underline{A} \times \cdots \times \underline{B} \quad | \quad \underline{A} + \cdots + \underline{B} \quad | \quad F\ \underline{A} \cdots \underline{B} \quad | \quad \mu F$$

An example of functor in Haskell is the type of trees of some type A, i.e. `Tree` A. The type μF is a recursive type, and denotes the least fixpoint of some functor F. Listing 2.1 presents a recursive datatype in Haskell, `List`, a possible definition of the μ type, and a functor `L`.

Listing 2.1 Haskell List type and base functor

```
1  data List = Nil | Cons Int List
2
3  data μ f = In (f (μ f))
4  data L a = N | C Int a
```

The type `List` can be defined as the least fixpoint μ `L`. For example, the list of elements `[1,2]` can be represented in the following two ways:

```
Cons  1 (Cons  2  Nil)      :: List
In (C 1 (In (C 2 (In N)))) :: μ L
```

$$p_i \in \mathcal{P} ::= \pi_i^j \mid \mathtt{inj}_i^j \mid \&_i \mid \triangledown_i \mid \mathtt{in}_F \mid \mathtt{out}_F$$

Figure 2.5: Primitive Operations

Primitive Combinators The primitive product and coproduct operations are treated in a slightly non-standard way in this thesis. Figure 2.5 on page 42 summarises the basic product and coproduct operations. Note that product and coproduct operations are generalised to more than two arguments. The superscript j on π_i^j and $\mathtt{inj}_i^j$, in Figure 2.5, denotes the number of components of the product/coproduct types, and it will be omitted whenever there is no ambiguity. A standard treatment of product types is defined as the cartesian product, and requires the definition of tuple projections

$$\pi_1\ (x_1, x_2) = x_1$$
$$\pi_2\ (x_1, x_2) = x_2$$

and the *split* function:

$$(f\ \triangle\ g)\ x = (f\ x, g\ x).$$

The split function is defined in terms of the *tuple constructor*. Here, the tuple constructor is denoted by &, and the split function is defined as follows:

$$\&_i\ x_1 \cdots x_i = (x_1, \ldots, x_i) \quad (\triangle_i\ \ f_1 \cdots f_i)\ x = \&_i\ (f_1\ x) \cdots (f_i\ x).$$

The standard treatment of coproducts is by defining them as a disjoint union, together with two kinds of operations: coproduct *injections*, and the *either* combinator. The primitive $\mathtt{inj}_i^j$ denotes the injection into a separated sum of j types. It is the constructor of an either type of j elements. Again, whenever there is no ambiguity, the superscript j will be omitted. The *either* combinator, $\triangledown_i$ "pattern-matches" an input *either* type, selects the appropriate function, and applies it to the input:

$$(\triangledown_i\ f_1 \cdots f_i)\ (\mathtt{inj}_j\ x) = f_j\ x$$

Finally, the operations $\mathtt{in}_F : F\ (\mu F) \to \mu F$ and $\mathtt{out}_F : \mu F \to F\ (\mu F)$ capture the isomorphism between $F\ (\mu F)$ and μF. These functions will take different interpretations in Chapter 3 and Chapter 5, i.e. the initial algebras are interpreted differently. However, since initial algebras are equivalent, this distinction does not affect the results of this thesis.

2.3.1 Example | Lists. Given the bifunctor

$$L\ A\ B = 1 + A \times B,$$

the polymorphic List data type is defined by the fixpoint of $L\ A$:

$$\mathsf{List}\ A = \mu(L\ A).$$

The two list constructors are defined as expected:

$$\begin{array}{lcl} \mathtt{nil} & : & \mathsf{List}\ A \\ \mathtt{nil} & = & \mathtt{in}_{L\,A}\ (\mathtt{inj}_1\ ()) \end{array}$$

$$\begin{array}{lcl} \mathtt{cons} & : & A \to \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathtt{cons}\ x\ l & = & \mathtt{in}_{L\,A}\ (\mathtt{inj}_2\ (x, l)) \end{array}$$

The usual notation for lists is used

$$[x_1, x_2, \ldots, x_n] \ = \ \mathtt{cons}\ x_1\ (\mathtt{cons}\ x_2\ (\mathtt{cons}\ \ldots\ (\mathtt{cons}\ x_n\ \mathtt{nil}))).$$

2.3.2 Example | Trees. The polymorphic binary tree type can be defined in an analogous way:

$$\begin{array}{lcl} T\ A\ B & = & 1 + A \times B \times B \text{ where} \\ \mathsf{Tree}\ A & = & \mu(T\ A). \end{array}$$

The two tree constructors are defined below:

$$\begin{array}{lcl} \mathtt{empty} & : & \mathsf{Tree}\ A \\ \mathtt{empty} & = & \mathtt{in}_{T\,A}\ (\mathtt{inj}_1\ ()) \end{array}$$

$$\begin{array}{lcl} \mathtt{node} & : & \mathsf{Tree}\ A \to A \to \mathsf{Tree}\ A \to \mathsf{Tree}\ A \\ \mathtt{node}\ t_1\ x\ t_2 & = & \mathtt{in}_{T\,A}\ (\mathtt{inj}_2\ (x, t_1, t_2)) \end{array}$$

2.3.3 Recursion Patterns

Recursion schemes are defined in terms of the basic types and combinators in Figures 2.4 and 2.5, presented in previous sections. The recursion

schemes used in this thesis are *catamorphisms*, *anamorphisms* and *hylomorphisms*. A catamorphism is a recursive function that uses primitive recursion to consume an input data type, e.g. adding all the elements of a list of integers. An *anamorphism* is the dual function, that takes an input value and generates an output data type, e.g. generating a list of numbers from zero up to some value. A *hylomorphism* is a *divide and conquer* recursive function, where the *divide* part can be split into an anamorphism, and the combine part into a catamorphism. The map function for polymorphic recursive datatypes can be defined in terms of catamorphisms.

F-Algebras(Coalgebras) Given a functor F, an F-algebra is a pair (A, f), where A is an object, and f is a morphism $f : F\ A \to A$. For example, in the context of the basic types and combinators, consider the following functor L:

$$L\ X = 1 + \texttt{Int} \times X$$

This functor takes a type, e.g. A, to the type $1 + \texttt{Int} \times A$. An L-algebra in the context of types is a pair of a type A, and a function $f : L\ A \to A$. Consider the following function:

$$\begin{array}{ll} \texttt{sum} & : L\ \texttt{Int} \to \texttt{Int} \\ \texttt{sum}\ (\texttt{inj}_1\ ()) & = 0 \\ \texttt{sum}\ (\texttt{inj}_2\ (i, j)) & = i + j \end{array}$$

The pair $(\texttt{Int}, \texttt{sum})$ is an L-algebra.

The dual concept is that of a *coalgebra*. Given a functor G, a G-coalgebra is a pair (B, g), where B is an object, and g is a morphism $g : B \to G\ B$. An example in terms of types and functions, consider again the functor L. The pair $(\texttt{Int}, \texttt{next})$ is an L-coalgebra, where the function $\texttt{next}$ is defined below:

$$\begin{array}{ll} \texttt{next} & : \texttt{Int} \to L\ \texttt{Int} \\ \texttt{next}\ n & = \texttt{if}\ n = 0\ \texttt{then}\ \texttt{inj}_1\ () \\ & \qquad \texttt{else}\ \texttt{inj}_2\ (n, n-1) \end{array}$$

Homomorphisms A homomorphism between algebras (A, f) and (B, g) is a function $h : A \to B$ such that

$$h \circ f = g \circ F\ h.$$

Dually, a homomorphism between coalgebras (A, f) and (B, g) is a function $h : A \to B$ such that

$$F\ h \circ f = g \circ h.$$

Initial(Final) (Co-)Algebra An F-algebra (A, f) is *initial* if, for any F-algebra (B, g), there is a unique homomorphism from (A, f) to (B, g). There may be more than one initial algebra, but they are all equivalent, since there must be a unique homomorphism between them. An F-coalgebra (A, f) is terminal if there is a unique homomorphism from any other F-coalgebra (A, g) to (A, f). In the types and basic combinators defined in Figures 2.4 and 2.5, the pair $(\mu F, \mathtt{in}_F)$ is an initial F-algebra:

$$\mathtt{in}_F : F\mu F \to \mu F.$$

Dually, the pair $(\mu F, \mathtt{out}_F)$ is a terminal coalgebra, where

$$\mathtt{out}_F : \mu F \to F\mu F.$$

Note that there is no distinction between "greatest" or "least" fixpoint. In the semantic model of *CPO*, datatypes (i.e. those formed from initial algebras) and co-datatypes (i.e. those built from terminal co-algebras) coincide, this is a property called *algebraic compactness*, and makes it possible to combine datatypes and co-datatypes, with some limitations that are discussed in Chapter 3. This implies that $\mathtt{in}_F$ and $\mathtt{out}_F$ are inverses:

$$\mathtt{in}_F \circ \mathtt{out}_F = \mathtt{id} \qquad\qquad \mathtt{out}_F \circ \mathtt{in}_F = \mathtt{id}.$$

Catamorphisms Given the initial F-algebra $(\mu F, \mathtt{in}_F)$, there is a unique homomorphism to any other F-algebra (B, g). By the definition of homomorphisms, this implies that there must be a function $h : \mu F \to B$ such that

$$h \circ \mathtt{in}_F = g \circ F\ h,$$

and this function h must be unique. This h is a *catamorphism*:

$$\begin{aligned} &\mathtt{cata}_F\ g = h \\ &\quad \text{where } h \circ \mathtt{in}_F = g \circ F\ h \end{aligned}$$

$$
\begin{array}{ll}
& \mathtt{cata}_L\ \mathtt{sum}\quad (\mathtt{in}_L\ (\mathtt{inj}_2\ (3, \mathtt{in}_L\ (\mathtt{inj}_2\ (2, \mathtt{in}_L\ (\mathtt{inj}_1\ ())))))) \\
= & \quad \{\ \text{By the definition of } \mathtt{cata}_L\ \} \\
& (\mathtt{sum} \circ L\ (\mathtt{cata}_L\ \mathtt{sum}) \circ \mathtt{out}_L) \\
& \qquad (\mathtt{in}_L\ (\mathtt{inj}_2\ (3, \mathtt{in}_L\ (\mathtt{inj}_2\ (2, \mathtt{in}_L\ (\mathtt{inj}_1\ ())))))) \\
= & \quad \{\ \mathtt{in}_L \text{ and } \mathtt{out}_L \text{ are inverses}\ \} \\
& (\mathtt{sum} \circ L\ (\mathtt{cata}_L\ \mathtt{sum}))\quad (\mathtt{inj}_2\ (3, \mathtt{in}_L\ (\mathtt{inj}_2\ (2, \mathtt{in}_L\ (\mathtt{inj}_1\ ()))))) \\
= & \quad \{\ \text{Definition of } L \text{ functor}\ \} \\
& \mathtt{sum}\quad (\mathtt{inj}_2\ (3, \mathtt{cata}_L\ \mathtt{sum}\ (\mathtt{in}_L\ (\mathtt{inj}_2\ (2, \mathtt{in}_L\ (\mathtt{inj}_1\ ())))))) \\
= & \quad \{\ \text{Repeatedly applying } \mathtt{cata}_L\ \} \\
& \mathtt{sum}\quad (\mathtt{inj}_2\ (3, \mathtt{sum}\ (\mathtt{inj}_2\ (2, \mathtt{sum}\ (\mathtt{inj}_1\ ()))))) \\
= & \quad \{\ \text{By the definition of } \mathtt{sum}\ \} \\
& \mathtt{sum}\quad (\mathtt{inj}_2\ (3, \mathtt{sum}\ (\mathtt{inj}_2\ (2, 0)))) \\
= & \quad \{\ \text{By the definition of } \mathtt{sum}\ \} \\
& \mathtt{sum}\quad (\mathtt{inj}_2\ (3, 2)) \\
= & \quad \{\ \text{By the definition of } \mathtt{sum}\ \} \\
& 5
\end{array}
$$

Figure 2.6: Example of application of $\mathtt{cata}_L\ \mathtt{sum}$ to the list $[2, 3]$.

Since $\mathtt{in}_F$ and $\mathtt{out}_F$ are inverses, this can be simplified to the following expression:

$$
\begin{array}{l}
\mathtt{cata}_F\ g = h \\
\quad \text{where } h = g \circ F\ h \circ \mathtt{out}_F
\end{array}
$$

Essentially, $\mathtt{cata}_F\ g$ is a recursive function that takes a recursive datatype, μF, and returns a value of type B. This is done by first applying the $\mathtt{out}_F$ operation, and then applying $\mathtt{cata}_F\ g$ to the recursive positions of the resulting $F\ \mu F$. This will turn $F\ \mu F$ to $F\ B$. This output of type $F\ B$ is the passed to g, that returns the final value of type B. For example, recall the function $\mathtt{sum}$:

$$
\begin{array}{ll}
\mathtt{sum} & : L\ \mathtt{Int} \to \mathtt{Int} \\
\mathtt{sum}\ (\mathtt{inj}_1\ ()) & = 0 \\
\mathtt{sum}\ (\mathtt{inj}_2\ (i, j)) & = i + j
\end{array}
$$

The catamorphism $\mathtt{cata}_L\ \mathtt{sum}$ takes a value of type μL, i.e. a list of integers,

and returns the value that results of adding all of them. For example, the list $[3, 2]$ is represented in terms of the initial L-algebra as follows:

$$[3, 2] = \mathtt{in}_L\ (\mathtt{inj}_2\ (3, \mathtt{in}_L\ (\mathtt{inj}_2\ (2, \mathtt{in}_L\ (\mathtt{inj}_1\ ())))))$$

Applying $\mathtt{cata}_L\ \mathtt{sum}$ to this list proceeds as we show in Figure 2.6.

Polymorphic datatypes As was previously stated, the following definition of F is also a functor:

$$F\ A = \mu(G\ A)$$

This is a well-known fact (see e.g. [Gib02b]) that can be shown by defining the function $F\ f$ for any given function $f : A \to B$. This function $F\ f$ is going to be called $\mathtt{map}_F\ f$, which is more common in the functional programming community. Given a function $f : A \to B$,

$$F\ f = \mathtt{map}_F\ f = \mathtt{cata}_{G\ A}\ (\mathtt{in}_G\ \circ\ G\ f\ \mathtt{id})$$

In order to show that $\mathtt{map}_F\ f$ defines a functor, it must respect identities

$$\mathtt{map}_F\ \mathtt{id} = \mathtt{cata}_{G\ A}\ (\mathtt{in}_G\ \circ\ G\ \mathtt{id}\ \mathtt{id}) = \mathtt{cata}_{G\ A}\ \mathtt{in}_G = \mathtt{id},$$

and compositions,

$$\begin{aligned}
\mathtt{map}_F\ (f \circ g) &= \mathtt{cata}_{G\ A}\ (\mathtt{in}_G\ \circ\ G\ (f \circ g)\ \mathtt{id}) \\
&= \mathtt{cata}_{G\ A}\ (\mathtt{in}_G\ \circ\ G\ f\ \mathtt{id} \circ\ G\ g\ \mathtt{id}) \\
&= \mathtt{cata}_{G\ B}\ (\mathtt{in}_G\ \circ\ G\ f\ \mathtt{id}) \circ\ \mathtt{cata}_{G\ A}\ (\mathtt{in}_G\ \circ\ G\ g\ \mathtt{id}) \\
&= \mathtt{map}_F\ f \circ\ \mathtt{map}_F\ g
\end{aligned}$$

Anamorphisms Given the terminal F-coalgebra $(\mu F, \mathtt{out}_F)$, there is a unique homomorphism from any other F-coalgebra (A, f). By the definition of homomorphisms, that implies that there must be a function $h : A \to \mu F$ such that

$$F\ h \circ f = \mathtt{out}_F \circ h,$$

and this function h must be unique. This h is an *anamorphism*:

$$\begin{aligned}
&\mathsf{ana}_F\ f = h \\
&\quad \text{where } F\ h \circ f = \mathtt{out}_F \circ h.
\end{aligned}$$

Again, this can be simplified, since $\texttt{in}_F \circ \texttt{out}_F = \texttt{id}$:

$$\begin{aligned} &\texttt{ana}_F\ f = h \\ &\quad \text{where } h = \texttt{in}_F \circ F\ h \circ f. \end{aligned}$$

As an example of anamorphism, recall the function `next`:

$$\begin{aligned} \texttt{next} \quad &: \texttt{Int} \to L\ \texttt{Int} \\ \texttt{next}\ n \quad &= \texttt{if}\ n = 0\ \texttt{then}\ \texttt{inj}_1\ () \\ &\qquad\qquad \texttt{else}\ \texttt{inj}_2\ (n, n-1) \end{aligned}$$

The anamorphism $\texttt{ana}_L\ \texttt{next}$ takes an integer i as input, and generates a list of all numbers from i to 0. Note that if $i < 0$, this function would generate an infinite list.

2.3.4 Hylomorphisms

Hylomorphisms [MFP91] capture a generalisation of *divide and conquer* algorithms. Intuitively, $hylo_F\ f\ g$ is a recursive algorithm whose recursive call tree can be represented by the datatype μF, where g describes how the algorithm divides the input problem into sub-problems, and f describes how the results are combined.

$$\begin{aligned} \texttt{hylo}_F \quad &: \quad (F\ B \to B) \to (A \to F\ A) \to A \to B \\ \texttt{hylo}_F\ f\ g \quad &= \quad h \\ &\quad \text{where} \quad h = f \circ F\ h \circ g \end{aligned}$$

In this definition, the "divide" function g is applied, and the result is returned in the structure F. The hylomorphism is applied recursively to the sub-inputs in F, and the result is combined by the "combine" function f. From this definition, it is clear that instantiating f as $\texttt{in}_F$ results in an anamorphism, while instantiating g as $\texttt{out}_F$ results in a catamorphism. Catamorphisms, anamorphisms, and map recursion schemes are therefore particular instances of hylomorphisms.

$$\begin{aligned} \texttt{map}_T\ f \quad &= \quad \texttt{hylo}_{F\,A}\ (\texttt{in}_{F\,B} \circ (F\ f\ \texttt{id}))\ \texttt{out}_{F\,A} \\ &\quad\ \ \text{where } A = \mathit{dom}(f) \text{ and } B = \mathit{codom}(f) \\ &\quad\ \ \text{and } T\ A = \mu(F\ A) \\ \texttt{cata}_F\ f \quad &= \quad \texttt{hylo}_F\ f\ \texttt{out}_F \\ \texttt{ana}_F\ f \quad &= \quad \texttt{hylo}_F\ \texttt{in}_F\ f \end{aligned}$$

2.3.3 Example | Factorial. We show below the definition of a partial factorial function using explicit recursion:

$$\begin{array}{lcl} \texttt{fact} & : & \texttt{Int} \to \texttt{Int} \\ \texttt{fact}\ n & = & \texttt{if}\ n = 0 \\ & & \quad \texttt{then}\ 1 \\ & & \quad \texttt{else}\ n \times \texttt{fact}\ (n-1) \end{array}$$

The equivalent definition, as a hylomorphism, is as follows:

$$L\ A = 1 + \texttt{Int} \times A$$

$$\begin{array}{lcl} \texttt{split} & : & \texttt{Int} \to L\ \texttt{Int} \\ \texttt{split}\ n & = & \texttt{if}\ n = 0 \\ & & \quad \texttt{then}\ \texttt{inj}_1\ () \\ & & \quad \texttt{else}\ \texttt{inj}_2\ (n,\ n-1) \end{array}$$

$$\begin{array}{lcl} \texttt{join} & : & L\ \texttt{Int} \to \texttt{Int} \\ \texttt{join}\ (\texttt{inj}_1\ ()) & = & 1 \\ \texttt{join}\ (\texttt{inj}_2\ (n,m)) & = & n \times m \end{array}$$

$$\texttt{fact} = \texttt{hylo}_L\ \texttt{join}\ \texttt{split}$$

The definition of recursive functions as a hylomorphisms makes explicit their recursive call-tree. In this example, it is a list of numbers from n to 0. This list of is also produced in the definition with explicit recursion, but it will be stored in the stack, as arguments to $\times$ and the recursive calls to `fact`.

2.3.4 Example | Quicksort. We assume a type A, and two functions:

$$\texttt{leq},\ \texttt{gt}\ :\ A \to \texttt{List}\ A \to \texttt{List}\ A,$$

The naïve quicksort implementation using explicit recursion is as follows:

$$\begin{array}{lcl} \texttt{qsort} & : & \texttt{List}\ A \to \texttt{List}\ A \\ \texttt{qsort}\ \texttt{nil} & = & [] \\ \texttt{qsort}\ (\texttt{cons}\ x\ l) & = & \texttt{qsort}\ (\texttt{leq}\ x\ l) +\!\!+ \texttt{cons}\ x\ (\texttt{qsort}\ (\texttt{gt}\ x\ l)) \end{array}$$

The following is an implementation of quicksort as a hylomorphism:

$$
\begin{array}{lcl}
T\ A\ B = 1 + A \times B \times B & & \\
\texttt{split} & : & \texttt{List}\ A \to T\ A\ (\textsf{List}\ A) \\
\texttt{split}\ \texttt{nil} & = & \texttt{inj}_1\ () \\
\texttt{split}\ (\texttt{cons}\ x\ l) & = & \texttt{inj}_2\ (x,\ \texttt{leq}\ x\ l,\ \texttt{gt}\ x\ l) \\
 & & \\
\texttt{join} & : & T\ A\ (\textsf{List}\ A) \to \textsf{List}\ A \\
\texttt{join}\ (\texttt{inj}_1\ ()) & = & \texttt{nil} \\
\texttt{join}\ (\texttt{inj}_2\ (x, l, r)) & = & l +\!\!+ \texttt{cons}\ x\ r \\
 & & \\
\texttt{qsort} = \texttt{hylo}_{T\,A}\ \texttt{join}\ \texttt{split} & &
\end{array}
$$

The recursive call-tree of quicksort is a binary tree. Note that the full call tree is never produced, and is only produced as required. The recursive calls to quicksort will happen in the B positions of the T bifunctor, sequentially: first the left sublist will be sorted, and then the second sublist. This behaviour is *almost exactly* the same as the explicit recursive function. The only difference is that the intermediate results of `split` will be stored in a datatype $T\ A$ (List A). In contrast, the intermediate results of `qsort` will be stored in the stack, as arguments to $+\!\!+$ and the recursive calls to `qsort`.

There are other kinds of recursion schemes, some of which subsume hylomorphisms [MFP91, HWG15, Hin10]. However, for the purposes of this thesis, hylomorphisms are general enough, as will be described in Chapter 3.

2.3.5 Program Calculation Approaches

Program calculation techniques and recursion patterns have been successfully studied in a number of different domains. The framework DrHylo [CPP05, Cun05] implements a translation scheme from typing judgements to a point-free representation of the corresponding point-wise program, based on [HIT96] and [Cun05]. In their framework, a pointed to point-free program transformation is explored, and program transformations/optimisations are studied within the point-free framework of DrHylo. Bahr and Hutton [BH15] show how to derive compilers from simple, compositional definitions of the high-level semantics of a programming language. Hackett and Hutton [HH15]

developed a reusable version of the well-known worker-wrapper transformation [GH09] that can be instantiated by a number of recursion operators. They also develop their own version of the improvement theory for reasoning about the efficiency of the optimised programs, which does not rely on an operational semantics.

Program Calculation Techniques and Parallelism Program calculation techniques have also played an important role in structured parallelism. The third homomorphism theorem, list homomorphisms, and the Bird-Meertens Formalism are amongst the many techniques that have been explored for calculating parallel programs [GL95, HIT97, HTC98, KC98, MHT06b, Mis94, MM10, Rei93, Ski93b, Ski91]. For example, the third homomorphism theorem states that if a function can be written as both a left fold and a right fold, then it can also be evaluated in a divide-and-conquer manner [Gib96b]. This theorem has many potential applications for automatic parallelism [CM11, GG99, Gib96a, Gor99, LHM11, Mor13, MMM$^+$07]. Some of these approaches will be discussed again in the next section, in the context of algorithmic skeletons. Work on the third list homomorphism theorem focuses on applying particular non-trivial rewritings. The work presented in this thesis has a slightly different focus: it is aimed at defining a general framework for encoding a number of different rewritings, not at performing a particular program transformation rule.

Pointed to point-free transformations and hylomorphism derivation techniques [HIT96] are related to a field known in the structured parallelism community as *pattern discovery*. Bozo et al. [BFH$^+$14] find instances of *map* and *fold* in Erlang programs, as potential sources of parallelism. Barwell's thesis [Bar17] explores this issue in more depth, using a technique called *antiunification* for finding instances of recursion patterns in recursive functions.

Kannan and Hamilton [KH17] describe a technique for deriving parallel implementations using algorithmic skeletons, using a technique called *distillation*. Distillation is a program transformation technique similar to *deforestation* for reducing the amount of intermediate data structures in a computation. Kannan and Hamilton use distillation [Ham07] to first opti-

mise the intermediate data structures, and then use an encoding technique for merging the input of a program into a single input of a data type that matches the structure of the program. Kannan and Hamilton take the reverse approach to the one described in this paper. They *first* optimise a program using distillation [Ham07], and then they parallelise it using a datatype that captures the program structure. In our approach, we first *decompose* a program into all individual pieces using *reforestation*, and then study how these pieces can be recombined and parallelised. While both approaches expose many opportunities for parallelism, our approach allows programmers to explore many alternative parallelisations in terms of different combinations of algorithmic skeletons.

2.4 Algorithmic Skeleton Frameworks

There are many different algorithmic skeleton implementations, frameworks and languages. Some of them are targeted at different architectures, some focus on data parallelism, while others focus on task parallelism, or combined data and task parallelism. This section provides a brief overview of a number of current algorithmic skeleton frameworks, based on [GzVL10], and extended with recent approaches. Finally, these frameworks are compared based on their usage of the connection between algorithmic skeletons and patterns of recursion, on their support for code rewriting and performance analysis, and on whether they support some degree of automatic parallelisation.

The algorithmic skeleton frameworks that are presented in this section are categorised w.r.t. the following four characteristics:

1. *Abstraction:* is the skeleton framework high-level, or requires programmers to supply low-level details, e.g. selecting a scheduling policy for task farms.

2. *Predictability:* is there any support for static prediction of the runtime performance of programs? Is the underlying model predictable?

3. *Flexibility:* does the framework provide support for defining new algorithmic skeletons? Are the new skeletons defined as combinations of

simpler skeletons? Can the programmer define their operational behaviour? How easy is it to change the parallel structure of a program?

4. *Generality:* how general are the skeletons provided by the skeletal library? What is the most general recursion pattern that the skeletons provided can capture? Does the framework target a single architecture or multiple architectures?

Most of approaches trade a high-level of abstraction for predictability, with the exception of the OSL library, since the skeletons are built on top of the BSP model. Many skeletal approaches are very general, since they can be applied to a wide range of problems and multiple target architectures. However, most of the approaches are not very flexible, since they do not provide mechanisms for code rewriting, or allow the specification of new skeleton in terms of other simpler skeletons, but they do not provide abstractions to specify their operational behaviour.

2.4.1 The Need for a General Framework

Table 2.1 on page 56 contains a summary of the comparison of the algorithmic skeleton frameworks discussed in this section. The comparison takes into account a number of characteristics that are derived from the connection between algorithmic skeletons and recursion patterns, and cost models. These characteristics are the following:

Parallel Behaviour refers to whether the algorithmic skeleton framework provides abstractions for defining/implementing the operational behaviour of new basic skeletons. This does not take into account whether the framework is extensible or by defining more complex structures by nesting simpler skeletons. If an algorithmic skeleton framework relies on a direct low-level mechanism, such as C+MPI for implementing the parallel behaviour of algorithmic skeletons, then the parallel behaviour of skeletons in that framework is considered *built-in*.

Functional Behaviour specifies whether the algorithmic skeleton framework allows the specification of the functional behaviour of new algorithmic skeletons.

Cost Models refers to the existence of static performance analysis tools or techniques for this specific framework, and whether they are formally derived from the operational behaviour of the corresponding algorithmic skeletons.

Almost all frameworks for algorithmic skeletons in Table 2.1 rely on a built-in parallel behaviour of the algorithmic skeletons. Most of these approaches rely on a fixed set of algorithmic skeletons, implemented as a library using a low-level technique, such as C+MPI. Some of the algorithmic skeleton frameworks compile instances of high level patterns of recursion to low level code, such as SkelML, Parallel SML or the Lift data-parallel language. The Skipper framework separates the definition of the functional behaviour of an algorithmic skeleton from its operational behaviour, which is described as *Process Network Templates* [SG02]. However, Skipper still uses a limited pre-fixed set of skeletons.

A similar situation happens with respect to the functional behaviour of algorithmic skeletons. Many of the approaches rely implicitly on the functional behaviour of algorithmic skeletons, and do not allow the definition of new skeletons in terms of patterns of recursion. Many of them use a fixed set of algorithmic skeletons, and rely on this functional behaviour implicitly for automating code transformations. An example of exception of this is the AS framework [MMT15], which allows the extension of new algorithmic skeletons, as long as they correspond to *map* and *reduce* patterns of recursion.

An exception is the *Eden* functional programming language. As a parallel functional programming language, new HOFs that implement patterns of parallelism can be defined. The functionality of the new skeletons are then defined by Eden expressions, and the parallel behaviour is specified in terms of the *spawn* function.

Finally, many algorithmic skeleton frameworks provide some form of cost models, or static performance prediction for parallel programs within the framework. However, those cost models are derived by measurement and approximation, and they need to be developed and tested for each new parallel structure that is introduced to the language. Exceptions of this are the OSL framework and Lithium. In the OSL framework, the usage of

the BSP model makes it possible to derive cost models for new algorithmic skeletons within this model. For the Lithium framework, an operational semantics of algorithmic skeletons was developed [AD07]. This operational semantics is aimed at reasoning about different aspects of skeletal programs, derived both from the functional and parallel behaviour of programs. This operational semantics provides, therefore, a mechanism that can be used for reasoning about performance of parallel programs.

The need for a general framework. None of the existing frameworks provides, *at the same time*, abstractions for:

1. specifying the functional behaviour of parallel programs, i.e. the *denotational semantics*;

2. specifying the parallel behaviour of parallel programs, proving that it is consistent with the functional behaviour, i.e. an *operational semantics*;

3. formally deriving *cost models* from the operational semantics.

The closest to providing all is Lithium, by using the operational semantics of skeletons defined in [AD07], as a labelled state-transition system. However, combining both the functional and parallel behaviour makes it more complicated to reason about code rewritings than relying on a specification of the functional behaviour in terms of well-known patterns of recursion, while keeping the parallel behaviour at a high-level that makes it difficult to reason about performance in terms of low-level characteristics of parallel programs. The frameworks that operate by compiling HOFs to low-level parallel code do so without providing any abstraction for the programmer to control the resulting parallel code, and use automatic optimisation mechanisms.

eSkel

The Edinburgh Skeleton Library (eSkel) [BCGH05, Col04] is a library aimed at structured parallel programming that offers a range of algorithmic skeletons, implemented in terms of C+MPI, including farms and pipelines.

	Parallel Behaviour	**Functional Behaviour**	**Cost Models**
StA	*queue-based*	*hylomorphisms*	yes
AS	task graphs	map/reduce	yes
FastFlow	built-in	implicit	yes
$\mathcal{HDC}$	built-in	d&c	no
P^3L	built-in	implicit	yes
SkeTo	built-in	list/tree/matrix homomorphisms	yes
SkelML	built-in	map/fold	yes
Parallel SML	built-in	map/fold	yes
eSkel	built-in	implicit	no
OSL	parallel vectors	map/reduce/zip/...	yes
SCL	built-in	implicit	no
Eden	*spawn* function	Eden exprs.	no
Lithium/Muskel	built-in	implicit	yes
Skipper	Process Network Templates	built-in	yes
Quaff	built-in	implicit	yes
FAN	built-in	map/reduce/scan	yes
RPL	built-in	implicit	yes
Lift	built-in	map/reduce	no
PType	built-in	list/tree homomorphisms	no

Table 2.1: Algorithmic Skeleton Approaches to Parallelism

Skeletons in eSkel can be transient, i.e. instantiated and then destroyed for each use, or persistent, i.e. the skeleton is instantiated once and reused multiple times. The skeletons in eSkel accept implicit interaction modes, defined by the nesting of algorithmic skeletons, or explicit interaction modes, where the standard flow of data defined by a skeleton can be modified. The library eSkel is aimed at writing parallel software by using high-level constructs, but it is not aimed at providing automatic support for code rewriting. The eSkel library provides high-level constructs for parallel programming. However, each skeleton has a single implementation. Although it is possible to port these constructs for other architectures, or using alternative implementations, it is not very general or flexible. Finally, eSkel was built with the purpose of making parallel programming easier by providing high-level constructs, but deriving accurate cost models for these constructs was not in their original goals.

FastFlow

FastFlow [ADK+11, ADKT13] is a framework for C++ developed at the University of Pisa, aimed at developing skeletal programs, and originally focused on stream-based parallel computations. The FastFlow library provides the usual farm and pipeline skeletons, as well as feedback loops, and more complex structures that can be implemented in terms of the basic skeletons, such as a *divide and conquer* skeleton. Skeletons in FastFlow are highly configurable. Task farms are one example of this, since farm instances can use multiple different scheduling strategies. Another example is the possibility to control (i.e. pin) threads to cores within the FastFlow framework, and not just relying on GCC intrinsic operations. Tools for refactoring FastFlow programs into more efficient forms were developed as part of the ParaPhrase project [HAB+11]. One important novelty of FastFlow is the development of a RISC approach to parallel programming: complex algorithmic skeletons can be defined by using a number of small, basic parallel skeletons/components within the FastFlow framework [DT13].

The SPar language [GDTF17] is a DSL for annotating C++ programs with parallel annotations. The SPar annotations allow programmers to separate different stages in the computation, specify regions that are par-

allelised with streaming skeletons, and the degree of parallelism within a region. The toolset for the SPar language can be used to generate FastFlow code from annotated programs.

The RISC approach provides flexibility to the library, although programmers choose alternative implementations for the skeletons. FastFlow provides high-level constructs, but allow controlling low-level details. Cost models can be easily defined for FastFlow, but the low-level details make it difficult to make these cost models fully general. Summarising, FastFlow is very flexible, but not as high-level or general as other approaches.

OSL

The Orléans Skeleton Library is a library for skeletal programming in C++, build on top of MPI [JL11b]. OSL is based on the Bulk Synchronous parallel model [Val89, Val90], a parallel model that is characterised by being very predictable, due to the very regular computation, communication and synchronisation structure. A formal semantics has been developed for OSL [JL11a] in the Coq theorem prover [BC13], and it can be used to verify the correctness of parallel programs in the OSL library [JL12]. The OSL library is very high-level, and the usage of the BSP model makes it potentially very predictable, as the StA framework developed for this thesis. However, it is not as flexible or general, since skeletons are constrained to a specific model of parallelism.

SCL

The Structured Coordination Language [DFH+93, DGTY95] is one of the first languages for structured parallelism, and it is designed as a DSL for parallelism that needs to be integrated in a host language. SCL offers a number of data parallel skeletons such as map, scan and fold, as well as task parallel skeletons such as task farms. In the context of SCL, a number of program transformations were studied that allow applying a set of rules systematically to derive alternative parallelisations for a given parallel program. SCL is a very flexible framework, since it consists of a DSL whose constructs can accept multiple implementations.

HDC

The HDC language [Her00, HL00] is a language aimed at implementing parallel divide-and-conquer applications using a skeletal approach. It provides a hierarchy of parallel divide and conquer implementations, ranging from the more general versions, to more specific divide and conquer implementations with efficient implementations. The sizes of the sub-tasks and architecture-dependent parameters such as the number of processors are taken into account by the HDC compiler to generate parallel code.

Eden

Eden [BLOMPM96] is a parallel dialect of Haskell. Parallel programs are implementing by explicitly specifying a number of processes that communicate implicitly using unidirectional channels. Eden, as a functional language, has built-in support for algorithmic skeletons, which can be implemented as higher-order functions that implement different parallel strategies. In the context of Eden, Hammond et al. [HBL03] explored the use of meta-programming, paired with cost models, to automatically generate Eden skeleton instantiations that minimize the cost. However, they do not consider structural rewritings that change the parallel structure of a program.

Lithium and Muskel

Lithium [ADT03] and Muskel [DD06] are structured parallel programming frameworks in Java that provides nestable skeletons as Java libraries. A formal semantics of Lithium was developed [ADT03] that describes both the functional and parallel behaviour of Lithium programs as a labelled state transition system. Lithium supports parallel program optimisation by using a number of skeleton rewriting techniques [ADT03].

Skipper

Skipper [SGD99, SG02] is a skeletal programming framework aimed at developing parallel computer vision applications. Functions in skipper are implemented by using a purely functional specification of the algorithm, in

which parallelism is specified by calls to specific algorithmic skeletons. The basic sequential components of skipper applications are C functions. Skipper functions are compiled to a process graph, which is then mapped to a target architecture by using a third party application [Sor94].

P3L

P3L is a skeletal programming language that provides a number of basic algorithmic skeletons as language constructs [BDO+95]. It is described as a parallel coordination language, where the basic components are sequential C functions that are coordinated using a number of algorithmic skeletons. The parallel programs are compiled using a template meta-programming approach, where algorithmic skeletons are expanded and optimised for some target architectures, and a performance model is used to guide the program transformations [BCD+97].

Quaff

Quaff is a skeleton-based programming library for C++, implemented using C++ template meta-programming techniques for code generation and optimisation, in a similar way to [BDO+95] and [HBL03]. A formal semantics of skeletons as process networks in CSP was defined by Falcou and Sérot [FS08], and used to generate C+MPI code from Quaff programs, via a CSP process network.

SkeTo

SkeTo [MIEH06] (Skeletons in Tokyo) is a constructive parallel programming library, based on the theory of *constructive algorithmics* [Bir89]. Constructive algorithmics is a field in which some calculus for program transformation is used to derive efficient programs from simple specifications. An example of this is the Bird theory of lists [Bir87]. Different skeletons for manipulating variable-length lists [TI09], trees [MHT06a] and matrices [KI08] have been developed in the context of the SkeTo library. Accurate cost models for efficient implementations of tree skeletons on distributed memory systems were developed in the context of the SkeTo library [Mat17].

The work developed in the context of this library is highly related to the work developed in this thesis, with a few important differences. First, the work in this thesis provides a formalism for both describing the operational semantics and the functional behaviour of algorithmic skeletons, while in the SkeTo library, for each newly defined skeleton, an implementation and cost models must be defined. Skeletons in SkeTo can achieve, therefore, better performance, while the skeletons defined in this thesis are coupled with cost models *for free*, i.e. they are predictable. Secondly, the particular model for describing the operational semantics of algorithmic skeletons in this thesis is not fixed and, therefore, Structured Arrows could be used as a front-end for writing structured parallel programs that use SkeTo skeletons.

Skeleton Parallelising Compilers

SkelML [Bra94] is a parallelising compiler for Standard ML [Mil97] that focuses on finding calls to higher-order functions in ML programs. The parallelising compiler would replace particular instances of HOFs with their parallel versions to achieve speedups. The usage of cost models is discussed. A more general presentation of this is the work of Michaelson et al. [MIK97] and further developed by Scaife et al. [SHMB05]. In this work, the suggested approach consists not only on finding instances of HoFs, but also on using a proved version of HoF transformations for parallelising programs written in terms of explicit instances of maps and folds.

Refactoring Pattern Language

Janjic *et al.* [JBM+16] define a high-level DSL, the Refactoring Pattern Language (RPL), which is aimed at representing the parallel structure of a program, and capturing its execution time. This DSL is a powerful tool, since it allows suitable parallelisations to be found for a given program, and then to apply them to a real C++ program. There are a number of differences between the approach that is described here and RPL. First, our type-based approach does not need to realise parallelisations as refactoring rules: parallel structures are tied to programs in a systematic way by each syntactic construct. The advantage of our approach is that we can use type information to automatically generate parallel code at compile-time.

The corresponding disadvantage is lack of flexibility: the RPL approach can be combined with a number of refactorings that can take into account the user input in a more interactive way. The second important difference is that we use hylomorphisms as a unifying construct. This enables us to use the rich theory of hylomorphisms for parallelism. Moreover, we base our approach on a decision procedure that is derived from an equational theory that is both sound and complete w.r.t. the rules of the underlying equational theory, and we use a standard type unification algorithm to instantiate parallel structures from sequential code. Finally, our parallel structures and cost models are not built-in, but are derived in a systematic way from an underlying cost model and operational semantics.

Glasgow's *Adaptive Skeleton* Framework

The *Adaptive Skeleton* framework [MMT16], developed at the University of Glasgow, provides a number of *adaptive* skeletons. Adaptive skeletons are algorithmic skeleton implementations that perform dynamic scheduling decisions and transformations to the parallel structure of a program, to adapt to changing inputs. Adaptive skeletons aim to provide *portable performance* across different architectures, thanks to their novel dynamic scheduling and transformations. In that sense, the AS framework is very general. However, they lack flexibility, since each new skeleton must be implemented to carefully fit their approach.

Lift

The Lift framework [SFLD15, SRD17] provides a mechanism to generate high-performance OpenCL code from a high-level specification by applying a set of rewrite rules, and using Monte-Carlo search to traverse the corresponding search space to find an implementation. The set of rules combine well-known rewriting rules for map/reduce programs, together with GPU-specific rewritings aimed at generating efficient low-level code. Since their approach applies specifically to GPU programming, we could benefit from exploiting Steuwer *et al.*'s work in GPU-specific rewriting rules and skeletons to extend the StA framework.

PType

The PType [XKCH03] is a type-based skeletal programming framework, where the type system provides a set of rules to detect the parallelisability of functions. PType requires an implementation of the different skeletons that can be used to parallelise different patterns of recursion. PType [XKCH03] is in a sense more general than StA, since they are able to parallelise more expressive recursive functions, such as functions defined using mutual recursion. The parallel implementations are left abstract and chosen in compilation time. However, their approach lacks flexibility, since skeletons need to be implemented using a low-level technique, and it is difficult to add new implementations for new parallel structures and support more architectures.

Structured Arrows

The StA framework that we developed for this thesis is a type-based framework that fully separates the specification of the functionality of a program, and its possible parallelisations. StA comprises two different languages: one for specifying the functional behaviour of a program, and another for specifying the parallel behaviour of new parallel structures. The language for specifying the functional behaviour is a purely functional language that is a subset of Haskell, which does not allow mutual or nested recursion. StA builds on the connection between algorithmic skeletons and patterns of recursion. The parallel implementations can be selected and fine-tuned by the programmer adding type annotations to the functions that need to be parallelised. The StA framework provides also a lower-level language for specifying the operational behaviour of new parallel structures. Cost models can be systematically derived from the operational specifications. However, the programmer must manually prove the correspondence between any newly defined parallel structures and the corresponding pattern of recursion.

2.4.2 Cost Models for Algorithmic Skeletons

The usage of cost models for statically predicting the performance of parallel programs is very powerful. This allows programmers to *predict* the run-time performance of their programs on different inputs without having to run

them. Since profiling a parallel program can be time consuming and hard, a static prediction of the run-time behaviour of a program can ease the task of developing parallel software, since the least efficient parallelisations can be discarded without needing to run them. A number of models of parallel programming have been developed, aimed at predicting the run-time performance of the parallel programs defined within their particular model. A brief overview of representative cost models, both for structured and unstructured parallelism is discussed below.

Parallel Abstract Machines and other Models of Parallelism

PRAM The Parallel Random Access Machine is an abstract machine designed to model the performance of parallel algorithms [FW78]. The PRAM model focused mainly on MIMD machines, but its application was studied to other architectures, such as SIMD. The PRAM model makes a number of simplifications and assumptions that have an important effect in the real run-time of parallel applications, such as an unbounded number of processors in the machine, or not considering resource contention.

NESL Blelloch and Greiner have demonstrated provable time and space bounds for nested data parallel computations in NESL [Ble95]. NESL is a parallel programming language that heavily influenced other more recent approaches to data parallelism, such as Data Parallel Haskell [CLPJ+07]. Despite providing provable time and space cost models, NESL, is specifically designed for nested data parallelism, and is not directly applicable to task parallelism. Moreover, nested data parallelism is handled by applying a flattening transformation that converts it to flat data parallelism. This may lead to an increase of the space complexity of algorithms, and is a potential source of inefficiency [SBHG08].

BSP The Bulk Synchronous Parallel abstract computer serves a similar purpose to the PRAM model [Val90, Val89]. Contrary to the PRAM model, in the BSP model the communication and synchronisation between components of a parallel application is taken into account. In the BSP model, computation takes place as a sequence of *supersteps*. A superstep consists

on a sequence of three different phases. First, a number of independent parallel processes perform some local computation in parallel, by using local data. Then, these processes communicate to each other, sharing their local data with the other processes. Finally, the processes wait in a synchronisation point to ensure that all communication has taken place, before continuing. BSP algorithms have, therefore, a very regular computation and communication structure that can be exploited to predict the run-time performance of parallel programs. The use of the BSP model for determining worst-case execution times of parallel programs has been explored [GGL12]. Although the BSP model has been explored in the context of algorithmic skeletons [JL11a], this work is limited to data-parallel skeletons. Extending it to task-parallel skeletons is not straightforward. For example, defining what is a superstep in a stream-based parallel program implemented in terms of farms and pipelines is not clear, since local computation and communication happens in an interleaved way.

Synchronous Data Flow Languages Synchronous data flow languages provide constructs that are highly related to streaming algorithmic skeletons. Examples of this are Lucid [AW77] and LUSTRE [CPHP87]. These languages define a set of constructs that operate on streams, and have a natural notion of clocks [CP96, CDE+06], that has been formalised as a *clock calculus* [CP95]. Most recently, two different teams have used a specialisation of the clock calculus to infer rates in data flow programs: Rate Types [BL14] to infer maximum throughput; and Data Flow Fusion [LCKR13], infer costs to guide a stream fusion procedure.

Cost Models for Algorithmic Skeletons

Algorithmic skeletons have a clear computation and communication structure. This clear structure makes them predictable, when compared to unstructured parallel programming approaches.

A notable example of cost models for algorithmic skeletons was developed by Skillicorn et al. [SC95]. They use a cost calculus for optimising parallel implementations developed in Bird's theory of lists [Ski93b]. The most

important advantage of this work is that it is applicable to any skeleton-based approach that is derived from Bird's theory of lists.

Many alternative cost models for algorithmic skeleton frameworks have been studied [Ham99]. The usefulness of cost modelling has been convincingly demonstrated more recently by using both static analysis [HC02] and dynamic profiling [LL10] More recently, Matsuzaki [Mat17] has demonstrated this by not only defining efficient tree skeletons on distributed memory systems, but also deriving accurate cost models for them.

Armih et al. [AMT11] develop cost models for algorithmic skeletons on heterogeneous architectures that take into account important low-level information about the underlying architecture, such as cache sizes.

Cost models have been used for deriving efficient parallel programs from specifications [Ski93a], or guiding software refactorings that improve the program's efficiency [BDH+13].

All the current approaches in cost models for algorithmic skeletons are either derived manually, obtained through approximation and measurement, or ignore important low-level information about synchronisation and communication. None of the state-of-the-art approaches derive accurate cost models from an operational semantics, while also taking into account low-level communication and synchronisation details. Outside the algorithmic skeleton community, the BSP model is the closest to achieving this. However, the BSP model requires programmers to focus on low-level implementation details. The operational semantics of algorithmic skeletons developed in the context of the StA framework tackle this problem by providing a mechanism for both: (a) deriving systematically cost models from the operational semantics of parallel structures; and (b) taking low-level communication and synchronisation into account for the cost models of the derived algorithmic skeletons. However, some of the work on cost modelling for algorithmic skeletons is orthogonal to the one defined in this thesis, and can be used to further refine the approach described in this thesis, e.g. taking cache sizes into account [AMT11].

2.4.3 Algorithmic Skeletons and Recursion Patterns

The expressive power of hylomorphisms for parallel programming was first explored by Fischer and Gorlatch [FG02], who showed that a programming language based on catamorphisms and anamorphisms is *Turing-universal*. The idea of using hylomorphisms for parallel programming also appears in Morihata's work [Mor13]. Morihata explores a theory for developing parallelisation theorems based on the *third homomorphism theorem* and *shortcut fusion*, and generalises it to hylomorphisms. The third homomorphism theorem, list homomorphisms, and the Bird-Meertens Formalism are amongst the many techniques that have been explored [GL95, HIT97, HTC98, KC98, MHT06b, Mis94, MM10, Rei93, Ski93b, Ski91]. The third homomorphism theorem states that if a function can be written as both a left fold and a right fold, then it can also be evaluated in a divide-and-conquer manner [Gib96b]. This theorem has been widely used for parallelism [CM11, GG99, Gib96a, Gor99, LHM11, Mor13, MMM+07]. Steuwer et al. [SFLD15, SRD17] generate high-performance OpenCL code from a high-level specification by applying a set of rewrite rules, and using Monte-Carlo search to traverse the corresponding search space to find an implementation.

The integration between cost calculus and code rewriting using equational reasoning has been explored by Skillicorn and Cai [SC95]. In this work, a cost calculus for a skeletal language derived from Bird-Meertens theory of lists is derived, and used to guide code rewritings so that the cost of the resulting parallel programs is reduced. This idea is also explored by the *FAN* framework [AGLP01]. In this framework, the functional behaviour of a number of algorithmic skeletons is used to derive a number of semantically correct rewriting rules for program transformation which, when coupled with cost models can be used to guide the selection of a combination of algorithmic skeletons that minimise the cost of a given parallel program. Finally, Janjic *et al.* [JBM+16] define a high-level DSL, the Refactoring Pattern Language (RPL), that aims to represent the parallel structure of a program, and capture its execution time. This DSL is a powerful tool, since it allows suitable parallelisations to be found for a given program, and then to apply them to a real C++ program.

Current state-of-the-art approaches, however, either rely on a built-in set of algorithmic skeletons, built-in cost models or do not allow the specification of the operational behaviour of parallel structures. There is a need for new general framework that combines both the ability to reason about program transformations, and execution times of parallel programs that take into account low-level communication and synchronisation times.

2.5 Summary

The current approaches to structured parallelism do not provide a general framework for parallel programming that combines the ability to reason about the correctness of program transformations, together with the ability to reason about the run-time performance of parallel programs. The few approaches that do so are either not general, i.e. they are targeted at a specific kind of problems or architectures, or not flexible, i.e. they cannot be easily extended with new parallel structures, or alternative parallel implementations for the supported parallel structures.

The *Structured Arrows (StA)* developed in this thesis is a novel skeletal programming framework aimed at being extensible with new parallel structures that have associated cost models. The structures and cost models are exposed as type abstractions, which enables a program to be implemented *once*, and then parallelised by adding the associated type annotations. The StA framework combines two formalisms: a queue-based operational semantics, and *hylomorphisms*. A parallel structure specified in the queue-based model must be proven equivalent to the corresponding hylomorphism. However, when this is done, a simple type annotation can be used to replace a program structure to use the newly defined structure. The queue-based model is simple and predictable. The next chapter introduces the core part of the StA framework: the type-and-effect system and the notion of *structured arrows*.

Chapter 3

Structured Arrows: A Type System for Parallelism

This chapter contains a formal description of the main part of this thesis: the novel skeletal programming type-based framework, *Structured Arrows (StA)*. StA aims to provide a general framework for parallel programming that allows simultaneously reasoning about:

1. code rewritings that introduce/change parallelism to a program; and
2. run-time performance of alternative parallelisations of a program.

This chapter presents the core language, a point-free, purely functional language with hylomorphisms; and its typing system, which is the one that guides the introduction of parallelism to sequential functions. A prototype implementation of this framework, and extensions, is available in `https://bitbucket.org/david_castro/skel`.

3.1 Overview

The main idea behind this framework is to separate the specification of the *functionality* of a program from the possible parallelisations. At the value level, the language Hylo of *structured expressions* is used to specify the functionality of programs. There is no parallelism at this level. At the type-level, a type-and-effect system is provided that annotates function types,

arrows, with arbitrary program structures. During type-checking, function definitions are annotated with the underlying structure of the program, and checked against the structure annotations. This is illustrated by Figure 3.1 on page 71. At the value level, structured expressions in Hylo describe the functionality of a program as a composition of hylomorphisms. At the type level, the type $A \overset{\sigma}{\mapsto} B$ represents the type of a function from A to B ($A \to B$), that can be parallelised according to the structure σ. This structure σ is an abstraction that captures the composition of hylomorphisms and algorithm skeletons that can be used to parallelise e. Structures σ do not need to be fully specified, and can contain *holes* that can be automatically instantiated by minimising the *cost* of the resulting expression.

To illustrate this, consider the function *image merge* below:

```
imgMerge : List (Img × Img) → List Img
imgMerge = map (merge ∘ mark)
```

This function takes a list of pairs of images as input, and returns a list of images that result from merging the pairs of input images. The images are merged in two stages. First, a function `mark` marks the pixels from both images that need to be merged, according to some dynamic condition on the images. Then, the function `merge` computes the resulting image from the marked pixels. The body of `imgMerge` is a structured expression in the language Hylo. The type annotation is a regular function type List(`Img` × `Img`) → List `Img`.

In the StA framework, parallelising this function can be done by providing a suitable type annotation. StA introduces a new form of *structure-annotated arrows*, or *structured arrows*. A structured arrow is a function type $A \overset{\sigma}{\mapsto} B$, that is annotated with a target parallel structure σ. For example, a programmer might specify that `imgMerge` can be parallelised using a parallel pipeline, by instantiating $\sigma = \_ \parallel \_$. This structure uses $\parallel$ to represent parallel pipelines, and underscore to represent *holes*.

$$\texttt{imgMerge} : \mathsf{List}\ (\texttt{Img} \times \texttt{Img}) \overset{\_\parallel\_}{\longmapsto} \mathsf{List}\ \texttt{Img}$$

A code-generation stage would need to instantiate these holes with sub-expressions from the body of `imgMerge`, for example:

```
fun mark ‖ fun mark
```

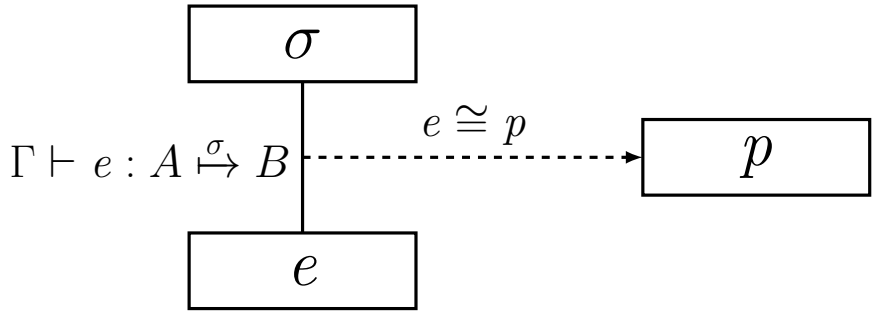


Figure 3.1: *Structured Arrows*: well-typed structured expressions (e) against a target (parallel) structure (σ) can be rewritten turned into functionally equivalent parallel programs (p) according to the specified structure.

The type-checking algorithm determines whether it is possible to instantiate the structure with sub-expressions from `imgMerge`, and then the code-generation selects an arbitrary instantiation from the set of possibilities.

In the StA framework, the structures σ can be parameterised with run-time information that can be used to statically predict the speedups of alternative parallelisations. The type-checking algorithm can use *cost models* to predict statically how to instantiate any holes so that the cost of the overall structure is minimised. In the StA framework, cost models can be interpreted as functions from structures to run-time predictions.

$$\texttt{cost} : \Sigma \to \texttt{Integer}$$

Given a suitable cost model, the type of `imgMerge` would change to:

$$\texttt{imgMerge} : \mathsf{List}\ (\texttt{Img} \times \texttt{Img}) \xmapsto{\min\ \texttt{cost}\ (\_\|\_)} \mathsf{List}\ \texttt{Img}$$

The novel contributions of this chapter are:

- *A denotational semantics for common algorithmic skeletons in terms of hylomorphisms* (Section 3.2).

- *Structure-annotated Arrows (StA).* A novel type and effect system that annotates function types of a point-free programming language with the underlying *program structure*. This structure can be used to reason simultaneously about equivalent, alternative implementations and their cost on different architectures (Section 3.3).

- A *reforestation* procedure for deciding semantic equivalences between alternative parallel programs. This decision procedure is based on reintroducing intermediate data structures, "reforestation", rather than the more common approach of eliminating them for efficiency reasons, "deforestation". This enables the type system to introduce parallelism in a semi-automated and sound way (Section 3.4)

3.2 Structured Parallel Programs

Before introducing the StA framework, we present a skeletal Domain Specific Language, similar to that in [JBM+16]. The denotational semantics of this language is explained in terms of hylomorphisms, and this will motivate the definition of the *structured expressions* of the language Hylo, in the next Section 3.3.

3.2.1 Structured Parallel Processes

We define a language P of structured parallel processes, built by composing skeletons over atomic operations.

$$p \in P \ ::= \ \mathsf{fun}_T\ f \mid p_1 \parallel p_2 \mid \mathsf{dc}_{n,T,F}\ f\ g \mid \mathsf{farm}\ n\ p \mid \mathsf{fb}\ p$$

The $\mathsf{fun}_T\ f$ construct *lifts* an atomic function to a streaming operation on a collection T. The arguments of the dc skeleton are: the number of *levels* of the divide-and-conquer, n; the collection T on which the dc skeleton works; and the functor F that describes the divide-and-conquer call tree.

Denotational Semantics. The denotational semantics is split into two parts: $\mathcal{S}[\![\cdot]\!]$ describes the base semantics, and $[\![\cdot]\!]$ lifts this to a *streaming* form. We use a global environment for atomic function types, ρ, and the corresponding global environment of functions, $\hat{\rho}$:

$$\rho \ = \ \{f : A \to B,\ \ldots\} \qquad \hat{\rho} \ = \ \{[\![f]\!] \in [\![A \to B]\!],\ \ldots\}$$

$$
\begin{array}{lcl}
\mathcal{S}[\![p \ : \ T\ A \to T\ B]\!] & : & [\![A \to B]\!] \\
\mathcal{S}[\![\mathsf{fun}\ f]\!] & = & \hat{\rho}(f) \\
\mathcal{S}[\![p_1 \ \| \ p_2]\!] & = & \mathcal{S}[\![p_2]\!] \circ \mathcal{S}[\![p_1]\!] \\
\mathcal{S}[\![\mathsf{farm}\ n\ p]\!] & = & \mathcal{S}[\![p]\!] \\
\mathcal{S}[\![\mathsf{fb}\ p]\!] & = & iter\ \mathcal{S}[\![p]\!] \\
\mathcal{S}[\![\mathsf{dc}_{n,T,F}\ f\ g]\!] & = & cata_F\ (\hat{\rho}(f)) \circ ana_F\ (\hat{\rho}(g))
\end{array}
$$

$$
\begin{array}{lcl}
[\![p \ : \ T\ A \to T\ B]\!] & : & [\![T\ A \to T\ B]\!] \\
[\![p]\!] & = & map_T\ \mathcal{S}[\![p]\!]
\end{array}
$$

An atomic function, f, is applied to all the elements of a collection of data. A parallel pipeline, $p_1 \parallel p_2$, is the composition of two parallel processes, p_1 and p_2. A task farm, $\mathsf{farm}\ n\ p$, replicates a parallel process, p, so has the same denotational semantics as p. A feedback skeleton, $\mathsf{fb}\ p$, applies the computation p iteratively, i.e. *trampolined*, to the elements in the input collection. Its semantics is given in terms of the function *iter*.

$$
\begin{array}{lcl}
iter & : & (A \to A + B) \to A \to B \\
iter\ f & = & \mathtt{Y}\ (\lambda\ g.(g \triangledown id) \circ f)
\end{array}
$$

Finally, a `dc` is equivalent to *folding*, using f, the tree-like structure that results from *unfolding* the input using g. It is defined to be the composition of a *catamorphism* with an *anamorphism*.

$$
\begin{array}{lcl}
cata_F & : & (F\ A \to A) \to \mu F \to A \\
cata_F\ f & = & f \circ F\ (cata_F\ f) \circ out_F
\end{array}
$$

$$
\begin{array}{lcl}
ana_F & : & (A \to F\ A) \to A \to \mu F \\
ana_F\ g & = & in_F \circ F\ (ana_F\ g) \circ g
\end{array}
$$

3.2.1 Example | List catamorphism. Let L be the base bifunctor of a polymorphic list. We define the function f to be:

$$
\begin{array}{lcl}
f & : & L\ \mathbb{N}\ \mathbb{N} \to \mathbb{N} \\
f\ (inj_1\ ()) & = & 0 \\
f\ (inj_2\ (x, n)) & = & \mathtt{add}\ x\ n
\end{array}
$$

$$\frac{\rho(f) \;=\; A \to B}{f \;:\; A \to B} \qquad \frac{\begin{array}{c} e_2 \;:\; B \to C \\ e_1 \;:\; A \to B \end{array}}{e_2 \circ e_1 \;:\; A \to C} \qquad \frac{\begin{array}{c} e_1 \;:\; F\ B \to B \\ e_2 \;:\; A \to F\ A \end{array}}{\mathtt{hylo}_F\ e_1\ e_2 \;:\; A \to B}$$

$$\frac{p \;:\; T\ A \to T\ B}{\mathsf{par}_T\ p \;:\; T\ A \to T\ B}$$

Figure 3.2: Simple types for Structured Expressions, E.

$$\frac{s \;:\; A \to B}{\mathsf{fun}\ s \;:\; T\ A \to T\ B} \qquad \frac{n \;:\; \mathbb{N} \qquad p \;:\; T\ A \to T\ B}{\mathsf{farm}\ n\ p \;:\; T\ A \to T\ B}$$

$$\frac{p \;:\; T\ A \to T\ (A + B)}{\mathsf{fb}\ \ p \;:\; T\ A \to T\ B}$$

$$\frac{s_1 \;:\; F\ B \to B \qquad s_2 \;:\; A \to F\ A}{\mathsf{dc}_{n,F}\ \ s_1\ s_2 \;:\; T\ A \to T\ B}$$

$$\frac{p_1 \;:\; T\ A \to T\ B \qquad p_2 \;:\; T\ B \to T\ C}{p_1 \parallel p_2 \;:\; T\ A \to T\ C}$$

Figure 3.3: Simple types for Structured Parallel Processes, P.

Given an input list $[x_1, x_2, \ldots, x_n]$, the catamorphism $cata_{L\,\mathbb{N}}\ f$ applied to this input list returns the sum of the x_i:

$$cata_{L\,\mathbb{N}} f\,[x_1, x_2, \ldots, x_n] \;=\; \mathtt{add}\ x_1\ (\mathtt{add}\ x_2\ (\cdots\ (\mathtt{add}\ x_n\ 0))).$$

3.2.2 Example | List anamorphism. We define a function g that returns $()$ if the input n is zero, and $(n, n-1)$ otherwise.

$$\begin{array}{lcl} g & : & \mathbb{N} \to L\ \mathbb{N}\ \mathbb{N} \\ g\ n & = & \text{if } n = 0 \text{ then } inj_1\ () \text{ else } inj_2\ (n, n-1) \end{array}$$

The anamorphism $ana_{L\,\mathbb{N}}\ g$ applied to n returns a list of numbers descending from n to 1: $ana_{L\,\mathbb{N}}\ g\ n \;=\; [n, n-1, \ldots, 2, 1]$.

Figure 3.4 shows how catamorphisms and anamorphisms work on binary trees. In the anamorphism, we start with an input value, and apply the operation f recursively until the entire data structure is *unfolded*. In the

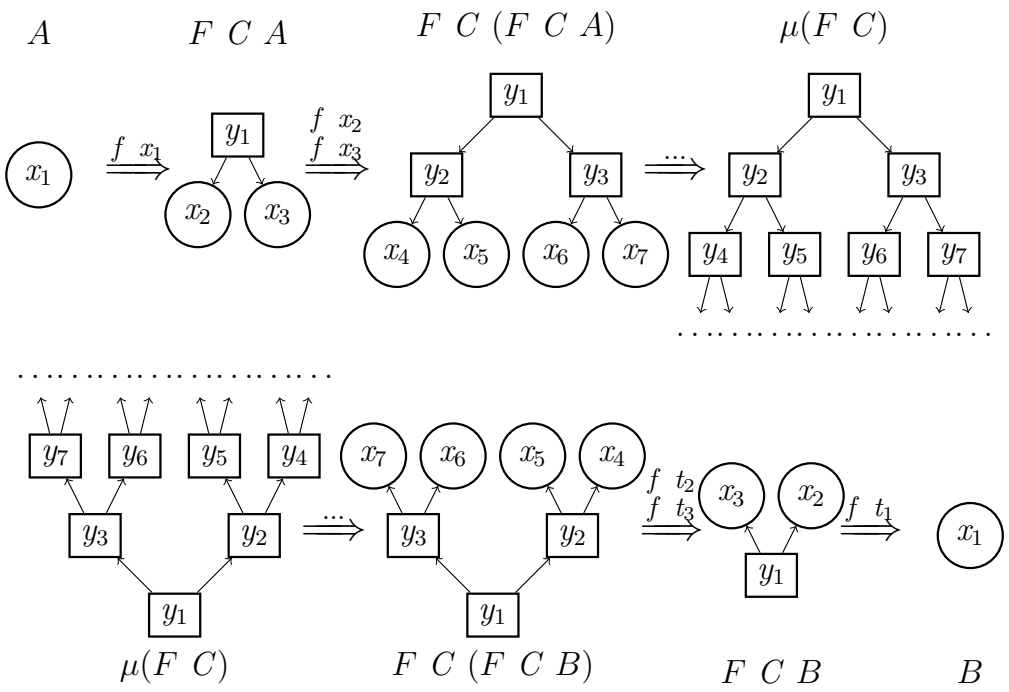


Figure 3.4: Binary Tree Anamorphism (above) and Catamorphism (below).

catamorphism, the operation f is applied recursively until the entire structure is *folded* into a single value. The operation map_T can be defined as a special case of a catamorphism or anamorphism. Given a bifunctor G, a type $T\ A = \mu(G\ A)$ is a polymorphic data type that is also a functor [Gib02a]. For all $f : A \to B$, the function $map_T\ f$ is the morphism $T\ f$, defined as:

$$\begin{aligned} map_T\ f &= cata_{G\,A}(in_{G\,B} \circ G\ f\ id) \\ &= ana_{G\,B}(G\ f\ id \circ out_{G\,A}) \end{aligned}$$

For uniformity, we will represent all map_T as catamorphisms.

3.2.3 Example | map_{List}. Given a function $f : A \to B$, $map_{\mathsf{List}}\ f$ applies f to all the elements of the input list:

$$map_{\mathsf{List}}\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

3.2.2 Hylomorphisms

Recall from Chapter 2 that *Hylomorphisms* are a well known recursion pattern [MFP91], that generalise the common notion of a divide-and-conquer

algorithm. Intuitively, $hylo_F\ f\ g$ is a recursive algorithm whose recursive call tree can be represented by μF, where g describes how the algorithm divides the input problem into sub-problems, and f describes how the results are combined.

$$\begin{array}{lcl} hylo_F & : & (F\ B \to B) \to (A \to F\ A) \to A \to B \\ hylo_F\ f\ g & = & f \circ F\ (hylo_F\ f\ g) \circ g \end{array}$$

Since $out_F \circ in_F = id$, we can show that

$$hylo_F\ f\ g = cata_F\ f \circ ana_F\ g.$$

This is done by equational reasoning:

$$\begin{array}{l} \quad cata_F\ f\ \circ\ ana_F\ g \\ = f \circ F\ (cata_F\ f) \circ out_F \circ in_F \circ F\ (ana_F\ g) \circ g \\ = f \circ F\ (cata_F\ f) \circ \mathtt{id} \circ F\ (ana_F\ g) \circ g \\ = f \circ F\ (cata_F\ f) \circ F\ (ana_F\ g) \circ g \\ = f \circ F\ (cata_F\ f \circ ana_F\ g) \circ g \end{array}$$

As was shown in Chapter 2, catamorphisms, anamorphisms and *map* are special cases of hylomorphisms. This can be shown using the facts that $cata_F\ in_F = id$ and $ana_F\ out_F = id$. Using bifunctors again, we can conclude that map_T, ana_F and $cata_F$ can all be defined in terms of this single construct. Given a bifunctor F,

$$\begin{array}{lcl} T\ A & = & \mu(F\ A) \\ map_T\ f & = & hylo_{F\,A}\ (in_{F\,B} \circ (F\ f\ id))\ out_{F\,A}, \\ & & \quad \text{where } A = dom(f) \text{ and } B = codom(f) \\ cata_F\ f & = & hylo_F\ f\ out_F \\ ana_F\ f & = & hylo_F\ in_F\ f \end{array}$$

Since the semantics of parallel constructs was defined in terms of map_F, $cata_F$, ana_F and *iter*, we also need to define *iter* in terms of *hylo* to be able to boil down all of our parallel constructs to hylomorphisms. Although the fixpoint combinator `Y` can be defined as a hylomorphism, we take a different approach. Observe that we can unfold the definition of *iter* as follows:

$$iter\ f\ =\ \mathtt{Y}\ (\lambda\ g.(g \triangledown id) \circ f)\ = (iter\ f \triangledown id) \circ f$$

Note that if $f, g \ : \ A + B \rightarrow C$, the function $f \triangledown g \ : \ A + B \rightarrow C$ can be written as the composition of $id \triangledown id \ : \ C + C \rightarrow C$ and $f \ + \ g \ : \ A + B \rightarrow C + C$. We use this to rewrite *iter* as follows:

$$iter\ f \ = \ (iter\ f \triangledown id) \circ f \ = \ (id \triangledown id) \circ (iter\ f + id) \circ f$$

If $f : A \rightarrow A + B$, we define the functor $(+B)$, with the $(+\ B)\ f \ = \ f + id$, which preserves identities and composition. Since $iter\ f = (id \triangledown id) \circ (+B)\ (iter\ f) \circ f$, then:

$$iter\ f \ = \ hylo_{(+B)}\ (id \triangledown id)\ f$$

3.2.3 Structured Expressions

We have now seen that the denotational semantics of all our parallel constructs can be given in terms of hylomorphisms. This semantic correspondence is not unexpected since it has been used to describe the formal foundations of data-parallel algorithmic skeletons [RG03]. We take this correspondence one step further by using hylomorphisms as a unifying structure, and by then exploiting the reasoning power provided by the fundamental laws of hylomorphisms. In order to define our type-based approach, we will first define a new language, E, that combines two levels, *Structured Expressions* (S), that enable us to describe a program as a composition of hylomorphisms; and *Structured Parallel Processes* (P), that build on S using nested algorithmic skeletons. A program in E is then either a structured expression $s \in S$ or a parallel program $\mathsf{par}_T\ p$, where $p \in P$. Our revised syntax is shown below. Note that since a $p \in P$ can only appear under a par_T construct, we no longer need to annotate each fun and $\mathtt{dc}$ with the collection T of tasks.

$$\begin{array}{lcl}
e \in E & ::= & s \ \mid \ \mathsf{par}_T\ p \\
s \in S & ::= & f \ \mid \ e_1 \circ e_2 \ \mid \ \mathtt{hylo}_F\ e_1\ e_2 \\
p \in P & ::= & \mathsf{fun}\ s \mid p_1 \parallel p_2 \mid \mathsf{dc}_{n,F}\ s_1\ s_2 \mid \mathsf{farm}\ n\ p \mid \mathsf{fb}\ p
\end{array}$$

The denotational semantics of P only changes in the rules that mention e, and by providing a semantics for par_T:

$$\begin{array}{lcl} [\![\mathsf{par}_T\ p]\!] & = & map_T\ \mathcal{S}[\![p]\!] \\ \ldots & & \\ \mathcal{S}[\![\mathsf{fun}\ e]\!] & = & [\![e]\!] \\ \mathcal{S}[\![\mathsf{dc}_{n,F}\ e_1\ e_2]\!] & = & hylo_F\ [\![e_2]\!]\ [\![e_1]\!] \\ \ldots & & \end{array}$$

The corresponding typing rules are entirely standard (Figures 3.2–3.3). Finally, it is convenient to define the "parallelism erasure of S", $\overline{S}$. Intuitively, $\overline{S}$ contains no nested parallelism: for all $s \in S$, $s \in \overline{S}$ if and only if s contains no occurrences of the par_T construct. In other words, $s \in \overline{S}$ if it is defined only in terms of composition, atomic functions and hylomorphisms:

$$s \in \overline{S} \ ::= \ f \quad | \quad s_1 \circ s_2 \quad | \quad \mathtt{hylo}_F\ s_1\ s_2$$

The structure-annotated type system given in Section 3.3 below describes how to introduce parallelism to an $s \in \overline{S}$ in a sound way.

Soundness and Completeness

It is straightforward to show that the type system from Figs. 3.2– 3.3 is both sound and complete *wrt* our denotational semantics. Our soundness property is: $\forall e \in E;\ A, B \in \mathtt{Type},\ \vdash e : A \to B \implies ([\![e]\!] \in [\![A \to B]\!])$. The proof is by structural induction over the terms in E, using the definitions of $\vdash e : T$ from Figs. 3.2–3.3 and $[\![.]\!]$ above. The corresponding completeness property is: $\forall e \in E;\ A, B \in \mathtt{Type},\ ([\![e]\!] \in [\![A \to B]\!]) \implies \vdash e : A \to B$. The proof is also by structural induction over the terms in E, using the definitions of $\vdash e : A \to B$ from Figs. 3.2–3.3 and $[\![.]\!]$ above.

3.3 A Type System for Introducing Parallelism

In this section, we present a rigorous way to introduce parallelism without affecting a program's functional behaviour. We annotate top-level program

$$\frac{\rho(f) = A \to B}{f \ : \ A \xrightarrow{\mathrm{A}} B}$$

$$\frac{e_1 \ : \ B \xrightarrow{\sigma_1} C \qquad e_2 \ : \ A \xrightarrow{\sigma_2} B}{e_1 \circ e_2 \ : \ A \xrightarrow{\sigma_1 \circ \sigma_2} C}$$

$$\frac{\begin{array}{c} e_1 \ : \ F \ B \xrightarrow{\sigma_1} B \\ e_2 \ : \ A \xrightarrow{\sigma_2} F \ A \qquad G \ = \ \mathtt{base} \ F \end{array}}{\mathtt{hylo}_F \ e_1 \ e_2 \ : \ A \xrightarrow{\mathrm{HYLO}_G \ \sigma_1 \ \sigma_2} B}$$

$$\frac{p \ : \ T \ A \xrightarrow{\sigma} T \ B \qquad F \ = \ \mathtt{base} \ T}{\mathsf{par}_T \ p \ : \ T \ A \xrightarrow{\mathrm{PAR}_F \ \sigma} T \ B}$$

(a) Structure-Annotated Type System for E.

$$\frac{s \ : \ A \xrightarrow{\sigma} B}{\mathsf{fun} \ s \ : \ T \ A \xrightarrow{\mathrm{FUN} \ \sigma} T \ B}$$

$$\frac{s_1 \ : \ F \ B \xrightarrow{\sigma_1} B \qquad s_2 \ : \ A \xrightarrow{\sigma_2} F \ A \qquad G \ = \ \mathtt{base} \ F}{\mathsf{dc}_{n,F} \ \ s_1 \ s_2 \ : \ T \ A \xrightarrow{\mathrm{DC}_{n,G} \ \sigma_1 \ \sigma_2} T \ B}$$

$$\frac{n \ : \ \mathbb{N} \qquad p \ : \ T \ A \xrightarrow{\sigma} T \ B}{\mathsf{farm} \ n \ p \ : \ T \ A \xrightarrow{\mathrm{FARM}_n \ \sigma} T \ B}$$

$$\frac{p_1 \ : \ T \ A \xrightarrow{\sigma_1} T \ B \qquad p_2 \ : \ T \ B \xrightarrow{\sigma_2} T \ C}{p_1 \parallel p_2 \ : \ T \ A \xrightarrow{\sigma_1 \ \parallel \ \sigma_2} T \ C}$$

$$\frac{p \ : \ T \ A \xrightarrow{\sigma} T \ (A + B)}{\mathsf{fb} \ \ p \ : \ T \ A \xrightarrow{\mathrm{FB} \ \sigma} T \ B}$$

(b) Structure-Annotated Type System for P.

Figure 3.5: Structured Arrows

types with an abstraction of the *structure* of the program, $\sigma \in \Sigma$. We define the associated type system together with mechanisms for reasoning about these programs using this structure. Intuitively, Σ is a "pruned" version of E that retains information about *how* the computation is performed, while removing as many details as possible about *what* is being computed.

3.3.1 Definition | Families of equivalent programs. *We say that an $e \in E$ is in the family of programs that are functionally equivalent to $s \in \overline{S}$, $e \in E_s$,* if and only if *$e \mathcal{E} s$, for the relation $\mathcal{E}$ that is defined later in this section.*

Let $=_{\text{ext}}$ denote extensional equality: $f =_{\text{ext}} g \Leftrightarrow \forall x, f\ x = g\ x$. Since this is not decidable, we use instead a decidable relation $\mathcal{E}$ which implies extensional equality. Each E_s is a family of programs indexed by their structure, i.e. for each family E_s, there is a function $\phi_s : \Sigma \to E_s$ that returns an $e \in E_s$ with the desired structure. Note that not all structures $\sigma \in \Sigma$ are indices of a family E_s, so ϕ_s is a partial function. Given a structure $\sigma \in \Sigma$ and a $s \in \overline{S}$, we use a superscript, s^σ, as notation for $\phi_s(\sigma)$. We define the structure Σ and the relation $\mathcal{E}$ later in this section, and the function ϕ_s in Section 3.4.

3.3.2 Definition | Structure-annotated arrows. *Given a structure $\sigma \in \Sigma$, and an $s \in \overline{S}$ with type $A \to B$, we say that s has type $A \overset{\sigma}{\mapsto} B$, if s is equivalent to a parallel program with structure σ.*

By typechecking $s : A \overset{\sigma}{\mapsto} B$, the type system guarantees that there is an equivalent program with structure σ, $s^\sigma \in E_s$. That is, the structured expression s typechecks *if and only if* σ is an index of the family E_s. The definition of ϕ_s is actually an algorithm for deriving a parallel program from a sequential program and a type-level structure, i.e. ϕ_s provides a mechanism for selecting a parallel program that is equivalent to s and has structure σ, for well-typed programs. We state this formally in the form of our main soundness and completeness properties later in this section.

3.3.1 The Structure-Annotated Type System

The program structure abstraction, $\sigma \in \Sigma$, is defined below.

$$\begin{array}{rcl} \sigma \in \Sigma & ::= & \sigma_{\mathsf{s}} \mid \textsc{par}_F\ \sigma_{\mathsf{p}} \\ \sigma_{\mathsf{s}} \in \Sigma_{\mathsf{s}} & ::= & \textsc{a} \mid \sigma \circ \sigma \mid \textsc{hylo}_F\ \sigma\ \sigma \\ \sigma_{\mathsf{p}} \in \Sigma_{\mathsf{p}} & ::= & \textsc{fun}\ \sigma_{\mathsf{s}} \mid \textsc{dc}_{n,F}\ \sigma_{\mathsf{s}}\ \sigma_{\mathsf{s}} \\ & \mid & \sigma_{\mathsf{p}} \parallel \sigma_{\mathsf{p}} \mid \textsc{farm}_n\ \sigma_{\mathsf{p}} \mid \textsc{fb}\ \sigma_{\mathsf{p}} \end{array}$$

Figs. 3.5a– 3.5b define the annotated type system that extends our type system from Figs. 3.2– 3.3, and that associates expressions $e \in E$ with structures $\sigma \in \Sigma$. As before, the global environment, ρ, maps primitive functions to their types. The annotated arrow, $e : A \xrightarrow{\sigma} B$, states that e has *exactly* the structure σ. In order to define $A \overset{\sigma}{\mapsto} B$, we need to extend the type system further with a *convertibility relation*.

Convertibility

We extend our type system with a non-structural rule that captures the *convertibility relation*, $\equiv$, for Σ.

$$\frac{e\ :\ A \xrightarrow{\sigma_1} B \qquad \sigma_1 \equiv \sigma_2}{e\ :\ A \overset{\sigma_2}{\mapsto} B}$$

The relation $\equiv$ is defined in terms of the relations $\equiv_{\mathsf{s}} \in \Sigma_{\mathsf{s}} \times \Sigma_{\mathsf{s}}$ and $\equiv_{\mathsf{p}} \in \Sigma_{\mathsf{p}} \times \Sigma_{\mathsf{p}}$, plus a rule that links the Σ_{s} and Σ_{p} levels, PAR-EQUIV.

$$\frac{\sigma_1 \equiv_{\mathsf{s}} \sigma_2}{\sigma_1 \equiv \sigma_2} \qquad \frac{\sigma_1 \equiv_{\mathsf{p}} \sigma_2}{\textsc{par}_F\ \sigma_1 \equiv \textsc{par}_F\ \sigma_2}$$

$$\textsc{par}_F\ (\textsc{fun}\ \sigma) \equiv \textsc{map}_F\ \sigma \qquad (\textsc{par-equiv})$$

The structures MAP and ITER are defined in Section 3.4, and represent the structures of the hylomorphism representations of `map` and `iter`. We define a number of equivalences, starting with $\equiv_{\mathsf{p}}$. A parallel pipeline structure ($\parallel$) is functionally equivalent to a function composition; a task farm FARM can be introduced for any structure; and divide-and-conquer DC and feedback FB can be derived from hylomorphisms.

$$\begin{array}{rcll} \textsc{fun}\ \sigma_1 \parallel \textsc{fun}\ \sigma_2 & \equiv_{\mathsf{p}} & \textsc{fun}\ (\sigma_2 \circ \sigma_1) & (\textsc{pipe-equiv}) \\ \textsc{dc}_{n,F}\ \sigma_1\ \sigma_2 & \equiv_{\mathsf{p}} & \textsc{fun}\ (\textsc{hylo}_F\ \sigma_1\ \sigma_2) & (\textsc{dc-equiv}) \\ \textsc{farm}_n\ \sigma & \equiv_{\mathsf{p}} & \sigma & (\textsc{farm-equiv}) \\ \textsc{fb}(\textsc{fun}\ \sigma) & \equiv_{\mathsf{p}} & \textsc{fun}\ (\textsc{iter}\ \sigma) & (\textsc{fb-equiv}) \end{array}$$

These equivalences, plus reflexivity, symmetry and transitivity, define an equational theory that allows conversion between different parallel forms, as well as conversion between structured expressions and parallel forms, as required by our type system. In these equivalences, we implicitly assume the necessary well-formedness constraints: any structure under a FUN or DC must be in Σ_{s}, and the structure under FARM must be in Σ_{p}. Note that, thanks to the transitivity of $\equiv$, we can use these equivalences to derive interesting properties of our parallel structures. For example, the associativity of parallel pipelines does not need to be defined explicitly, since it can be derived from the associativity of composition. We defer the definition of $\equiv_{\mathsf{s}}$ to Section 3.3.2.

3.3.3 Definition | Convertibility in E. *For all convertibility rules in Σ, there is an equivalent rule in E. We define the equivalence relation $\mathcal{E} \in E \times E$ to be the relation $\equiv$ lifted to E.*

An example that illustrates this is that the PIPE-EQUIV rule corresponds to the rule $(\mathsf{fun}\ s_1 \parallel \mathsf{fun}\ s_2) \quad \mathcal{E}_{\mathsf{p}} \quad (\mathsf{fun}\ (s_2 \circ s_1))$.

3.3.1 Lemma *Semantic equivalence.*

$$\forall e_1, e_2 \in E, \quad e_1\ \mathcal{E}\ e_2 \Rightarrow [\![e_1]\!]\ =_{ext}\ [\![e_2]\!]$$

Proof Straightforward by induction on the structure of the equivalence relation $\mathcal{E}$, using the denotational semantics of P, and the laws of hylomorphisms (Section 3.4). □

As a consequence of Lemma 3.3.1, the $\mathcal{E}$ relation can be used to define the families E_s. Any extension to $\equiv$ and $\mathcal{E}$ may expose more opportunities for parallelisation in E_s. There remains only the definition of the function ϕ_s. We defer this to Section 3.4, together with the decision procedure for $\equiv$ and $\mathcal{E}$.

Soundness and Completeness

Since the annotated type system is an extension of that from Figs 3.2–3.3 and since the convertibility rule only applies to structures, it is trivial to

$$hylo_F\ in_F\ out_F\ =\ id_{\mu F} \qquad \textsc{Hylo-Reflex}$$

$$hylo_F\ (f \circ \eta)\ g\ =\ hylo_G\ f\ (\eta \circ g)$$
$$\Leftarrow\quad \eta\ :\ F \to G \qquad \textsc{Hylo-Shift}$$

$$(hylo_F\ f\ h_1)\ \circ\ (hylo_F\ h_2\ g)$$
$$\Leftarrow\quad h_1 \circ h_2 = id \qquad \textsc{Hylo-Compose}$$

$$f_1 \circ (hylo_F\ g_1\ g_2) \circ f_2\ =\ hylo_F\ g_1'\ g_2' \qquad \textsc{Hylo-Fusion}$$
$$\Leftarrow\quad f_1 \text{ strict } \wedge\ f_1 \circ g_1 = g_1' \circ F\ f_1\ \wedge\ g_2 \circ f_2 = F\ f_2 \circ g_2'$$

$$hylo_F\ f\ g \text{ strict}$$
$$\Leftarrow\quad f, g \text{ strict} \qquad \textsc{Hylo-Strict}$$

Figure 3.6: Hylomorphism Laws

show that the new type system is both sound and complete *wrt* the original system, once structure is removed, since $\forall e \in E, \sigma \in \Sigma,\quad e : A \overset{\sigma}{\mapsto} B \implies e : A \to B$. We therefore omit these proofs. Our main soundness and completeness theorems for convertibility ensure that the type system derives only functionally equivalent parallel structures from structured expressions. The proofs of these properties build on a number of details that are introduced in Section 3.4.

3.3.1 Theorem *Soundness of Conversion.*

$$\forall s \in \overline{S}, \sigma \in \Sigma,\quad s : A \overset{\sigma}{\mapsto} B\ \Rightarrow\ s^\sigma \in E_s$$

Proof Since s^σ is a synonym for $\phi_s(\sigma)$, s^σ is in E_s if ϕ_s is defined for σ. This property follows directly from the definition of $\phi_s(\sigma)$ (Def. 3.4.1) and from Thm 3.4.1 in Section 3.4. □

A consequence of the soundness of the conversion and of Lemma 3.3.1 is that if a structured expression typechecks with type $A \overset{\sigma}{\mapsto} B$, then there always exists a functionally equivalent e whose structure is σ.

3.3.1 Corollary $\forall s \in \overline{S}, \sigma \in \Sigma,\quad s : A \overset{\sigma}{\mapsto} B\ \Rightarrow\ \exists e \in E$ *such that* $e\ :\ A \overset{\sigma}{\to} B$ *and* $[\![e]\!] =_{ext} [\![s]\!]$.

3.3.2 Theorem *Completeness of Conversion.*

$$\forall s \in \overline{S}; \sigma, \sigma' \in \Sigma; \quad s \ : \ A \xrightarrow{\sigma'} B \wedge s^{\sigma} \in E_s \ \Rightarrow \ s : A \overset{\sigma}{\mapsto} B$$

Proof This follows directly from the definition of $\phi_s(\sigma)$ (Def. 3.4.1) and Thm 3.4.1 in Sec. 3.4. □

Remark Note that the rules presented in Figure 3.5 could be implemented as a deep-embedded EDSL in a dependently typed language. This was our original approach in [CH14]. However, the possibility of non-termination makes reasoning about the semantics of these programs quite a hard task. We chose to first prototype these ideas as a standalone DSL, implemented in Haskell, but it would be interesting to implement these ideas as a dependently typed, deeply embedded EDSL. An important challenge would be to implement the typechecking algorithm, and the rewriting system in Section 3.4. In a dependently typed programming, it may be worth exploring whether some of the algebraic properties that are used for proving equivalences can be captured by an algebraic structure in Slama and Brady's hierarchy of provers [SB17].

3.3.2 Functional Equivalence

The proofs of soundness and completeness rely on a decision procedure for $\equiv$, as well as on the definition of the ϕ_s function for the families E_s. The definition of $\equiv$ requires a definition of $\equiv_{\mathsf{s}} \in \Sigma_{\mathsf{s}} \times \Sigma_{\mathsf{s}}$. Our $\equiv_{\mathsf{s}}$ adapts the well known hylomorphism laws (Fig. 3.6) [FM91, Gib02a, MFP91], using restricted instances of those laws. These restrictions serve two purposes: i) we avoid checking *strictness* conditions by ensuring that all the functions we use are strict; ii) because the equivalences that we can capture are very limited if we assume no knowledge of *atomic functions*, due to the side conditions on the rules, we expose extra structure in our programs.

1. We explicitly represent the in_F, out_F and id functions.

2. We explicitly represent the section of a bifunctor F applied to a structured expression s, as $F\ s$ rather than $F\ s\ \mathsf{id}$. This, plus the strictness

assumption, enables us to apply some limited forms of HYLO-SHIFT and HYLO-FUSION.

3. We explicitly represent $\triangle$ and ∇. Although we do not define equivalences for these combinators, we use them to define the ITER structure later.

$$\begin{array}{lcl} s \in S & ::= & f \mid \langle prim \rangle \mid e_1 \circ e_2 \mid \mathtt{hylo}_F\ e_1\ e_2 \\ prim & ::= & \mathsf{in}_F \mid \mathsf{out}_F \mid \mathsf{id} \mid e_1\ \langle op \rangle\ e_2 \mid F\ e \\ op & ::= & \nabla \mid \triangle \\ \sigma_{\mathsf{s}} \in \Sigma_{\mathsf{s}} & ::= & \ldots \mid \text{IN} \mid \text{OUT} \mid \text{ID} \mid \sigma_1\ \langle op \rangle\ \sigma_2 \mid F\ \sigma \end{array}$$

With these structures, we can define the special cases of HYLO_F:

$$\begin{array}{lcl} \text{MAP}_F, \text{CATA}_F, \text{ANA}_F & : & \Sigma \to \Sigma \\ \text{MAP}_F\ \sigma & = & \text{HYLO}_F\ (\text{IN} \circ F\ \sigma)\ \text{OUT} \\ \text{CATA}_F\ \sigma & = & \text{HYLO}_F\ \sigma\ \text{OUT} \\ \text{ANA}_F\ \sigma & = & \text{HYLO}_F\ \text{IN}\ \sigma \\ \text{ITER}\ \sigma & = & \text{HYLO}_{(+)}\ (\text{ID} \nabla \text{ID})\ \sigma \end{array}$$

The typing rules are then extended.

$$\frac{}{\mathsf{id} : A \xrightarrow{\text{ID}} A} \qquad \frac{}{\mathsf{in}_F : F(\mu F) \xrightarrow{\text{IN}} \mu F}$$

$$\frac{}{\mathsf{out}_F : \mu F \xrightarrow{\text{OUT}} F(\mu F)}$$

$$\frac{e\ :\ A \xrightarrow{\sigma} B}{F\ e\ :\ F\ A\ C \xrightarrow{F\ \sigma} F\ B\ C}$$

$$\frac{e_1 : A \xrightarrow{\sigma_1} B \qquad e_2 : A \xrightarrow{\sigma_2} C}{e_1 \triangle e_2 : A \xrightarrow{\sigma_1 \triangle \sigma_2} B \times C}$$

$$\frac{e_1 : A \xrightarrow{\sigma_1} C e_2 : \qquad e_2 : B \xrightarrow{\sigma_2} C}{e_1 \nabla e_2 : A + B \xrightarrow{\sigma_1 \nabla \sigma_2} C}$$

The *convertibility relation* is also extended to include some equivalences that are derived from the hylomorphism laws:

$$
\begin{array}{lcll}
\textsc{id} \circ \sigma & \equiv_{\mathsf{s}} & \sigma & (\textsc{id-left}) \\
\sigma \circ \textsc{id} & \equiv_{\mathsf{s}} & \sigma & (\textsc{id-right}) \\
\textsc{out} \circ \textsc{in} & \equiv_{\mathsf{s}} & \textsc{id} & (\textsc{out-in-id}) \\
\textsc{in} \circ \textsc{out} & \equiv_{\mathsf{s}} & \textsc{id} & (\textsc{in-out-id}) \\
\textsc{hylo}_F\ \textsc{in}\ \textsc{out} & \equiv_{\mathsf{s}} & \textsc{id} & (\textsc{hylo-id}) \\
F\ (\sigma_1 \circ \sigma_2) & \equiv_{\mathsf{s}} & F\ \sigma_1 \circ F\ \sigma_2 & (\textsc{f-comp}) \\
\textsc{hylo}_F\ \sigma_1\ \sigma_2 & \equiv_{\mathsf{s}} & \textsc{cata}_F\ \sigma_1 \circ \textsc{ana}_F\ \sigma_2 & (\textsc{hylo-comp}) \\
\textsc{cata}_F\ (\sigma_1 \circ F\ \sigma_2) & \equiv_{\mathsf{s}} & \textsc{cata}_F\ \sigma_1 \circ \textsc{map}_F\ \sigma_2 & (\textsc{cata-comp}) \\
\textsc{ana}_F\ (F\ \sigma_1 \circ \sigma_2) & \equiv_{\mathsf{s}} & \textsc{map}_F\ \sigma_1 \circ \textsc{ana}_F\ \sigma_2 & (\textsc{ana-comp}) \\
\textsc{ana}_F\ (F\ \sigma_1 \circ \textsc{out}) & \equiv_{\mathsf{s}} & \textsc{map}_F\ \sigma_1 & (\textsc{ana-map})
\end{array}
$$

We extend $\mathcal{E}$ in the expected way with the lifted $\equiv_{\mathsf{s}}$, $\mathcal{E}_{\mathsf{s}}$. The rule HYLO-COMP is derived from the HYLO-COMPOSE law. The rules CATA-COMP and ANA-COMP are derived from HYLO-FUSION. Since we force all atomic functions to be strict on their arguments, the strictness conditions hold in those rules. Finally, the rule ANA-MAP is derived from the HYLO-SHIFT law. It is used only to give a uniform representation of the $\textsc{map}_F$ structure. Note that the typing system describes when an expression is well-typed, using this equivalence relation. For a decision procedure, used to implement the typechecking algorithm, we refer to Section 3.4.

3.4 Determining Functional Equivalence

Recall that for all $s \in \overline{S}$, there is a Σ-indexed family E_s. For all well-typed structured expression $s : A \overset{\sigma}{\mapsto} B$, σ is an index of the family defined by s, i.e. $s^\sigma \in E_s$. The function $\phi_s : \Sigma \to E_s$ is a partial function whose result is defined for any structure σ that is an index of the family E_s. Given an $s : A \xrightarrow{\sigma'} B$, both the typechecking algorithm and the function ϕ_s need to decide whether $\sigma \equiv \sigma'$. This problem has been extensively studied for bicartesian closed categories [Har89, Yok89, Gha95], and it is beyond the scope of this thesis to produce a novel decision procedure for the equality of terms. We consequently use a simple decision procedure, but one that enables interesting parallelisations.

$$
\begin{array}{rcll}
\textsc{id} \circ \sigma & \rightsquigarrow_{\mathsf{s}} & \sigma & (\textsc{id-cancel-l}) \\
\sigma \circ \textsc{id} & \rightsquigarrow_{\mathsf{s}} & \sigma & (\textsc{id-cancel-r}) \\
\sigma \circ \sigma^{-1} & \rightsquigarrow_{\mathsf{s}} & \textsc{id} & (\textsc{inverses-cancel}) \\
F\ (\sigma_1 \circ \sigma_2) & \rightsquigarrow_{\mathsf{s}} & F\ \sigma_1 \circ F\ \sigma_2 & (\textsc{f-split}) \\
\textsc{hylo}_F\ \textsc{in}\ \textsc{out} & \rightsquigarrow_{\mathsf{s}} & \textsc{id} & (\textsc{hylo-cancel}) \\
F\ \textsc{id} & \rightsquigarrow_{\mathsf{s}} & \textsc{id} & (\textsc{f-id-cancel}) \\
\textsc{ana}_F\ (F\ \sigma_1 \circ \textsc{out}) & \rightsquigarrow_{\mathsf{s}} & \textsc{map}_F\ \sigma_1 & (\textsc{ana-map}) \\
\\
\textsc{hylo}_F\ \sigma_1\ \sigma_2 & \rightsquigarrow_{\mathsf{s}} & \textsc{cata}_F\ \sigma_1 \circ \textsc{ana}_F\ \sigma_2 & \\
\quad \Leftarrow & & \sigma_1 \neq \textsc{in}\ \wedge\ \sigma_2 \neq \textsc{out} & (\textsc{hylo-split}) \\
\\
\textsc{cata}_F\ (\sigma_1 \circ F\ \sigma_2) & \rightsquigarrow_{\mathsf{s}} & \textsc{cata}_F\ \sigma_1 \circ \textsc{map}_F\ \sigma_2 & \\
\quad \Leftarrow & & \sigma_1 \neq \textsc{in} & (\textsc{cata-split}) \\
\\
\textsc{ana}_F\ (F\ \sigma_1 \circ \sigma_2) & \rightsquigarrow_{\mathsf{s}} & \textsc{map}_F\ \sigma_1 \circ \textsc{ana}_F\ \sigma_2 & \\
\quad \Leftarrow & & \sigma_2 \neq \textsc{out} & (\textsc{ana-split})
\end{array}
$$

Figure 3.7: Rewriting system in S

3.4.1 Reforestation

We take the standard approach of using term rewriting systems to decide equality in Σ (and hence E). It is well known that if a rewriting system is confluent, then two terms have the same normal form *if and only if* they are equal with respect to the underlying equational theory. If we define a confluent term rewriting system with $\equiv$ as underlying theory, we can use the syntactic equality of normalised forms as our decision procedure. We present the rewriting system in two parts. The first part is derived from orienting the rules in $\equiv$ so that parallelism is erased:

$$
\begin{array}{lcl}
\textsc{farm}_n\ \sigma_{\mathsf{p}} & \rightsquigarrow_{\mathsf{p}} & \sigma_{\mathsf{p}} \\
\textsc{fun}\ \sigma_1 \parallel \textsc{fun}\ \sigma_2 & \rightsquigarrow_{\mathsf{p}} & \textsc{fun}\ (\sigma_1 \circ \sigma_2) \\
\textsc{dc}_{n,F}\ \sigma_1\ \sigma_2 & \rightsquigarrow_{\mathsf{p}} & \textsc{fun}\ (\textsc{hylo}_F\ \sigma_1\ \sigma_2) \\
\textsc{fb}\ (\textsc{fun}\ \sigma_1) & \rightsquigarrow_{\mathsf{p}} & \textsc{fun}\ (\textsc{iter}\ \sigma_1) \\
\\
\textsc{par}_T\ (\textsc{fun}\ \sigma_{\mathsf{s}}) & \rightsquigarrow_{\mathsf{p}} & \textsc{map}_T\ \sigma_{\mathsf{s}}
\end{array}
$$

The first four rules rewrite terms in Σ_{p} to terms in Σ_{p}, and the last rule rewrites terms in Σ to terms in Σ. We define $\overline{\Sigma_{\mathsf{s}}}$ in an analogous way to $\overline{S}$,

and `erase` as any normalisation procedure for a rewriting system $\rightsquigarrow_{\mathsf{p}}$:

$$\begin{array}{lcl} \mathsf{erase} & : & \Sigma \to \overline{\Sigma}_{\mathsf{s}} \\ \mathsf{erase}\ \sigma & = & \sigma',\ \text{s.t. } \sigma \rightsquigarrow_{\mathsf{p}}^{*} \sigma' \ \wedge\ \nexists \sigma'' \text{ s.t. } \sigma'' \rightsquigarrow_{\mathsf{p}} \sigma'' \end{array}$$

3.4.1 Lemma *The rewriting system $\rightsquigarrow_{\mathsf{p}}$ is confluent.*

Proof The rewriting system is terminating, since the number of redexes is precisely the number of parallel structures (including $\textsc{par}_T$), which is reduced following each rewriting step. It is also easy to show that any critical pairs arising from these rules have the same normal form (Appendix A). For example, a farm of a pipeline reduces to the same expression regardless of which structure is erased first. By Newman's lemma [BN98] we can conclude that $\rightsquigarrow_{\mathsf{p}}$ is confluent. □

Since $\rightsquigarrow_{\mathsf{p}}$ is confluent, we know that the result of erase is unique. Recall that all the results that are derived from the equational theory $\equiv$ can be lifted to $\mathcal{E}$. This implies that there is an $\mathsf{erase}_E : E \to \overline{S}$ procedure that is equivalent to erase defined with the rewritings lifted to E. The second step is the normalisation of $\sigma \in \overline{\Sigma}_{\mathsf{s}}$. We once again use a confluent rewriting system derived from $\equiv_{\mathsf{s}}$, and define it modulo associativity of the composition $\circ$. The rewriting system is shown in Figure 3.7 on page 87. The direction of the rewriting is chosen so a "reforestation" rewriting is performed. Hylomorphisms are first split into catamorphisms and anamorphisms, which are themselves split into compositions of maps, catamorphisms and anamorphisms. We omit some trivial cases, e.g. $F\ \sigma \circ F\ \sigma^{-1} \rightsquigarrow \textsc{id}$, and prioritise the rules that deal with $\textsc{id}$ to simplify the confluence of the rewriting system. We define σ^{-1} as follows:

$$\begin{array}{l} (\textsc{in})^{-1} = \textsc{out} \\ (\textsc{out})^{-1} = \textsc{in} \\ (F\ \sigma)^{-1} = F\ \sigma^{-1} \\ (\textsc{map}_F\ \sigma)^{-1} = \textsc{map}_F\ \sigma^{-1} \end{array}$$

The rule $\textsc{inverses-cancel}$ only applies for the structures σ that have an inverse σ^{-1}. For uniformity reasons, $\textsc{ana-map}$ is applied to the anamorphisms that perform a map computation.

3.4.2 Lemma *The term rewriting system* $\rightsquigarrow_{\mathsf{s}}$ *is confluent.*

Proof The rewriting system is terminating, since the preconditions of the rules ensure that no cycles are introduced. The rewriting system is also locally confluent. It is trivial to observe that the terms of any critical pair arising from the ID rules have the same normal form. The critical pairs arising from rules MAP-SPLIT and CATA/ANA-SPLIT can be reduced to the same normal form, by applying F-SPLIT and CATA/ANA-SPLIT and/or ANA-MAP in a different order. For example, we can rewrite any $\text{CATA}_F\ (\sigma_1 \circ F\ (\sigma_2 \circ \sigma_3)) \rightsquigarrow_{\mathsf{s}}^* \text{CATA}_F\ \sigma_1 \circ \text{MAP}_F\ \sigma_2 \circ \text{MAP}_F\ \sigma_3$. Any problems that appear from the critical pairs of the ID rules and the SPLIT rules can be solved by forcing the ID rules to be applied first, and working modulo associativity. See Appendix A for details. As before, Newman's lemma completes the proof.

Finally, we define the normalisation procedures for $\overline{\Sigma}_{\mathsf{s}}$ and Σ.

$$\begin{array}{lcl} \mathsf{norm}_{\mathsf{s}}\ \sigma & = & \sigma',\ \text{s.t. } \sigma \rightsquigarrow_{\mathsf{s}}^* \sigma' \ \wedge\ \nexists \sigma'' \text{ s.t. } \sigma' \rightsquigarrow_{\mathsf{s}} \sigma'' \\ \mathsf{norm} & = & \mathsf{norm}_{\mathsf{s}} \circ \mathsf{erase} \end{array}$$

We use a subscript, norm_E, to denote this normalisation procedure lifted to E. Given that the underlying equational theory of the term rewriting system is $\mathcal{E}$, we know that: $\forall e_1, e_2 \in E, (\mathsf{norm}_E\ e_1 = \mathsf{norm}_E\ e_2) \Leftrightarrow (e_1 \leftrightsquigarrow^*\ e_2) \Leftrightarrow (e_1\ \mathcal{E}\ e_2)$.

3.4.1 Theorem | norm defines a decision procedure for $\equiv$.
For all $\sigma_1, \sigma_2 \in \Sigma$, $\sigma_1 \equiv \sigma_2$ *if and only if* $\mathsf{norm}\ \sigma_1 = \mathsf{norm}\ \sigma_2$.

Proof From the properties of $\rightsquigarrow_{\mathsf{p}}$, we derive that it is always true that $\sigma_i \equiv \mathsf{erase}\ \sigma_i$. Since $\rightsquigarrow_{\mathsf{s}}$ is confluent, by the properties of term rewriting systems, we know that $\mathsf{erase}\ \sigma_1 \equiv \mathsf{erase}\ \sigma_2$ if and only if $\mathsf{norm}_{\mathsf{s}}(\mathsf{erase}(\sigma_1)) = \mathsf{norm}_{\mathsf{s}}(\mathsf{erase}(\sigma_2))$. We finish the proof by combining these two facts using the transitivity of $\equiv$ with the definition of norm. □

The fact that we can lift the results from Σ to E implies that we can use this rewriting system not only to reason about program equivalences, but also to define an algorithm to *derive* a parallel program from some $s \in \overline{S}$ and a type-level parallel structure. We sketch this algorithm as the definition of ϕ_s.

$$\text{EQ}\frac{\Delta = \{\{\}\}}{\sigma \sim \sigma \Rightarrow \Delta} \qquad \text{META}_1^r\frac{\Delta = \{\{m \sim \sigma\}\}}{m \sim \sigma \Rightarrow \Delta}$$

$$\text{MAP}_1\frac{\sigma_1 \sim \sigma_1' \Rightarrow \Delta_1}{F\ \sigma_1 \sim F\ \sigma_1' \Rightarrow \Delta_1} \qquad \text{MAP}_2^r\frac{\begin{array}{c}\sigma_1 \circ \sigma_2 \sim F\ m_2 \circ F\ m_3 \Rightarrow \Delta_2 \\ \Delta_1 = \{\{m_1 \sim m_2 \circ m_3\}\}\end{array}}{\sigma_1 \circ \sigma_2 \sim F\ m_1 \Rightarrow \Delta_1 \otimes \Delta_2}$$

$$\text{COMP}_1\frac{\begin{array}{c}\sigma_1 \sim \sigma_1' \Rightarrow \Delta_1 \\ \sigma_2 \sim \sigma_2' \Rightarrow \Delta_2\end{array}}{\sigma_1 \circ \sigma_2 \sim \sigma_1' \circ \sigma_2' \Rightarrow \Delta_1 \otimes \Delta_2}$$

$$\text{COMP}_2^r\frac{\begin{array}{cc}\sigma_1 \circ \sigma_2 \sim \sigma_1' \Rightarrow \Delta_{11} & \sigma_3 \sim \sigma_2' \Rightarrow \Delta_{12} \\ \sigma_2 \circ \sigma_3 \sim \sigma_2' \Rightarrow \Delta_{22} & \sigma_1 \sim \sigma_1' \Rightarrow \Delta_{21}\end{array}}{\sigma_1 \circ \sigma_2 \circ \sigma_3 \sim \sigma_1' \circ \sigma_2' \Rightarrow \Delta_{11} \otimes \Delta_{12} \cup \Delta_{21} \otimes \Delta_{22}}$$

$$\text{OP}\frac{\sigma_1 \sim \sigma_1' \Rightarrow \Delta_1 \qquad \sigma_2 \sim \sigma_2' \Rightarrow \Delta_2}{\sigma_1\langle op\rangle\sigma_2 \sim \sigma_1'\langle op\rangle\sigma_2' \Rightarrow \Delta_1 \otimes \Delta_2}$$

Figure 3.8: Unification rules (1) for metavariables, functors, composition and primitive operations.

3.4.1 Definition | ϕ_s. *Let $s \in \overline{S}$, $\sigma_1 \in \overline{\Sigma}_{\mathbf{s}}$, such that $s : A \xrightarrow{\sigma_1} B$, and $\sigma_2 \in \Sigma$. We define $\phi_s(\sigma_2)$ as follows:*

Let $\sigma_i' = \mathsf{norm}\ \sigma_i$. If $\sigma_1' = \sigma_2'$, then:

1. *Reverse the rewriting steps from σ_2 to σ_2': $\sigma_2' \rightsquigarrow^* \sigma_2$.*
2. *Obtain the proof of $\sigma_1 \equiv \sigma_2$ by using $\sigma_1 \rightsquigarrow^* \sigma_1'$ and (1).*
3. *Obtain the rewriting steps $\sigma_1 \rightsquigarrow^* \sigma_2$ from (2).*
4. *Lift the rewriting steps to E, and apply them to s: $s \rightsquigarrow_E^* e$.*

Since the typechecking algorithm for our type system needs to decide $\sigma_1 \equiv \sigma_2$ (e.g. using Thm. 3.4.1), steps *(1)* to *(2)* can be omitted if we know that $s : A \xrightarrow{\sigma_2} B$ (recall the proof of Thm. 3.3.1). Conversely, if there is some $sA \xrightarrow{\sigma_1} B$, and $s^{\sigma_2} \in E_s$, we know that there is a proof $\sigma_1 \equiv \sigma_2$ (step (2) in Def. 3.4.1), and therefore $sA \xrightarrow{\sigma_2} B$ (recall the proof of Thm. 3.3.2).

$$\textsc{hylo}_1 \frac{\sigma_1 \sim \sigma_1' \Rightarrow \Delta_1 \qquad \sigma_2 \sim \sigma_2' \Rightarrow \Delta_2}{\textsc{hylo}_F\ \sigma_1\ \sigma_2 \sim \textsc{hylo}_F\ \sigma_1'\ \sigma_2' \Rightarrow \Delta_1 \otimes \Delta_2}$$

$$\textsc{hylo}_2^r \frac{\begin{array}{c}\sigma_1 \circ \sigma_2 \ \sim\ \textsc{hylo}_F\ m_1\ \textsc{out} \circ \textsc{hylo}_F\ \textsc{in}\ m_2 \Rightarrow \Delta_1 \\ \sigma_1 \circ \sigma_2 \ \sim\ \textsc{hylo}_F\ m_1\ \textsc{out} \Rightarrow \Delta_2 \\ \sigma_1 \circ \sigma_2 \ \sim\ \textsc{hylo}_F\ \textsc{in}\ m_2 \Rightarrow \Delta_3\end{array}}{\begin{array}{l}\sigma_1 \circ \sigma_2 \ \sim\ \textsc{hylo}_F\ m_1\ m_2 \\ \quad \Rightarrow \Delta_1 \cup \{\{m_2 \sim \textsc{out}\}\} \otimes \Delta_2 \cup \{\{m_1 \sim \textsc{in}\}\} \otimes \Delta_3\end{array}}$$

$$\textsc{hylo}_3^r \frac{\begin{array}{c}\sigma_1 \circ \sigma_2 \ \sim \textsc{hylo}_F\ (\textsc{in} \circ F\ m_2)\ \textsc{out} \circ \textsc{hylo}_F\ \textsc{in}\ m_3 \Rightarrow \Delta_1 \\ \sigma_1 \circ \sigma_2 \ \sim \textsc{hylo}_F\ (\textsc{in} \circ F\ m_2)\ \textsc{out} \Rightarrow \Delta_2 \\ \Delta = \{\{m_1 \sim F\ m_2 \circ m_3\}\}\end{array}}{\begin{array}{l}\sigma_1 \circ \sigma_2 \ \sim\ \textsc{hylo}_F\ \textsc{in}\ m_1 \\ \quad \Rightarrow \Delta \otimes \Delta_1 \cup \Delta \otimes \{\{m_3 \sim \textsc{out}\}\} \otimes \Delta_2\end{array}}$$

$$\textsc{hylo}_4^r \frac{\sigma_1 \circ \sigma_2 \ \sim \textsc{hylo}_F\ m_2\ \textsc{out} \circ \textsc{hylo}_F\ (\textsc{in} \circ F\ m_3)\ \textsc{out} \Rightarrow \Delta}{\sigma_1 \circ \sigma_2 \ \sim\ \textsc{hylo}_F\ m_1\ \textsc{out} \Rightarrow \{\{m_1 \sim m_2 \circ F\ m_3\}\} \otimes \Delta}$$

$$\textsc{hylo}_5^r \frac{\begin{array}{c}\sigma_1 \circ \sigma_2 \sim \textsc{hylo}_F\ m_1\ \textsc{out} \circ \textsc{hylo}_F\ \textsc{in}\ \sigma_3 \Rightarrow \Delta_1 \\ \sigma_1 \circ \sigma_2 \sim \mathsf{norm}\ (\textsc{hylo}_F\ \textsc{in}\ \sigma_3) \Rightarrow \Delta_2 \\ \sigma_3 \neq \textsc{out}\end{array}}{\begin{array}{l}\sigma_1 \circ \sigma_2 \sim \textsc{hylo}_F\ m_1\ \sigma_3 \\ \quad \Rightarrow \Delta_1 \cup \{\{m_1 \sim \textsc{in}\}\} \otimes \Delta_2\end{array}}$$

$$\textsc{hylo}_6^r \frac{\begin{array}{c}\sigma_1 \circ \sigma_2 \sim \textsc{hylo}_F\ \sigma_3\ \textsc{out} \circ \textsc{hylo}_F\ \textsc{in}\ m_1 \Rightarrow \Delta_1 \\ \sigma_1 \circ \sigma_2 \sim \mathsf{norm}\ (\textsc{hylo}_F\ \sigma_3\ \textsc{out}) \Rightarrow \Delta_2 \qquad \sigma_3 \neq \textsc{in}\end{array}}{\sigma_1 \circ \sigma_2 \sim \textsc{hylo}_F\ \sigma_3\ m_1 \Rightarrow \Delta_1 \cup \{\{m_1 \sim \textsc{out}\}\} \otimes \Delta_2}$$

$$\textsc{hylo}_7^r \frac{\sigma_1 \circ \sigma_2 \sim \textsc{hylo}_F\ (\textsc{in} \circ F\ m_2)\ \textsc{out} \circ \textsc{hylo}_F\ (\textsc{in} \circ F\ m_3)\ \textsc{out} \Rightarrow \Delta}{\sigma_1 \circ \sigma_2 \sim \textsc{hylo}_F\ (\textsc{in} \circ F\ m_1)\ \textsc{out} \Rightarrow \{\{m_1 \sim m_2 \circ m_3\}\} \otimes \Delta}$$

Figure 3.9: Unification rules (2) for hylomorphisms with metavariables.

3.4.2 Structure Unification

The structure-annotated types that we have presented so far require the specification of a full structure $\sigma \in \Sigma$. However, it is sometimes sufficient, or desirable, to specify only the relevant parts of this structure. We allow this

by introducing *structure metavariables* in Σ. Selecting suitable substitutions for these metavariables can be automated in different ways, as we will see later in this section. Given a set of metavariables, $\mathcal{M}$, we extend the syntax of Σ as follows:

$$\begin{array}{llll} m \in \mathcal{M} & & \sigma \in \Sigma & ::= \ \ldots \mid m \\ \sigma_{\mathsf{s}} \in \Sigma_{\mathsf{s}} & ::= \ \ldots \mid m & \sigma_{\mathsf{p}} \in \Sigma_{\mathsf{p}} & ::= \ \ldots \mid m \end{array}$$

The underscore character denotes a fresh metavariable, e.g. given a fresh metavariable m, $\textsc{farm}_n \ \_$ is equivalent to $\textsc{farm}_n \ m$.

3.4.2 Definition | Substitution Environments. *A substitution environment δ is a mapping of metavariables to structures, $\{m_1 \sim \sigma_1, m_2 \sim \sigma_2, \ldots\}$. We use Δ to denote sets of environments δ.*

The two basic operations with substitution environments are the *application* and the *extension*. We apply a substitution environment δ to a structure σ, denoted by $\delta\sigma$, by replacing all metavariables as defined by δ. The extension of δ_1 with δ_2, $\delta_1\delta_2$, is defined in the expected way. If both substitution environments introduce a cycle or conflicting metavariables, the operation fails. Finally, for sets of substitution environments, we define the set of extensions:

$$\Delta_1 \otimes \Delta_2 \ = \ \{\delta_1\delta_2 \mid \delta_1 \in \Delta_1 \wedge \delta_2 \in \Delta_2\}$$

3.4.3 Lemma *For all substitutions δ, for all $\sigma \in \Sigma$, $\mathsf{norm}\ \delta\sigma \equiv \delta(\mathsf{norm}\ \sigma)$.*

Proof If $\sigma_1 \equiv \sigma_2$, then $\delta\sigma_1 \equiv \delta\sigma_2$, since δ will apply the same substitution in both σ_1 and σ_2. We know that $\forall\sigma,\ \sigma \equiv \mathsf{norm}\ \sigma$. We use these two facts to conclude:

$$\begin{array}{l} \sigma \equiv \mathsf{norm}\ \sigma \\ \quad \Rightarrow \delta\sigma \equiv \delta(\mathsf{norm}\ \sigma) \\ \quad \Rightarrow \mathsf{norm}\ \delta\sigma \equiv \delta(\mathsf{norm}\ \sigma) \end{array}$$

□

Note that we can no longer use the relation $\equiv$ in our typechecking rules, since it does not handle metavariables. Instead, we define the relation $\cong$, and define a decision procedure for it.

3.4.3 Definition | Equivalence of the Unified Forms. *We say that* $\sigma_1 \cong \sigma_2$ *if there is at least a substitution* δ *that makes* $\delta\sigma_1 \equiv \delta\sigma_2$.

$$\sigma_1 \cong \sigma_2 \doteq \exists \delta, \delta\sigma_1 \equiv \delta\sigma_2$$

The type rule for equivalence changes to use the new relation:

$$\frac{e \ : \ A \xrightarrow{\sigma_1} B \qquad \sigma_1 \cong \sigma_2}{e \ : \ A \xmapsto{\sigma_2} B}$$

In order to typecheck a structured expression with a structure containing metavariables, we need to: i) modify the normalisation procedure; and ii) define a *unification algorithm*. The normalisation procedure is modified as follows:

1. Any erase step on a structure with meta-variables always succeeds by adding new meta-variables and a substitution environment δ for those metavariables, e.g.

   $m_1 \parallel m_2 \quad \rightsquigarrow \quad \text{FUN } (m_2' \circ m_1')$

   $\delta = \{m_1 \sim \text{FUN } m_1',\ m_2 \sim \text{FUN } m_2'\}$.

2. The constraints of the rules used by the norm_S procedure are modified so that they are never satisfied by metavariables, e.g.

   $\text{HYLO}_F\ \sigma_1\ \sigma_2 \ \rightsquigarrow_{\mathsf{s}}\ \text{CATA}_F\ \sigma_1 \circ \text{ANA}_F\ \sigma_2 \qquad (\text{HYLO-SPLIT})$

   $\Leftarrow \quad \sigma_1 \neq \text{IN} \ \wedge \ \sigma_2 \neq \text{OUT} \ \wedge \ \sigma_1 \notin \mathcal{M} \ \wedge \ \sigma_2 \notin \mathcal{M}$

Although condition 2 is not necessary, it simplifies the unification of structures where σ_1 or σ_2 can be unified to IN or OUT.

Unification Rules. Since unifying two structures may lead to different, but valid unifying substitutions, the unification rules yield the set of all possible *unifying substitutions*, Δ. Disambiguating such situations can be done using cost models, or some other procedure. The rules presented in Figures 3.8–3.9 on pages 90 and 91 define unification of terms written as compositions of hylomorphisms, modulo associativity (rule COMP_2). Each rule with superscript r has a symmetric version l. A statement $\sigma_1 \sim \sigma_2 \Rightarrow \Delta$

means that structure σ_1 unifies with structure σ_2, under a non-empty set of substitutions, $\Delta \neq \varnothing$.

The intuition behind those rules is that, whenever two structures do not correspond to the same syntactic structure, the unification rules make any valid assumption about the metavariables that would allow further rewritings to take place. For example, rule HYLO_3^r, on page 91 states that whenever we are unifying a composition of two structures, $\sigma_1 \circ \sigma_2$ with a hylomorphism with a metavariable, HYLO_F IN m_1, we know that m_1 must unify to $F\ m_2 \circ m_3$, since this is the only way to split the hylomorphism into a composition.

3.4.2 Theorem | Soundness of the Unification.
For all $\sigma_1, \sigma_2 \in \Sigma$,
$\sigma_1 \sim \sigma_2 \Rightarrow \Delta \Longrightarrow \Delta \neq \varnothing \wedge \forall \delta \in \Delta,\ \delta\sigma_1 \equiv \delta\sigma_2$

3.4.3 Theorem | Completeness of the Unification.
For all $\sigma_1, \sigma_2 \in \Sigma$, *and substitution* δ,
$\delta\sigma_1 \equiv \delta\sigma_2 \Longrightarrow \exists \Delta$ *s.t.* $\Delta \neq \varnothing \wedge \sigma_1 \sim \sigma_2 \Rightarrow \Delta$

The proofs of those theorems are standard proofs by induction on the derivations of $\sim$ and $\equiv$, and by case analysis on the metavariables.

3.4.1 Corollary $\cong$*:* $\sigma_1 \cong \sigma_2 \ \Leftrightarrow\ \mathsf{norm}\ \sigma_1 \sim \mathsf{norm}\ \sigma_2 \Rightarrow \Delta$*, i.e. the unification algorithm can be used as a decision procedure for* $\cong$.

Proof For the proof of the $\Rightarrow$ case, we know that there is at least a δ such that $\delta\sigma_1 \equiv \delta\sigma_2$. From the properties of $\equiv$, we know that $\mathsf{norm}\ \delta\sigma_1 = \mathsf{norm}\ \delta\sigma_2$. Using Lemma 3.4.3, we derive that $\delta\ (\mathsf{norm}\ \sigma_1) \equiv \delta\ (\mathsf{norm}\ \sigma_2)$. The completeness of the unification allows us to conclude that $\mathsf{norm}\ \sigma_1 \sim \mathsf{norm}\ \sigma_2 \Rightarrow \Delta$.
The proof of $\Leftarrow$ follows from the soundness of the unification algorithm. We know that $\mathsf{norm}\ \sigma_1 \sim \mathsf{norm}\ \sigma_2 \Rightarrow \Delta$ implies that Δ is non-empty, and that for all $\delta \in \Delta$, $\delta\ (\mathsf{norm}\ \sigma_1) \equiv \delta\ (\mathsf{norm}\ \sigma_2)$. We conclude by selecting any δ from Δ, and then using Lemma 3.4.3. □

Using metavariables in structures has some implications. Given a $sA \xrightarrow{\sigma_1} B$, and a structure σ_2 containing one or more metavariables, σ_2 can no longer

be used as an index for a family E_s. Since there may be alternative, but valid substitutions for the metavariables, it follows that $s^{\sigma_2} \subseteq E_s$. Given a unification $\sigma_1 \sim \sigma_2 \Rightarrow \Delta$, we need to apply a $\delta \in \Delta$ to σ_2 in order to use it as an index, $s^{\delta\sigma_2} \in E_s$. This implies that there are many ways to use our approach. On one hand, fixing this σ_2 to be a closed structure without any metavariables, or one that unifies with σ_1 with a unique substitution, provides a way to manually parallelise a program. On the other hand, if σ_2 is defined to be a metavariable, then a fully automated method for selecting a parallel structure would be needed. In between, there are a wide range of *semi-automated* possibilities that can be used to reason about the introduction of parallelism to a program. The automated selection mechanism for a $\delta \in \Delta$ can be extended with further parallelisation opportunities. Furthermore, it can be parameterised by architecture-specific details, so that compiling a program for different architectures leads to alternative parallelisations. This is, however, beyond the scope of this thesis.

Compositionality and Higher-Order Structured-Arrows. We finish this section with a discussion of the compositionality of our approach. Most of this thesis deals with the typing rules for structure-annotated arrows. Extending our work for a language with definitions and with a limited-form of higher-order structured arrows can be done using the unification rules from Fig. 3.8–3.9. Applying some $e : A \overset{\sigma_1'}{\longmapsto} B$ to a $f : A \overset{\sigma_1}{\longmapsto} B \to C \overset{\sigma_2}{\longmapsto} D$ typechecks only if $\sigma_1 \sim \sigma_1' \Rightarrow \Delta$, and the type of this would be annotated with a structure resulting from applying any $\delta \in \Delta$, $f\ e : C \overset{\delta\sigma_2}{\longmapsto} D$. One advantage of this is to allow the specification of new structures that can be later used in our programs. The corresponding erasure rules would then be derived automatically from such a specification. We would need, however, to verify a back-end for such a language in order to ensure that the operational semantics of the new structures are sound with respect to the specification. Although we do not explain this idea in detail in this chapter, we will use a small example of a defined structure in Section 3.5.3.

Although it seems entirely feasible to support full higher-order structured arrows, there is the question of whether this is desirable: what would types such as $A \overset{\sigma}{\mapsto} B \overset{\sigma'}{\longmapsto} C$ or $(A \overset{\sigma}{\mapsto} B) \overset{\sigma'}{\longmapsto} C$ mean? Intuitively, the

first would be a parallel process with structure σ that produces a parallel process with structure σ', and the second would be a parallel process with structure σ' that takes a parallel process with structure σ as input, and produces an output of type C. In contrast to using functions with a type such as $X \overset{\sigma_1}{\longmapsto} Y \to A \overset{\sigma_2}{\longmapsto} B$, we have so far not seen the benefits of full higher-order functions in our examples, although this is an idea that may be worth exploring in future work.

3.5 Examples

Our first two examples revisit *image merge* from Section 3.1, and *quicksort* from Section 3.2. In both cases, we show how the type system calculates the convertibility of the structured expression to a functionally equivalent parallel process, and how the convertibility proof allows the structured expression to be rewritten to the desired parallel process. The final two examples show how to use types to parallelise different algorithms, described as structured expressions. These examples show how to introduce parallel structure to these algorithms using our type system.

3.5.1 Image Merge

Recall that *image merge* composes two functions: `mark` and `merge`. It can be directly parallelised using different combinations of farms and pipelines.

$$\begin{aligned}
&\textsc{im}_1\ (n, m)\ =\ \textsc{par}_L\ (\textsc{farm}\ n\ (\textsc{fun}\ \textsc{a})\ \|\ \textsc{farm}\ m\ (\textsc{fun}\ \textsc{a})) \\
&\texttt{imageMerge}\ :\ \mathsf{List}(\texttt{Img} \times \texttt{Img}) \overset{\textsc{im}_1\ (n,m)}{\longmapsto} \mathsf{List}(\texttt{Img} \times \texttt{Img}) \\
&\texttt{imageMerge}\ =\ \texttt{map}_{\mathsf{List}}\ (\texttt{merge} \circ \texttt{mark})
\end{aligned}$$

First, we use our annotated typing rules to produce a derivation tree with the structure of the expression (Fig. 3.10 on page 97). The key part is the convertibility proof, that $\textsc{map}_L\ (\textsc{a} \circ \textsc{a}) \cong \textsc{im}_1\ (n, m)$. We use the decision procedure defined in Section 3.3 to decide the equivalence of both structures. The erase step is applied as follows:

$$\textsc{im}_1\ (n, m)\ \rightsquigarrow^*\ \textsc{map}_L\ (\textsc{a} \circ \textsc{a})$$

$$\dfrac{\dfrac{\dfrac{\texttt{replace} \;:\; \texttt{Img} \times \texttt{Img} \overset{\mathrm{A}}{\longmapsto} \texttt{Img} \qquad \texttt{mark} \;:\; \texttt{Img} \times \texttt{Img} \overset{\mathrm{A}}{\longmapsto} \texttt{Img} \times \texttt{Img}}{\texttt{replace} \circ \texttt{mark} \;:\; \texttt{Img} \times \texttt{Img} \overset{\mathrm{A} \circ \mathrm{A}}{\longmapsto} \texttt{Img} \times \texttt{Img}}}{\mathsf{map}_{\mathsf{List}}\,(\texttt{replace} \circ \texttt{mark}) \;:\; \mathsf{List}(\texttt{Img} \times \texttt{Img}) \overset{\mathrm{MAP}_L\ (\mathrm{A} \circ \mathrm{A})}{\longmapsto} \mathsf{List}(\texttt{Img} \times \texttt{Img})} \qquad \mathrm{MAP}_L\ (\mathrm{A} \circ \mathrm{A}) \;\cong\; \mathrm{IM}_1\ (n, m)}{\mathsf{map}_{\mathsf{List}}\,(\texttt{replace} \circ \texttt{mark}) \;:\; \mathsf{List}(\texttt{Img} \times \texttt{Img}) \overset{\mathrm{IM}_1\ (n,m)}{\longmapsto} \mathsf{List}(\texttt{Img} \times \texttt{Img})}$$

Figure 3.10: Typing Derivation Tree for Image Merge

The final step involves applying the decision procedure for equality of $\overline{S}$. Since the expressions are identical, this is a trivial step. We can now apply this equivalence to the original expression:

$$\begin{aligned}&\mathsf{map}_{\mathsf{List}}\ (\mathtt{merge}\ \circ\ \mathtt{mark}) \rightsquigarrow^{*}\\ &\quad \mathsf{par}_{\mathsf{List}}\ (\mathsf{farm}\ n\ (\mathsf{fun}\ \mathtt{mark})\ \|\ \mathsf{farm}\ m\ (\mathsf{fun}\ \mathtt{merge}))\end{aligned}$$

We now show an example of unification. We define $\textsc{im}_2\ n = \textsc{par}_L\ (\_\ \|\ \textsc{farm}\ n\ \_)$. First, we instantiate the structure with fresh metavariables m_1 and m_2. Then, we normalise the structure. We start by applying the erase rewriting system:

$$\textsc{map}_L\ (m_2' \circ m_1') \qquad\qquad \delta = \{m_1 \sim \textsc{fun}\ m_1',\ m_2 \sim \textsc{fun}\ m_2'\}$$

We then apply the normalisation in $\overline{\Sigma_{\mathsf{s}}}$, and the unification rules:

$$\textsc{map}_L\ \textsc{a} \circ \textsc{map}_L\ \textsc{a} \sim \textsc{map}_L\ m_2' \circ \textsc{map}_L\ m_1' \Rightarrow \{\{m_1' \sim \textsc{a}, m_2' \sim \textsc{a}\}\}$$

The final step is to calculate the extension of the environment δ, and the set of environments that are obtained from the unification:

$$\begin{aligned}\Delta = \{\delta\} \otimes \{\{m_1' \sim \textsc{a}, m_2' \sim \textsc{a}\}\} =\\ \{\{m_1 \sim \textsc{fun}\ m_1',\ m_2 \sim \textsc{fun}\ m_2', m_1' \sim \textsc{a}, m_2' \sim \textsc{a}\}\}\end{aligned}$$

Applying the substitution environment in Δ to the $\textsc{im}_2(n)$, we obtain the structure $\textsc{par}_L\ (\textsc{fun}\ \textsc{a}\ \|\ \textsc{farm}_n\ (\textsc{fun}\ \textsc{a}))$.

Finally, we briefly discuss how to extend the environment further using a procedure min `cost`. First, min attempts further rewritings to m_1 and m_2. To ensure termination, the process stops whenever the only option is to introduce a task farm to an existing task farm structure.

$$\begin{aligned}\delta_1 &= \{m_1 \sim \textsc{farm}\ n_1\ (\textsc{fun}\ \textsc{a}),\ \ m_2 \sim \textsc{farm}\ n_2\ (\textsc{fun}\ \textsc{a})\}\\ \delta_2 &= \{m_1 \sim \textsc{fun}\ \textsc{a},\ \ m_2 \sim \textsc{farm}\ n_2\ (\textsc{fun}\ \textsc{a})\}\\ \delta_3 &= \{m_1 \sim \textsc{farm}\ n_1\ (\textsc{fun}\ \textsc{a}),\ \ m_2 \sim \textsc{fun}\ \textsc{a}\}\\ \delta_4 &= \{m_1 \sim \textsc{fun}\ \textsc{a},\ \ m_2 \sim \textsc{fun}\ \textsc{a}\}\end{aligned}$$

We will show how to select one of those structures using a simple example cost model. In future work, we will consider how to extend and formalise this cost model. A cost model provides size functions $|\sigma|$ over structures, similar

to the idea of *sized types* [HPS96]. We assume that all atomic functions are annotated with their cost models, A_c. The cost of a structure is a function that receives a size, sz, and returns an estimation of its run-time in milliseconds.

In our example, we assume that $sz = [d]^{1000}$. This represents the size of 1000 pairs of images of d dimensions. The size function of the first stage $|\mathrm{A}_{c_1}|$ is the identity, since we are not modifying the images. The parameters for the number of farm workers are fixed to be those with the least cost, given some number of available cores. We assume that a maximum of 24 cores are available for this example. For δ_1, we determine that $n_1 = 9$, $n = 3$ and $n_2 = 5$. The values of the costs on those sizes, and the overheads of farms and pipelines (κ_1 and κ_2) are given below.

$$
\begin{array}{lll}
c_1\ [(2048 \times 2048, 2048 \times 2048)]^n & = & n \times 25.11ms \\
c_2\ [(2048 \times 2048, 2048 \times 2048)]^n & = & n \times 45.21ms \\
\kappa_1\ (9) = 29.66ms & & \kappa_1\ (3 \times 5) = 60.93ms \\
\multicolumn{3}{c}{\kappa_2\ (9, 3 \times 5) = 114.4ms}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{cost}\ (\delta_1 \mathrm{IM}_2(n))\ sz} \\
\multicolumn{3}{l}{= \max\ \{c_1\ (\frac{sz}{n_1}) + \kappa_1(n_1),\ c_2\ (\frac{|\mathrm{A}_{c_1}|(sz)}{n \times n_2}) + \kappa_1(n \times n_2)\}} \\
\multicolumn{3}{l}{\quad + \kappa_2(n_1,\ n \times n_2) = 3145.69ms} \\
\texttt{cost}\ (\delta_2 \mathrm{IM}_2(n))\ sz & = & 25123.81ms \\
\texttt{cost}\ (\delta_3 \mathrm{IM}_2(n))\ sz & = & 3189.60ms \\
\texttt{cost}\ (\delta_4 \mathrm{IM}_2(n))\ sz & = & 25123.81ms
\end{array}
$$

The structure that results from applying δ_1 is the least cost one, $\delta_1(\mathrm{IM}_2(3))$, with $n_1 = 9$ and $n_2 = 5$.

3.5.2 Quicksort

We will now revisit *quicksort* and show how it can exploit a divide-and-conquer parallel structure.

$$
\begin{array}{l}
\texttt{qsort}\ :\ \mathsf{List}(\mathsf{List}\ A) \to \mathsf{List}(\mathsf{List}\ A) \\
\texttt{qsort}\ =\ \texttt{map}_{\mathsf{List}}\ (\texttt{hylo}_{F\,A}\ \texttt{merge}\ \texttt{div})
\end{array}
$$

In order to introduce a divide-and-conquer parallel structure, the type system needs to decide:

$$\textsc{map}_L(\textsc{hylo}_F\ \textsc{a}\ \textsc{a}) \cong \textsc{par}_L\ (\textsc{dc}_{n,F}\ \textsc{a}\ \textsc{a})$$

This can be achieved using a simple parallelism erasure. Consider now a slightly more complex structure:

$$\textsc{map}_L\ (\textsc{hylo}_F\ \textsc{a}\ \textsc{a})\ \cong\ \textsc{par}_L\ (\textsc{farm}_n\ \_\ \|\ \_)$$

Let m_1, m_2 be two fresh metavariables. The parallelism erasure of the right hand side returns the following structure and substitution:

$$\begin{aligned}\textsc{par}_L\ (\textsc{farm}\ n\ m_2\ \|\ m_1)\ &\rightsquigarrow^*\ \textsc{map}_L\ (m_1' \circ m_2')\\ \delta = \{m_1\ &\sim\ \textsc{fun}\ m_1', m_2 \sim \textsc{fun}\ m_2'\}\end{aligned}$$

The normalisation procedure continues by normalising the left and right hand sides of the equivalence following a parallelism erasure. The left hand side is normalised by applying HYLO-SPLIT, F-SPLIT and CATA-SPLIT:

$$\textsc{map}_L\ (\textsc{hylo}_F\ \textsc{a}\ \textsc{a}) \rightsquigarrow^*\ \textsc{map}_L\ (\textsc{cata}_F\ \textsc{a})\ \circ\ \textsc{map}_L\ (\textsc{ana}_F\ \textsc{a})$$

The right hand side of the equivalence is normalised by applying F-SPLIT and CATA-SPLIT:

$$\textsc{map}_L\ (m_1' \circ m_2') \rightsquigarrow^*\ \textsc{map}_L\ m_1' \circ \textsc{map}_L\ m_2'$$

The decision procedure finishes by unifying both structures, and extending the substitution δ with all possible unifications.

$$\begin{aligned}\textsc{map}_L\ (\textsc{cata}_F\ \textsc{a})\ \circ\ \textsc{map}_L\ (\textsc{ana}_F\ \textsc{a}) &\sim \textsc{map}_L\ m_1' \circ \textsc{map}_L\ m_2'\\ \Rightarrow \Delta_1 = \{m_1' &\sim \textsc{cata}_F\ \textsc{a}, m_2' \sim \textsc{ana}_F\ \textsc{a}\}\end{aligned}$$

$$\Delta = \{\delta\} \otimes \Delta_1$$

Again, by applying the only substitution in $\delta' \in \Delta$, we select the final structure:

$$\textsc{par}_L\ (\textsc{farm}_n\ (\textsc{fun}\ (\textsc{ana}_F\ \textsc{a}))\ \|\ \textsc{fun}\ (\textsc{cata}_F\ \textsc{a}))$$

The full proof of equivalence ($\cong$) allows us to rewrite quicksort to our desired parallel structure:

$$\begin{aligned}&\mathsf{map}_{\mathsf{List}}(\mathtt{hylo}_{F\ A}\mathtt{merge}\ \mathtt{div}) \rightsquigarrow^{*}\\ &\quad \mathsf{par}_{\mathsf{List}}\ (\mathsf{farm}\ n\ (\mathsf{fun}\ (\mathsf{ana}_{F\ A}\ \mathtt{div}))\ \|\ \mathsf{fun}\ (\mathtt{cata}_{F\ A}\mathtt{merge}))\end{aligned}$$

We can use our cost model again, where κ_3 is the overhead of a divide-and-conquer structure. In this example, we set the size parameter of our cost model to 1000 lists of 3,000,000 elements, and use the following structure:

$$\mathtt{qsort}\ :\ \mathsf{List}(\mathsf{List}\ A) \xmapsto{\text{min cost}} \mathsf{List}(\mathsf{List}\ A)$$

$$\begin{aligned}&\mathtt{cost}\ (\text{PAR}_L\ (\text{DC}_{n,F}\ \text{A}_{c_1}\ \text{A}_{c_2}))\ sz\\ &=\ \max\{\max_{1\leq i\leq n}\{c_2\ (|\text{A}_{c_2}|^i sz)\\ &\quad ,\mathtt{cost}\ (\text{HYLO}_F\ \text{A}_{c_1}\ \text{A}_{c_2})\ (|\text{A}_{c_2}|^n sz)\\ &\quad ,\max_{1\leq i\leq n}\{c_1\ (|\text{A}_{c_1}|^i|\text{A}_{c_2}|^n sz)\}\}\ +\ \kappa_3(n)\ =\ 42602.72ms\\ &\mathtt{cost}\ (\text{PAR}_L\ (\text{FARM}_n\ (\text{FUN}\ (\text{ANA}_L\ \text{A}_{c_2}))\ \|\ (\text{FUN}\ (\text{CATA}_L\ \text{A}_{c_1}))))\ sz\\ &=\ 27846.13ms\\ &\mathtt{cost}\ (\text{PAR}_L\ (\text{FARM}_n\ (\text{FUN}\ (\text{HYLO}_F\ \text{A}_{c_1}\ \text{A}_{c_2}))))\ sz\\ &=\ 32179.77ms\\ &\ldots\end{aligned}$$

Since the most expensive part of the quicksort is the divide, and flattening a tree is linear, the cost of adding a farm to the divide part is less than using a divide-and-conquer skeleton for this example.

3.5.3 N-Body Simulation

N-Body simulations are widely used in astrophysics. They comprise a simulation of a dynamic system of particles, usually under the influence of physical forces. The Barnes-Hut simulation recursively divides the n bodies storing them in an `Octree`, or an 8-ary tree. Each node in the tree represents a region of the space, where the topmost node represents the whole space and the eight children the eight octants of the space. The leaves of the tree contain the bodies. Then, the cumulative mass and centre of mass of each region of the space are calculated. Finally, the algorithm calculates the net force on each particular body by traversing the tree, and updates its

velocity and position. This process is repeated for a number of iterations. We will here abstract most of the concrete, well known details of the algorithm, and present its high-level structure, using the following types and functions:

$$\begin{array}{lcl} C & = & \mathbb{Q} \times \mathbb{Q} \\ F\ A\ B & = & A\ +\ C \times B^8 \\ G\ A & = & F\ \texttt{Body} \\ \texttt{Octree} & = & \mu G \\ \texttt{insert} & : & \texttt{Body} \times \texttt{Octree} \to \texttt{Octree} \end{array}$$

Since this algorithm also involves iterating for a fixed number of steps, we define iteration as a hylomorphism. We assume that the combinator $+$ (Section 2.3.2 on page 40) is also defined in Σ_{s}. Additionally, we assume a primitive combinator, that tests a predicate on a value, $(\cdot\ ?)\ :\ (A \to \texttt{Bool}) \to A \to A + A$.

$$\begin{array}{l} \textsc{loop}\ :\ \Sigma \to \Sigma \\ \textsc{loop}\ \sigma\ =\ \textsc{hylo}_{(+)}\ (\textsc{id} \triangledown \sigma)\ ((\textsc{a} + (\textsc{a} \vartriangle (\textsc{a} \circ \textsc{a}))) \circ (\textsc{a} \circ \textsc{a}?)) \end{array}$$

$$\begin{array}{l} \texttt{loop}_A\ :\ (A \overset{m}{\mapsto} A) \to A \times \mathbb{N} \overset{\text{LOOP } m}{\longmapsto} A \\ \texttt{loop}_A\ s\ = \\ \quad \texttt{hylo}_{(A+)}\ (\mathsf{id} \triangledown s) \\ \qquad ((\pi_1 + (\pi_1 \vartriangle ((-1) \circ \pi_2))) \circ ((== 0) \circ \pi_2)?) \end{array}$$

This example uses some additional functions: `calcMass` annotates each node with the total mass and centre of mass; `dist` distributes the octree to all the bodies, to allow independent calculations, `calcForce` calculates the force of one body; and `move` updates the velocity and position of the body.

$$\begin{array}{l} \texttt{calcMass}\ :\ G\ \texttt{Octree} \to G\ \texttt{Octree} \\ \texttt{dist}\ :\ \texttt{Octree} \times \mathsf{List}\ \texttt{Body} \\ \qquad \to L\ (\texttt{Octree} \times \texttt{Body})\ (\texttt{Octree} \times \mathsf{List}\ \texttt{Body}) \end{array}$$

The algorithm is:

$$\begin{array}{l} \texttt{nbody}\ :\ \mathsf{List}\ \texttt{Body} \times \mathbb{N} \overset{\text{LOOP } \sigma}{\longmapsto} \mathsf{List}\ \texttt{Body} \\ \texttt{nbody}\ =\ \texttt{loop}\ (\texttt{ana}_L\ (L\ (\texttt{move} \circ \texttt{calcForce}) \circ \texttt{dist}) \\ \qquad \circ ((\texttt{cata}_G\ (\mathsf{in}_G \circ \texttt{calcMass}) \circ \texttt{cata}_L\ \texttt{insert}) \vartriangle \mathsf{id})) \end{array}$$

Since the LOOP defines a fixed structure, we do not allow any rewriting that changes this structure. However, note that our type system still enables some interesting rewritings. In particular, the structure of the loop body is:

$$\sigma = \text{ANA}_L(L\,(\text{A} \circ \text{A}) \circ \text{A}) \circ (\text{CATA}_G(\text{IN} \circ \text{A}) \circ \text{CATA}_L\ \text{A})\ \triangle\ \text{ID}$$

The normalised structure reveals more opportunities for parallelism introduction:

$$\sigma = \text{MAP}_L\text{A} \circ \text{MAP}_L\ \text{A} \circ \text{ANA}_L\ \text{A} \circ (\text{CATA}_G(\text{IN} \circ \text{A}) \circ \text{CATA}_L\ \text{A})\ \triangle\ \text{ID}$$

After normalisation, this structure is equivalent to:

$$\sigma = \text{PAR}_L\ (\text{FUN}\ (\text{A} \circ \text{A})) \circ \_$$

The structure makes it clear that there are many possibilities for parallelism using farms and pipelines. As before, parallelism can be introduced semi-automatically using a cost model. For example, setting the input size to 20,000 bodies:

$$\begin{aligned}
\sigma &= \text{PAR}_L\ (\text{FARM}\ n\ _ \parallel _) \circ _ \\
\sigma' &= \text{PAR}_L\ (\min\ \texttt{cost}\ (_ \parallel _)) \circ _
\end{aligned}$$

$$\begin{aligned}
&\texttt{cost}\ (\text{FUN}\ \text{A}_{c_1} \parallel \text{FUN}\ \text{A}_{c_2})\ sz\ =\ 310525.67ms \\
&\texttt{cost}\ (\text{FARM}_6\ (\text{FUN}\ \text{A}_{c_1}) \parallel (\text{FUN}\ \text{A}_{c_2}))\ sz\ =\ 55755.43ms \\
&\texttt{cost}\ (\text{FUN}\ \text{A}_{c_1} \parallel \text{FARM}_1\ (\text{FUN}\ \text{A}_{c_2}))\ sz\ =\ 310525.67ms \\
&\texttt{cost}\ (\text{FARM}_{20}(\text{FUN}\ \text{A}_{c_1}) \parallel \text{FARM}_4(\text{FUN}\ \text{A}_{c_2}))\ sz\ =\ 15730.46ms
\end{aligned}$$

3.5.4 Iterative Convolution

Image convolution is also widely used in image processing applications. We assume the type `Img` of images, the type `Kern` of kernels, the functor $F\ A\ B = A + B \times B \times B \times B$, and the following functions. The `split` function splits an image into four sub-images with overlapping borders, as required for the kernel. The `combine` function concatenates the sub-images in the corresponding positions. The `kern` function applies a kernel to an image. Finally, the `finished` function tests whether an image has the desired

properties, in which case the computation terminates. We can represent image convolution on a list of input images as follows:

$$
\begin{aligned}
&\texttt{conv} \;:\; \texttt{Kern} \to (\mathsf{List}\ \texttt{Img} \overset{\sigma}{\mapsto} \mathsf{List}\ \texttt{Img}) \\
&\texttt{conv}\ k \;=\; \\
&\ \mathsf{map}_{\mathsf{List}}\ (\mathsf{iter}_{\texttt{Img}}\ (\texttt{finished?} \circ \texttt{hylo}_F\ (\texttt{combine} \circ F\ (\texttt{kern}\ k)) \\
&\qquad\qquad\qquad\qquad (\texttt{split}\ k)))
\end{aligned}
$$

The structure of `conv` is equivalent to a feedback loop, which exposes many opportunities for parallelism. Again, we assume a suitable cost model. Our estimates are given for 1000 images, of size 2048×2048.

$$
\begin{aligned}
\sigma \;&=\; \textsc{par}_L\ (\textsc{fb}\ (\textsc{dc}_{n,L,F}\ (\textsc{a} \circ F\ \textsc{a})\ \textsc{a} \parallel \_)) \\
&=\; \textsc{par}_L\ (\textsc{fb}\ (\ \textsc{farm}\ n\ \_ \ \parallel\ \_ \ \parallel \_)) \\
&=\; \min\ \texttt{cost}\ (\textsc{par}_L\ (\textsc{fb}\ (\_ \ \parallel\ \_))) \\
&=\; \ldots
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{cost}\ (\textsc{par}_L\ (\textsc{fb}\ (\textsc{a}_{c_1} \parallel \textsc{a}_{c_2}))\ sz \;= \\
&\qquad \sum_{1 \le i, |\textsc{a}_{c_1} \parallel \textsc{a}_{c_2}|^i sz > 0} \texttt{cost}\ (\textsc{a}_{c_1} \parallel \textsc{a}_{c_2})\ (|\textsc{a}_{c_1} \parallel \textsc{a}_{c_2}|^i sz) \\
&\quad =\; 20923.02ms \\
&\texttt{cost}\ (\textsc{par}_L\ (\textsc{fb}\ (\textsc{farm}_4\ (\textsc{fun}\ \textsc{a}_{c_1}) \parallel (\textsc{fun}\ \textsc{a}_{c_2}))))\ sz \\
&\quad =\; 6649.55ms \\
&\texttt{cost}\ (\textsc{par}_L\ (\textsc{fb}\ (\textsc{fun}\ \textsc{a}_{c_1} \parallel \textsc{farm}_1\ (\textsc{fun}\ \textsc{a}_{c_2}))))\ sz \\
&\quad =\; 20923.02ms \\
&\texttt{cost}\ (\textsc{par}_L\ (\textsc{fb}\ (\textsc{farm}_{14}\ (\textsc{fun}\ \textsc{a}_{c_1}) \parallel \textsc{farm}_4\ (\textsc{fun}\ \textsc{a}_{c_2}))))\ sz \\
&\quad =\; 2694.30ms \\
&\ldots
\end{aligned}
$$

Collectively, our examples have demonstrated the use of our techniques for all the parallel structures we have considered, showing that we can automatically introduce parallelism according to some required structure, while maintaining functional equivalence with the original form.

3.6 Discussion

This chapter introduced the core part of the *Structured Arrows (StA)* framework, with the *structure-annotated* arrows as the key part: $\overset{\sigma}{\mapsto}$. Although

a set of parallel constructs, i.e. the language P, was considered, any other set of algorithmic skeletons could have been considered, as long as their denotational semantics can be defined in terms of *hylomorphisms*. This would imply that a *parallelism erasure* rewriting could be done. The set of algorithmic skeletons considered in this chapter is therefore not a limitation.

The StA framework uses the first known type-based representation of the parallel structure of a program. The first-ever type-based approach to parallelism [XKCH03] focuses on detecting the parallelisability of functions, which are then transformed by using a calculational approach. However, they do not annotate their types with the specific parallel structures that can be used to parallelise a program. Previous uses of types for parallelism have focused on proving particular properties of the underlying program, e.g. *productivity* [PnS01, PnS05]. In StA, type-annotations are skeletal programs that omit most of the implementation details that are irrelevant for the parallelisation of the underlying function. When coupled with cost information, most of these structures can be inferred from the underlying structure in a sound way.

Finally, although we defined the StA framework to tackle the problem of writing parallel programs, we should note that it is general enough to investigate generic program optimisations and transformations. One example of this, outside the scope of the context of parallelism, would be to replace specific instances of catamorphisms, e.g. catamorphisms on lists of some known size, by efficient low-level computation structures, such as loops for traversing an array containing the elements of this list. This usage of the StA framework would be completely orthogonal to the aims and objectives of this thesis.

Chapter 4

Operational Semantics

Chapter 3 presented the StA framework, which provides a new type-and-effect system that allows programmers to annotate function types with the intended parallelisation of functions. However, the StA framework does not provide a way to reason about the *cost* of alternative parallelisations, or to define the operational behaviour of parallel structures. This chapter addresses these limitations by introducing a new operational semantics for algorithmic skeletons. The operational semantics is designed to serve two purposes: a) to reason about the correctness of algorithmic skeleton implementations; and, b) to reason statically about the performance of alternative parallelisations.

4.1 An Operational Semantics for Queues

The main objective of the operational semantics of algorithmic skeletons is that it must be *predictable*. By predictable, we mean that, under the necessary assumptions, it must be possible to estimate execution times of parallel programs using this operational semantics. The model that was chosen for this purpose was a queue-based model that consists on three primitives: a) `enqueue`, a thread-safe enqueue operation; b) `dequeue`, a thread-safe dequeue operation; and, c) `eval`, a primitive that evaluates a pure function on its inputs. The intuition behind the queue-based model is to represent parallel patterns as computations on independent tasks that are stored in thread-safe shared queues for communication. The idea is

to model a parallel program as a set of workers operating in parallel, and communicating through these intermediate buffers.

The skeleton semantics (Sec. 4.2) is built on a small-step trace-based operational semantics for the queues that are used to link these skeletons. Each step in the queue semantics will describe a state transition within a simple parallel process abstraction. We first give a number of definitions.

State A *state* comprises a tuple of three main structures, $\mathcal{W} \times \mathcal{Q} \times \mathcal{S}$:

- an environment of worker definitions, $\mathcal{W}$, which is a mapping from worker identifiers to *worker loop* definitions, i.e. to the code that is run by each worker of the parallel process;
- an environment of *worker states*, $\mathcal{S}$, which represents the instruction that is currently being executed by the worker; and,
- a *queue environment*, $\mathcal{Q}$, which represents the buffers that link the workers.

We will assume that the worker environment, $\mathcal{W}$, is fixed. The rules in our operational semantics therefore have the form:

$$\boxed{(\mathcal{Q}, \mathcal{S}) \xrightarrow{\alpha} (\mathcal{Q}', \mathcal{S}')}$$

They are given in terms of a labelled state transition system. The labels are the actions $\alpha ::= \mathtt{g}^w q \mid \mathtt{e}^w(f, x) \mid \mathtt{p}^w(x, q)$, which respectively *get* an input from a queue, *evaluate* a function f on an input x, or *put* a result on a queue. Actions are performed on specific queues q, and are tagged with the worker, w, that performs the action.

Queue Environments A queue environment is a mapping from queue identifiers to sequences of values.

$$\mathcal{Q} = \begin{bmatrix} q_0 & \mapsto \langle x_0,\ x_1, \ldots \rangle \\ \ldots & \\ q_n & \mapsto \langle \ldots \rangle \end{bmatrix}$$

Queue Operations We assume the usual thread-safe *enqueue* and *dequeue* operations.

$$\begin{array}{lcl} \texttt{enqueue}\,(\mathcal{Q}[q \mapsto vs], x, q) & \to & \mathcal{Q}[q \mapsto \langle x \mid vs \rangle] \\ \texttt{dequeue}\,(\mathcal{Q}[q \mapsto \langle vs \mid x \rangle], q) & \to & \mathcal{Q}[q \mapsto vs],\ x \end{array}$$

We overload the notation for *enqueue*/*dequeue* operations to also work on sums and products. Enqueuing a product type value into a "product queue" results in a pairwise *enqueue* on each sub-queue. Enqueuing a sum type value into a "sum queue" yields a single *enqueue* operation, on the corresponding queue. We use Q to refer to these *queue structures*, and q to refer to queue identifiers.

$$Q \quad ::= \quad q \mid q_1 \times \cdots \times q_n \mid q_1 + \cdots + q_n$$

We interpret the *enqueue* operation on products of queues as a sequence of simple *enqueue* operations:

$$\begin{array}{l} \texttt{enqueue}\,(\mathcal{Q}, q_1 \times \cdots \times q_n, (x_1, \ldots, x_n)) \\ \quad = \ \texttt{enqueue}\,(\texttt{enqueue}\,(\ldots \texttt{enqueue}\,(\mathcal{Q}, q_1, x_1) \ldots), q_n, x_n) \end{array}$$

An *enqueue* on a sum of queues is an *enqueue* on the corresponding queue. For example, if $0 \leq i \leq n$:

$$\texttt{enqueue}\,(\mathcal{Q}, q_1 + \cdots + q_n, \texttt{inj}_1\ x) = \texttt{enqueue}\,(\mathcal{Q}, q_i, x)$$

Dequeue operations work similarly. We interpret the *dequeue* operation on products of queues as a sequence of simple *dequeue* operations:

$$\texttt{dequeue}\,(\mathcal{Q}, q_1 \times \cdots \times q_n) = (\texttt{dequeue}\,(\mathcal{Q}, q_1), \ldots, \texttt{dequeue}\,(\mathcal{Q}, q_1))$$

A *dequeue* operation on a sum of queues dequeues an element from the first non-empty queue. For example, if $0 \leq i \leq n$, and for all j, $0 \leq j < i$, $\mathcal{Q}[q_j \mapsto \langle\rangle]$, and $\mathcal{Q}[q_i \mapsto \langle vs | x \rangle]$, then:

$$\texttt{dequeue}\,(\mathcal{Q}, q_1 + \cdots + q_n) = \texttt{inj}_i\ (\texttt{dequeue}\,(\mathcal{Q}, q_i))$$

This is an arbitrary choice. Any alternative ordering (e.g. round-robin) is equally acceptable. However, this decision would need to be re-considered carefully if a parallel structure uses dequeue operations on sums of queues. So far, none of our structures require such operations. In the operational semantics, each action will represent an *enqueue* or *dequeue* operation on a single queue. So dequeuing from a tuple of n queues will result in n *dequeue* actions.

Workers In each iteration, a worker first performs a sequence of *dequeues*, then performs its local computation, f, and finally performs a sequence of *enqueue* operations. We represent this as:

$$\mathcal{W} = \left[\begin{array}{c} \cdots \\ w \mapsto \texttt{worker}\,(Q_i,\ f,\ Q_o) \\ \cdots \end{array}\right]$$

where Q_i and Q_o are the input and output queue structures. The example below shows the state of a worker that is in the process of dequeuing from a tuple of queue identifiers, and that has already *dequeue*d n elements from the first n queues:

$$\mathcal{S} = \left[\begin{array}{c} \cdots \\ w \mapsto (x_1, \ldots, x_n, \texttt{dequeue}\,(q_{n+1}), \ldots, \texttt{dequeue}\,(q_m)) \\ \cdots \end{array}\right]$$

Generally, a worker state can be either a sequence (or sum) of *dequeue* or *enqueue* operations, or an *eval* operation. Let v be a value:

$$\begin{array}{lll} \texttt{st} & ::= \texttt{inSt} \mid \texttt{outSt} \mid \texttt{eval}\,(x) \\ \texttt{inSt} & ::= v \mid \texttt{dequeue}\,(q) \mid (\texttt{inSt}, \ldots, \texttt{inSt}) \\ \texttt{outSt} & ::= v \mid \texttt{enqueue}\,(q, x) \mid \texttt{outSt}; \ldots; \texttt{outSt} \end{array}$$

The definition of `st` represents worker states, where `inSt` is a worker in a state performing dequeue operations, `outSt` is the state of a worker performing enqueue operations, and $\texttt{eval}\,(x)$ is a worker performing a pure computation on x.

The transition rules for each worker in the queue-based operational semantics are given in Figure 4.1. Note that we overload *enqueue*/*dequeue* operations to also deal with sums and products. We also assume that *enqueues* on simple queues are trivial. The full operational semantics nondeterministically chooses any possible transition in any worker w in the worker environment:

$$\frac{(\mathcal{Q}, \texttt{st}) \xrightarrow{\alpha^w} (\mathcal{Q}', \texttt{st}')}{(\mathcal{Q}, \mathcal{S}[w \mapsto \texttt{st}]) \xrightarrow{\alpha^w} (\mathcal{Q}', \mathcal{S}[w \mapsto \texttt{st}'])}$$

$$\frac{\texttt{dequeue}\,(\mathcal{Q}, q_{n+1}) \to (\mathcal{Q}', v)}{(\mathcal{Q}, (v_1, \ldots, v_n, \texttt{dequeue}\,(q_{n+1}), \ldots)) \xrightarrow{\texttt{g}^w q_{n+1}} (\mathcal{Q}', (v_1, \ldots, v_n, v, \ldots))}$$

$$\frac{\texttt{dequeue}\,(\mathcal{Q}, q_{n+1}) \to (\mathcal{Q}', v)}{(\mathcal{Q}, (v_1, \ldots, v_n, \texttt{dequeue}\,(q_{n+1}))) \xrightarrow{\texttt{g}^w q_{n+1}} (\mathcal{Q}', \texttt{eval}\,(v_1, \ldots, v_n, v))}$$

$$\frac{\texttt{enqueue}\,(\mathcal{Q}, q_1, x_1) \to \mathcal{Q}'}{(\mathcal{Q}, \texttt{enqueue}\,(q_1, x_1)\,; \ldots) \xrightarrow{\texttt{p}^w q_1} (\mathcal{Q}', \ldots)}$$

$$\frac{\texttt{enqueue}\,(\mathcal{Q}, q, x) \to \mathcal{Q}' \qquad \mathcal{W}[w \mapsto \texttt{worker}\,(Q_i, f, Q_o)]}{(\mathcal{Q}, \texttt{enqueue}\,(q, x)) \xrightarrow{\texttt{p}^w q} (\mathcal{Q}', \texttt{dequeue}\,(Q_i, x))}$$

$$\frac{[\![f]\!](x) \;=\; y \qquad \mathcal{W}[w \mapsto \texttt{worker}\,(Q_i, f, Q_o)]}{(\mathcal{Q}, \texttt{eval}\,(x)) \xrightarrow{\texttt{e}^w(x)} (\mathcal{Q}, \texttt{enqueue}\,(Q_o, y))}$$

Figure 4.1: Transition Rules for queue operations.

4.1.1 Definition | Ready and Idle State. *A worker*

$$w \mapsto \texttt{worker}\,(Q_i, f, Q_o)$$

is in a ready *state if it is at the beginning of its worker loop. A worker is* idle *if it is ready and there are no inputs in its input queue structure* Q_i

4.1.2 Definition | Initial and Final State. *A parallel process is* initial *if all its workers are* ready *and all the queues except the input queue are empty. A parallel process is* final *if all its workers are* idle.

Parallel Process A parallel process is $\mathcal{P}$ is a program state that comprises a collection of workers and queues, where there are two distinct queues, an input and output queues, q_i and q_o, satisfying that for all sequence of inputs xs, final state $\mathcal{P}'$ and trace, if

$$\mathcal{P}[q_i \leftarrow xs] \xrightarrow{\alpha_1, \alpha_2, \cdots} \mathcal{P}'[q_o \mapsto ys],$$

then there exists a function f such that

$$\forall x, x \in xs \Leftrightarrow f\ x \in ys.$$

In other words, ys is equal to (a permutation of) $\texttt{map}_{\langle\rangle}\ f\ xs$. For simplicity, we write the following for *running a parallel process*:

$$\begin{aligned}[\![\mathcal{P}]\!] &= \lambda xs.ys \\ &\quad \text{where} \quad \mathcal{P}[q_i \leftarrow xs] \xrightarrow{\alpha_1,\alpha_2,\cdots} \mathcal{P}'[q_o \mapsto ys]\end{aligned}$$

Due to the non-determinism in the operational semantics, note that $[\![\mathcal{P}]\!]$ is not a function: for the same input, there are multiple possible outputs. However, since we require parallel processes to return a permutation of $map_{\langle\rangle}\ f$, for some f, then we impose the restriction that any trace in $[\![\mathcal{P}]\!]$ must write the results in the output queue in order. This allows us to formulate the key requirement for any newly defined parallel structure: a parallel process $\mathcal{P}$ is any program state such that $[\![\mathcal{P}]\!] = map_{\langle\rangle}\ f$, for some function f.

4.2 Queue-Based Skeleton Semantics

The queue-based language that we described in the previous section is powerful enough to describe the operational semantics of a number of common patterns of parallel computation. Recall the full syntax of parallel processes described in Section 3.2.3 on page 77.

$$\begin{aligned} e \in E &::= s \mid \mathsf{par}_T\ p \\ s \in S &::= f \mid e_1 \circ e_2 \mid \texttt{hylo}_F\ e_1\ e_2 \\ p \in P &::= \mathsf{fun}\ s \mid p_1 \parallel p_2 \mid \mathsf{dc}_{n,F}\ s_1\ s_2 \mid \mathsf{farm}\ n\ p \mid \mathsf{fb}\ p \end{aligned}$$

This section defines a translation from any $p \in P$ to the queue-based model described in the previous section. The translation is denoted by:

$$\texttt{runskel}_T\, p.$$

The skeletons in $p \in P$ work on arbitrary collections of data T, i.e. they take a collection of input tasks $T\ A$ and produce a collection of outputs $T\ B$. In order to define their operational semantics, we need to show how to process an arbitrary collection $T\ A$ in the queue-based model.

4.2.1 Streaming Arbitrary Tree-Like Types

We take an approach similar to the notion of *containers* [AAG03], a mathematical formalisation that separate the *contents* of a datatype from its *shape*. Note that sequences of values $\langle A \rangle$ can be interpreted as lists, defined as the fixpoint of the corresponding list base functor L, as defined in Chapter 2:

$$\begin{aligned} L\ A\ B &= 1 + A \times B \\ \langle A \rangle &= \mu(L\ A) \end{aligned}$$

Denotationally, the operations $\texttt{dequeue}\,(q)$ and $\texttt{enqueue}\,(x, q)$ can be defined in terms of the following functions:

$$\begin{aligned} &\texttt{enqueue}_L\ :\ A \times \langle A \rangle\ \to \langle A \rangle \\ &\texttt{enqueue}_L\ (x, l)\ =\ \texttt{cata}_L\ \texttt{addX} \\ &\quad \text{where } \texttt{addX}\ (\texttt{inj}_1\ ())\ =\ \langle x \rangle \\ &\qquad\qquad \texttt{addX}\ l\ \ =\ \texttt{in}_L\ l \end{aligned}$$

$$\begin{aligned} &\texttt{dequeue}_L\ :\ \langle A \rangle\ \to L\ A\ \langle A \rangle \\ &\texttt{dequeue}_L\ =\ \texttt{out}_L \end{aligned}$$

It should be clear that those functions describe the required queue operations, which can be used to implement the behaviour of the $\texttt{enqueue}\,(x, q)$ and $\texttt{dequeue}\,(q)$. This implies that these sequences of values, $\langle A \rangle$, can be treated as functors, with all the necessary properties.

$$\texttt{map}_{\langle\rangle}\ :\ (A \to B) \to \langle A \rangle \to \langle B \rangle$$

We assume that, for a collection of data T, there are two functions

$$\begin{aligned} &\texttt{stream}_T\ :\ T\ A \to\ T\ \mathbb{N} \times \langle A \rangle \\ &\texttt{unstream}_T\ :\ T\ \mathbb{N} \times \langle A \rangle \to T\ A \end{aligned}$$

that are inverses,

$$\texttt{stream}_T \circ \texttt{unstream}_T\ =\ \texttt{id} \tag{4.1}$$

$$\texttt{unstream}_T \circ \texttt{stream}_T\ =\ \texttt{id} \tag{4.2}$$

with the following properties:

$$\texttt{stream}_T \circ \texttt{map}_T\ =\ (\texttt{id} \times \texttt{map}_{\langle\rangle}) \circ \texttt{stream}_T \tag{4.3}$$

$$\texttt{unstream}_T \circ (\texttt{id} \times \texttt{map}_{\langle\rangle}) \;=\; \texttt{map}_T \circ \texttt{unstream}_T \tag{4.4}$$

Given any functor $T\ A = \mu(F\ A)$, then these two properties imply that a functor T',

$$T'\ A = T\ \mathbb{N} \times \langle A \rangle,$$

is isomorphic to $T\ A$, where $\texttt{stream}_T$ and $\texttt{unstream}_T$ are the natural transformations between T and T'. The main idea behind these definitions to *stream* the tasks in a collection T, while keeping its structure, $T\ \mathbb{N}$. The natural numbers in the structure $T\ \mathbb{N}$ are the indices of the values in $\langle A \rangle$. Note that if a particular traversal on $T\ A$ was assumed, e.g. in-order or post-order traversals, then it would suffice to keep a structure $T\ 1$. However, keeping the indices of the values makes it possible to get the tasks of $T\ A$ in any arbitrary order. Assuming the operations $\texttt{stream}_T$ and $\texttt{unstream}_T$ with the required properties, then it is clear how to implement a parallel process that works with any arbitrary collection of data T.

4.2.1 Lemma *For all parallel process* $p\ :\ T\ A \to T\ B$ *and function* $f : A \to B$*, if*

$$[\![\texttt{runskel}_T\ p]\!] = \texttt{map}_T f,$$

then

$$[\![\texttt{runskel}_T\ p]\!] \;=\; \texttt{unstream}_T \circ \texttt{id} \times [\![\texttt{runskel}_{\langle\rangle}\ p]\!] \circ \texttt{stream}_T$$

.

Proof We know that there is some f such that

$$[\![\texttt{runskel}_T\ p]\!] = \;\texttt{map}_T f \tag{4.5}$$

We want to show that

$$\texttt{unstream}_T \circ \texttt{id} \times [\![\texttt{runskel}_{\langle\rangle}\ p]\!] \circ \texttt{stream}_T \;=\; \texttt{map}_T f$$

By equational reasoning,

$$\begin{aligned}
&\texttt{unstream}_T \circ \texttt{id} \times [\![\texttt{runskel}_{\langle\rangle}\ p]\!] \circ \texttt{stream}_T \\
&= \texttt{unstream}_T \circ \texttt{id} \times \texttt{map}_{\langle\rangle} f \circ \texttt{stream}_T && \{\textit{by Eqn. 4.5}\} \\
&= \texttt{map}_T f \circ \texttt{unstream}_T \circ \texttt{stream}_T && \{\textit{by Eqn. 4.4}\} \\
&= \texttt{map}_T f && \{\textit{by Eqn. 4.2, and } \texttt{id} \textit{ cancels}\}
\end{aligned}$$

□

Lemma 4.2.1 reduces the correctness of arbitrary skeletons, to the correctness on *streaming* skeletons, i.e. it suffices to show that

$$[\![\texttt{runskel}_{\langle\rangle}\, p]\!] = \texttt{map}_{\langle\rangle} f.$$

In the rest of this chapter, skeletons are assumed to work on queues, since we have shown how to extend them to work on arbitrary tree-like container types.

Remark An implementation of $\texttt{unstream}_T$ and $\texttt{stream}_T$ can potentially optimise the structure $T\ \mathbb{N}$ in a number of ways. For example, a main thread may generate it while streaming the values contained inside to the parallel skeleton, and use this same value at the end of the parallel computation. If the order in which the results are produced is irrelevant, then it can even be omitted.

4.2.2 Queue-Based Parallel Structures

In this section, an operational semantics is defined for the parallel structures in Section 3.2 on page 72. Given any $p \in P$, the queue-based interpretation $\texttt{runskel}_{\langle\rangle}\, p$ is defined. The approach that we follow is to define, for each parallel construct, an equivalent combinator in the queue-based model that produces an equivalent empty parallel process in the queue-based model $\mathcal{P}$, i.e. $\texttt{runskel}_{\langle\rangle}\, p = \mathcal{P}$. The notion of soundness will be established in terms of $\texttt{runskel}_{\langle\rangle}$. For each newly defined parallel construct, $\mathcal{P}$ we require that $[\![\mathcal{P}]\!] = \texttt{map}_{\langle\rangle}\ f$, for some function f.

Parallel Composition of Processes The queue-based interpretation, $\texttt{runskel}_T\, p$ will be defined in terms of a single construct: the parallel composition of processes. We define the parallel composition of processes as the union of their queues and workers, redefining any clashing worker identifier if needed, i.e. a union of queues, and a disjoint union of workers. If two queues are joined, their corresponding elements are concatenated.

$$(\mathcal{W}_1, \mathcal{Q}_1, \mathcal{S}_1) \parallel (\mathcal{W}_2, \mathcal{Q}_2, \mathcal{S}_2) = (\mathcal{W}_1 \uplus \mathcal{W}_2, \mathcal{Q}_1 \cup \mathcal{Q}_2, \mathcal{S}_1 \uplus \mathcal{S}_2)$$

Worker We lift sequential functions into workers with input and output queue structures. The new worker is *ready*.

$$\begin{array}{l}\texttt{qfun}(f)(Q_0, Q_1) = \\ \quad \mathcal{W} = [w \mapsto \texttt{worker}\,(Q_0, f, Q_1)]\,, \mathcal{Q} = Q_0 \cup Q_1, \mathcal{S} = [w \mapsto \texttt{dequeue}\,(Q_0)]\end{array}$$

4.2.2 Lemma $[\![\texttt{qfun}(f)(q_i, q_o)]\!] = [\![\texttt{fun}\ f]\!]$

Proof Recall that $[\![\texttt{qfun}(f)(q_i, q_o)]\!]$ is notation for

$$\begin{array}{l}[\![\mathcal{P}]\!] = \lambda xs.ys \\ \qquad \text{where} \quad \mathcal{P}[q_i \leftarrow xs] \xrightarrow{\alpha_1, \alpha_2, \cdots} \mathcal{P}'[q_o \mapsto ys]\end{array}$$

For workers, since there is only a $\texttt{worker}\,(q_i, f, q_o)$, the rules of the operational semantics in Figure 4.1 only allow to produce one trace that results of repeatedly selecting the actions for dequeuing, performing local computation, enqueuing: $\texttt{g}^w q_i, \texttt{e}^w, \texttt{p}^w q_o$. This has as a consequence that the output queue of the corresponding final state will contain $map_{\langle\rangle}\ f\ xs$, for any sequence xs in the input queue. □

Task farm A task farm replicates a structure n times:

$$\texttt{qfarm}(n, \mathcal{P})(Q_0, Q_1) = \overbrace{\mathcal{P}(Q_0, Q_1) \parallel \ldots \parallel \mathcal{P}(Q_0, Q_1)}^{n \text{ times}}$$

4.2.3 Lemma *If*

$$[\![\mathcal{P}]\!](q_i, q_o) = \texttt{map}_{\langle\rangle}\ f,$$

then

$$[\![\texttt{qfarm}(n, \mathcal{P})(q_i, q_o)]\!] = \texttt{map}_{\langle\rangle}\ f.$$

Proof This proof, and the subsequent proofs assume that $\texttt{dequeue}\,()$ and $\texttt{enqueue}\,()$ are thread-safe. Note that $\texttt{qfarm}(n, \mathcal{P})(q_i, q_o)$ is the replication, in parallel, of $\mathcal{P}$ n times, sharing the input and output queues. We can manually "rename" the queues and workers in $\mathcal{P}$ to create the equivalent resulting structure:

$$\overbrace{\mathcal{P}_1(q_i, q_o) \parallel \ldots \parallel \mathcal{P}_n(q_i, q_o)}^{n \text{ times}}.$$

Since all queues in $\mathcal{P}_i$ and $\mathcal{P}_j$, for $i \neq j$ are distinct, with the exception of q_i and q_o, then any trace taking $\texttt{qfarm}(n, \mathcal{P})(q_i, q_o)[xs \leftarrow xs]$ as initial state can safely interleave all the actions of $\mathcal{P}_1 \cdots \mathcal{P}_n$, with the exception of the actions affecting the input and output queues. That means that each trace on $\mathcal{P}_i$ will produce $\texttt{map}_{\langle\rangle}\ f\ x_i$, where $xs = \mathsf{interleave}_1(x_1, \ldots, x_n)$. Therefore,

$$ys = \mathsf{interleave}_2(\texttt{map}_{\langle\rangle}\ f\ x_1, \ldots, map_{\langle\rangle}\ f\ x_n) = \texttt{map}_{\langle\rangle}\ f\ \mathsf{interleave}_2(x_1, \ldots, x_n).$$

Any trace producing the outputs in order will, therefore, produce

$$ys = \texttt{map}_{\langle\rangle}\ f\ xs. \qquad \square$$

Parallel pipeline A parallel pipeline is the parallel composition of two structures, linked by an intermediate queue. Let q be a fresh queue identifier:

$$\texttt{qpipe}(\mathcal{P}_1, \mathcal{P}_2)(Q_0, Q_1) = \mathcal{P}_1(Q_0, q) \parallel \mathcal{P}_2(q, Q_1)$$

4.2.4 Lemma *If*

$$[\![\mathcal{P}_1]\!](q_i, q) = \texttt{map}_{\langle\rangle}\ g,$$

and

$$[\![\mathcal{P}_2]\!](q, q_o) = \texttt{map}_{\langle\rangle}\ f,$$

then

$$[\![\texttt{qpipe}(\mathcal{P}_1, \mathcal{P}_2)(q_i, q_o)]\!] = \texttt{map}_{\langle\rangle}\ (f \circ g).$$

Proof Straightforward, since

$$[\![\mathcal{P}_1]\!](q_i, q) = \texttt{map}_{\langle\rangle}\ g \quad \text{and} \quad [\![\mathcal{P}_1]\!](q_i, q) = \texttt{map}_{\langle\rangle}\ g,$$

and we are assuming thread-safe queue operations. Since the only contention is in queue q, and any operations in q are thread-safe, q must respect the FIFO order. Therefore, $[\![\texttt{qpipe}(\mathcal{P}_1, \mathcal{P}_2)(q_i, q_o)]\!]$ must be

$$\texttt{map}_{\langle\rangle}\ f \circ \texttt{map}_{\langle\rangle}\ g = \texttt{map}_{\langle\rangle}\ (f \circ g). \qquad \square$$

Feedback loop A feedback loop is created by inspecting the tag in the output tasks, and placing them back in the input queue depending on the value of this tag:

$$\mathtt{qfb}(\mathcal{P})(Q_0, Q_1) = \mathcal{P}(Q_0, Q_0 + Q_1)$$

4.2.5 Lemma *If*

$$[\![\mathcal{P}(q_i, q_o)]\!] = \mathtt{map}_{\langle\rangle}\ f$$

and

$$f : A \to A + B,$$

then

$$[\![\mathtt{qfb}(\mathcal{P})(q_i, q_o)]\!] = \mathtt{map}_{\langle\rangle}\ (\mathsf{iter}_{(+B)}\ f).$$

Proof Since

$$\mathtt{qfb}(\mathcal{P})(q_i, q_o) = \mathcal{P}(q_i, q_i + q_o),$$

then

$$[\![\mathtt{qfb}(\mathcal{P})(q_i, q_o)]\!] = [\![\mathcal{P}(q_i, q_i + q_o)]\!].$$

We start the proof on a singleton queue:

$$[\![\mathcal{P}(q_i, q_i + q_o)]\!]\langle x\rangle.$$

Since $[\![\mathcal{P}(q_i, q_o)]\!] = \mathtt{map}_{\langle\rangle}\ f$, there will be an action $\mathtt{enqueue}\,(f\ x, q_i + q_o)$. If $f\ x = \mathtt{inj}_2\ y$, this will result in $\mathtt{enqueue}\,(y, q_o)$. If $f\ x = \mathtt{inj}_1\ y$, this will result in $\mathtt{enqueue}\,(y, q_i)$. An enqueue operation on the input queue will result again in an initial state, so this sequence of actions will continue until the result is of the form $\mathtt{inj}_2\ z$. This is exactly the semantics of $\mathsf{iter}_{+B}\ f$. For queues with more than one element, note that $[\![\mathcal{P}(q_i, q_o)]\!] = \mathtt{map}_{\langle\rangle}\ f$ implies that each input is treated independently by $\mathcal{P}$, and that changing q_o for $q_i + q_o$ will only affect the $\mathtt{enqueue}\,()$ operations. This implies that the result of applying

$$[\![\mathcal{P}(q_i, q_i + q_o)]\!]$$

will return (a permutation of)

$$\mathtt{map}_{\langle\rangle}\ (\mathsf{iter}_{(+B)}\ f).$$

□

Parallelising Arbitrary Hylomorphisms We describe now how to parallelise an arbitrary *hylomorphism* using a divide-and-conquer skeleton. We will first describe a series of simple semantics-preserving transformations for any hylomorphism. The idea is that if the *anamorphism* part of a hylomorphism needs to split an input value into at most n sub-values, then we will create $n + 1$ queues, the first of which will send the corresponding input to the "combine" worker, and the remaining n of which will send the subdivided inputs to the subsequent divide stages. If an input cannot be divided any further, then a synchronisation token, 1, will be sent. Given a functor F described as a combination of sums, products and constants,

$$F\ A\ =\ T,$$

where A is nested at most n times within a product in T, we define the functor D_F as

$$D_F\ B = (1 + T[1/A]) \times \overbrace{(B \times \cdots \times B)}^{n \text{ times}}.$$

If we can convert any hylomorphism to this new structure, then we can use its regular structure to create a regular "divide-and-conquer graph" with the following communication structure:

1. The *divide* worker will communicate a value of type $1 + T[1/A]$ to the corresponding *combine* worker, and values of type B to the subsequent *divide* workers.

   - A value of type $1 + T[1/A]$ contains the "non-recursive" part of T, plus a unit to indicate that the input could not be divided any further.
   - A *combine* worker can use a value of this type to decide how to recombine the n values of type B that has been received from the previous stages of the divide-and-conquer skeleton.

2. A *divide* worker takes an element of type $1 + A$. If it is 1, then it transmits $\texttt{inj}_1\ ()$ to all its output queues. This indicates to the subsequent workers that there is no more work to be done. If a divide worker receives a value of type A, it divides the input value, splits the

recursive and non-recursive parts of T, and sends the corresponding elements to the output queues.

3. A *combine* worker that receives an element of type 1 from the corresponding *divide* worker will discard all values received from the previous level, and send 1 to the *combine* worker of the next level. If it receives an element of type $T[1/A]$, it will need to recombine the corresponding value of type $F\ A$ from the inputs, and apply the *combine* function to it.

We need a way to change a hylomorphism from an F structure to D_F. This is using the following functions:

$$\mathsf{div}\ :\ 1 + F\ A \to D_F\ (1 + A) \qquad\qquad \mathsf{comb}\ :\ D_F\ (1 + A) \to 1 + F\ A.$$

The div function separates the occurrences of values of type A in some structure $F\ A$ into the corresponding $T[1/A]$ and n-tuple of $1 + A$, and the function comb recomposes a structure $F\ A$ from the structure of D_F. If div receives $\mathtt{inj}_1\ ()$, then it returns a $n+1$ tuple of $\mathtt{inj}_1\ ()$. The comb function returns $\mathtt{inj}_1\ ()$ if the first component of the tuple $D_F\ (1 + A)$ is $\mathtt{inj}_1\ ()$. Note that the types F and D_F are not isomorphic, since the function comb is *partial*. However, the following properties do hold:

$$\mathsf{comb} \circ \mathsf{div}\ =\ \mathtt{id}$$
$$\mathsf{div} \circ (\underline{1} + F\ f)\ =\ D_F\ (\underline{1} + f) \circ \mathsf{div}.$$

Using these properties, we can show that the following condition holds for any functor F:

$$\underline{1} + \mathtt{hylo}_F\ g\ h$$
$$=$$
$$\underline{1} + g\ \circ\ F\ (\mathtt{hylo}_F\ g\ h)\ \circ\ h$$
$$=$$
$$(\underline{1} + g)\ \circ\ (\underline{1} + F\ (\mathtt{hylo}_F\ g\ h))\ \circ\ (\underline{1} + h)$$
$$=$$
$$((\underline{1} + g) \circ \mathsf{comb} \circ \mathsf{div})\ \circ\ (\underline{1} + F\ (\mathtt{hylo}_F\ g\ h))\ \circ\ (\underline{1} + h)$$
$$=$$
$$((\underline{1} + g) \circ \mathsf{comb}) \circ\ D_F\ (\underline{1} + \mathtt{hylo}_F\ g\ h)\ \circ\ (\mathsf{div} \circ (\underline{1} + h))$$
$$=$$
$$\mathtt{hylo}_{D_F}\ ((\underline{1} + g) \circ \mathsf{comb})\ (\mathsf{div} \circ (\underline{1} + h))$$

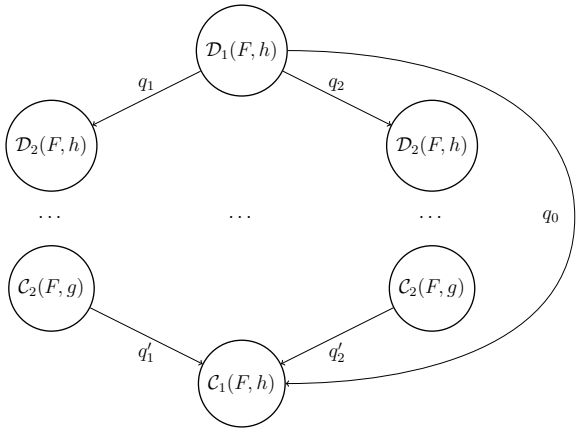


Figure 4.2: Divide-and-conquer skeleton: the q_i are the queues; circles represent the workers.

Using this, we can convert to a hylomorphism so that it has a regular, balanced call-tree. In this balanced call tree, values of unit type, (), are used for synchronisation. For all $\texttt{hylo}_F\ g\ h\ :\ A \to B$, given any $x\ :\ B$, then

$$\begin{gathered}
\underline{x}\triangledown\texttt{id} \circ (\texttt{hylo}_{D_F}\ ((\underline{1}+g)\circ\mathsf{comb})\ (\mathsf{div}\circ(\underline{1}+h)))\ \circ\ \texttt{inj}_2 \\
= \\
\underline{x}\triangledown\texttt{id} \circ (\underline{1}+\texttt{hylo}_F\ g\ h)\ \circ\ \texttt{inj}_2 \\
= \\
(\underline{x}\circ\underline{1})\triangledown(\texttt{id}\circ\texttt{hylo}_F\ g\ h)\ \circ\ \texttt{inj}_2 \\
= \\
\texttt{id}\circ\texttt{hylo}_F\ g\ h \\
= \\
\texttt{hylo}_F\ g\ h
\end{gathered}$$

This shows that the first level of a divide-and-conquer must wrap the input in $\texttt{inj}_2$, and the last level of a divide-and-conquer must unwrap the result using $\underline{x}\triangledown\texttt{id}$, for any arbitrary $x\ :\ B$. We define these as:

$$\begin{array}{lll}
\mathcal{D}_1(F,h) & = & \mathsf{div}\circ\texttt{inj}_2\circ h \qquad \mathcal{C}_2(F,g) \ = \ (\underline{1}+g)\circ\mathsf{comb} \\
\mathcal{D}_2(F,h) & = & \mathsf{div}\circ(\underline{1}+h) \quad \mathcal{C}_1(F,g) \ = \ (\underline{x}\triangledown\texttt{id})\circ(\underline{1}+g)\circ\mathsf{comb}
\end{array}$$

Although the structure D_F may appear to be complicated, it neatly fits our queue-based model, in that we can create queues that send/receive values

of types $(1 + T[1/A])$ and $1 + A$, and can create workers that split/combine values of these types, as shown in Figure 4.2. Values of type 1 are used to synchronise the different *divide* and *combine* workers, so that the different levels of a divide-and-conquer skeleton can operate on different inputs in parallel. We define this as:

$$\mathtt{qdc}(n, F, g, h)(Q_0, Q_1) = \mathtt{qdc'}(\mathtt{hylo}_F g\ h,\ n,\ F,\ g,\ h,\ \mathcal{C}_1,\ \mathcal{D}_1)(Q_0, Q_1)$$

$$\mathtt{qdc'}(f, 0, F, g, h, c, d)(Q_0, Q_1) = \mathtt{qfun}(f)(Q_0, Q_1)$$

$$\begin{aligned}
&\mathtt{qdc'}(f, n, F, g, h, c, d)(Q_0, Q_1) = \\
&\quad \mathtt{qfun}(d(F, h))(Q_0, q_0 \times q_1 \times \cdots \times q_n) \\
&\quad \parallel \mathtt{qdc'}(\underline{1} + \mathtt{hylo}_F g\ h, n-1, F, g, h, \mathcal{C}_2, \mathcal{D}_2)(q_1, q_1') \\
&\quad \parallel \cdots \\
&\quad \parallel \mathtt{qdc'}(\underline{1} + \mathtt{hylo}_F g\ h, n-1, F, g, h, \mathcal{C}_2, \mathcal{D}_2)(q_n, q_n') \\
&\quad \parallel \mathtt{qfun}(c(F, g))(q_0 \times q_1' \times \cdots \times q_n', Q_1)
\end{aligned}$$

4.2.6 Lemma

$$[\![\mathtt{qdc'}(\underline{1} + \mathtt{hylo}_F\ g\ h, n, F, g, h, \mathcal{C}_2, \mathcal{D}_2)(q_i, q_o)]\!] = \mathtt{map}_{\langle\rangle}\ (\underline{1} + \mathtt{hylo}_F\ g\ h)$$

Proof We follow an argument similar to `qfb`. It is easy to show that for each `dequeue` (q_i) there will be *exactly* one `enqueue` (q_o). This is proven by induction on n. The base case is defined by a `qfun`, for which it trivially holds. Assuming the induction hypothesis, it holds for `qdc'` applied to $n - 1$. For each `dequeue` (q_i), there is exactly one `enqueue` $(q_0 \times \cdots q_n)$, which in turn implies there is exactly one `enqueue` (q_0), ..., `enqueue` (q_n). By the induction hypothesis, we know that each `dequeue` (q_k) will be followed by exactly one `enqueue` (q_k'), except for q_0. Since there will be exactly one `enqueue` (q_0), and exactly one `enqueue` (q_k) for $1 \leq k \leq n$, we know that there will be exactly one `dequeue` $(q_0, q_1', \ldots, q_n')$. Since the last stage is another `qfun`, this implies that for each `dequeue` (q_i) there can be only one `enqueue` (q_o). This shows that $[\![\mathtt{qdc}(\underline{1} + \mathtt{hylo}_F\ g\ h, n, F, g, h, \mathcal{C}_2, \mathcal{D}_2)(q_i, q_o)]\!]$ must be some $\mathtt{map}_{\langle\rangle}\ f$. In order to show that this f corresponds to $\mathtt{hylo}_F\ g\ h$, we take a singleton queue $\langle x \rangle$, and reason by induction on n.

Case $n = 0$

$$[\![\mathtt{qdc'}(\underline{1} + \mathtt{hylo}_F\ g\ h, 0, F, g, h, \mathcal{C}_2, \mathcal{D}_2)(q_i, q_o)]\!] = [\![\mathtt{qfun}(\underline{1} + \mathtt{hylo}_F\ g\ h)(q_i, q_o)]\!].$$

This is the base case, which trivially holds, since $\texttt{qfun}(f)$ is equivalent to $\texttt{map}_{\langle\rangle}\ f$.

Case $n = m + 1$

$$\begin{aligned}
&\texttt{qdc}'(f, m+1, F, g, h, \mathcal{C}_2, \mathcal{D}_2)(Q_0, Q_1) = \\
&\quad \texttt{qfun}(d(F,h))(Q_0, q_0 \times q_1 \times \cdots \times q_n) \\
&\quad \parallel \texttt{qdc}'(\underline{1} + \texttt{hylo}_F g\ h, m, F, g, h, \mathcal{C}_2, \mathcal{D}_2)(q_1, q_1') \\
&\quad \parallel \cdots \\
&\quad \parallel \texttt{qdc}'(\underline{1} + \texttt{hylo}_F g\ h, m, F, g, h, \mathcal{C}_2, \mathcal{D}_2)(q_n, q_n') \\
&\quad \parallel \texttt{qfun}(c(F,g))(q_0 \times q_1' \times \cdots \times q_n', Q_1).
\end{aligned}$$

By the induction hypothesis, we know that this is equivalent to:

$$\begin{aligned}
&\texttt{qdc}'(f, m+1, F, g, h, \mathcal{C}_2, \mathcal{D}_2)(Q_0, Q_1) = \\
&\quad \texttt{qfun}(\mathcal{D}_2(F,h))(Q_0, q_0 \times q_1 \times \cdots \times q_n) \\
&\quad \parallel \texttt{qfun}(\underline{1} + \texttt{hylo}_F g\ h)(q_1, q_1') \\
&\quad \parallel \cdots \\
&\quad \parallel \texttt{qfun}(\underline{1} + \texttt{hylo}_F g\ h)(q_n, q_n') \\
&\quad \parallel \texttt{qfun}(\mathcal{C}_2(F,g))(q_0 \times q_1' \times \cdots \times q_n', Q_1).
\end{aligned}$$

By unfolding the definitions of $\mathcal{D}_2$ and $\mathcal{C}_2$, we can conclude that this is equivalent to:

$$(\underline{1} + g) \circ \mathsf{comb} \circ \texttt{id} \times (\underline{1} + \texttt{hylo}_F g\ h) \times \cdots \times (\underline{1} + \texttt{hylo}_F g\ h) \circ \mathsf{div} \circ (\underline{1} + h).$$

This corresponds to

$$(\underline{1} + g) \circ \mathsf{comb} \circ D_F\ (\underline{1} + \texttt{hylo}_F g\ h) \circ \mathsf{div} \circ (\underline{1} + h),$$

and we have show that this is the same as

$$\underline{1} + \texttt{hylo}_F\ g\ h.$$

□

4.2.1 Corollary

$$[\![\texttt{qdc}(n, F, g, h)(q_i, q_o)]\!] = \texttt{map}_{\langle\rangle}\ (\texttt{hylo}_F\ g\ h)$$

Proof Follows directly from Lemma 4.2.6, since the outermost level of `qdc` wraps the whole expression with $\mathcal{C}_1$ and $\mathcal{D}_1$, which turn $\overline{1} + \texttt{hylo}_F\ g\ h$ into $\texttt{hylo}_F\ g\ h$, as is required. □

Soundness The soundness of the operational semantics with respect to the denotational semantics can be reduced to correctly connecting the queues of the algorithmic skeletons, as has been shown in Lemmas 4.2.3, 4.2.4, 4.2.5 and Corollary 4.2.1.

4.3 Predicting Parallel Performance

We will now formally derive a set of cost equations from the operational semantics in a systematic way. These cost equations can then be used by our type-system to derive cost-models for the high-level parallel structures. This gives two main benefits: i) the cost models are sound w.r.t. the operational semantics *by construction*; and ii) this provides a way to *automatically* derive cost equations for newly defined parallel structures. In this chapter, we are mainly interested in predicting realistic *lower bounds* on the execution times of parallel programs. While the more usual *worst-case* predictions might be useful for safety reasons, they would not provide sufficient information to enable us to choose the best parallel implementation. Moreover, when "initialisation" and "finalisation" times are taken into account, the worst-case parallel execution time will generally be very similar to the sequential case, and so provide very little insight into parallel performance. The *amortised* average case timings that we use here are much more useful for predicting the actual parallel performance of the system.

4.3.1 Costs and Sizes

The sequential components of the algorithmic skeletons are given as suitably lifted functions. Before we can define their cost models, we must first explain how we derive cost models for hylomorphisms. Vasconcelos [Vas08] showed how to use sized types [HPS96] for developing an accurate space cost analysis. Sized types have also been used for estimating upper bounds of execution times of parallel functional programs [LH96]. Brady and Hammond [BH05] use dependent types to capture size indices of terms, predicates on sizes, and predicate transformers, which they use for program resource analysis. The success of using sized types for cost analysis motivated our approach. We exploit this previous research on *sized types* [Vas08], as

well as previous research on *cost equations* [San95]. A *cost equation* for a function $f : A \to B$ provides an estimate of the execution time of f given an input of a given size.

$$\texttt{cost}_f \ : \ \texttt{size}_A \to \texttt{time}$$

To obtain the inputs for these cost equations, we use a variant of the usual notion of *sized types* [HPS96]. The only difference is that, since we are interested in amortised costs, we use *average sizes* rather than upper or lower bounds. The notion of *average size* is dependent on each problem, so rather than defining a generic calculation, we require specific definitions for each function and *type constructor*. Suitable definitions include, e.g. the average depth of a tree, the average number of elements in a structure etc.

Type Constructors These are either constant types C, or recursive types defined as the fixpoint of some base functor μF.

Sizes and Size Constraints We reuse the idea of *stages* [BGR08]. Essentially, sizes are either size variables or sums of sizes, and size constraints are inequalities on sizes. We write $A^i \mid C$ for a sized type A with size i and constraint C.

Sized types We require the polynomial (bi-) functors to be annotated with size expressions for each alternative (given as a sum-type). For example, the base bifunctor of a binary tree can be annotated as follows:

$$F^{0 \vee 1 + i + j} \ A \ B \ = \ 1 + A \times B^i \times B^j$$

The size expression $0 \vee 1 + i + j$ states that functor F either has size 0, i.e. it contains no elements of type A; or it has size $1 + i + j$, i.e. it has one element of type A, plus the sizes of the elements of type B, one of size i and the other of size j. Note that there are alternative definitions for the size of a functor F, e.g: $F^{1 \vee \max(i,j)} \ A \ B \ = \ 1 + A \times B^i \times B^j$. In a more general sense, a sized-type is a type constructor that is annotated with size information, e.g. $Int^i, List^{j+k} \ A, \ldots$. The $\texttt{in}_F$ and $\texttt{out}_F$ functions for a sized base functor of

a recursive type must have the following type and constraints:

$$\begin{aligned}\texttt{in}_F &: F^j \mu F \to \mu^i F \mid i = j \\ \texttt{out}_F &: \mu^i F \to F^j \mu F \mid i = j\end{aligned}$$

That is, we require that the sizes of the base functor represent the same information as the sizes of the recursive type. Finally, we will use $|A|$ to denote all the nested size information and size constraints in a type A. This is useful for defining cost models that require access to nested size information.

Deriving Recurrences from Hylomorphisms We now show how we use sized types to derive recurrences. Essentially, we will define a *cost equation* for each syntactic construct, that takes some cost equation parameters and produces another cost equation. Although the cost of a function is not, in general, compositional, since it may depend on previous computations as well as on other properties of the input data, we will here take a compositional approach under the assumption that only the sizes will affect the execution times. This is a valid way to obtain *amortised* costs.

The cost equations for each primitive operation are assumed to be a constant value that depends on the target architecture. This must be provided as a parameter. The cost equations of atomic functions must also be provided as a parameter. The cost of a function composition is the sum of the run-times of each stage, plus some architecture-dependent overhead.

For recursive functions that are defined as hylomorphisms, we generate recurrence relations, as with Barbosa *et al.* [BCP05]. In a similar sense to Sands' work [San95], the cost equations that are explained in this section can be thought of as *functions* that we are integrating into the type-system. For example, assuming that we have the following sized-type for *quicksort*:

$$\begin{aligned}&\texttt{merge} : T^{0 \vee 1+i+j} A\ (\mathsf{List}\ A) \to \mathsf{List}^k\ A \mid k = 0 \vee k = i + 1 + j \\ &\texttt{split} : \mathsf{List}^n\ A \to T^{0 \vee 1+r+s} A\ (\mathsf{List}\ A) \mid n = 0 \vee n = 1 + r + s\end{aligned}$$

$$\begin{aligned}&\texttt{qs} : \mathsf{List}^i\ A \to \mathsf{List}^j\ A \mid i = j \\ &\texttt{qs} = \texttt{hylo}_T\ \texttt{merge}\ \texttt{split}\end{aligned}$$

Given suitable cost equations for *split* and *merge*, $\texttt{cost}_{\texttt{split}}$, $\texttt{cost}_{\texttt{merge}}$, then there are two cases. Either $n = 0$, in which case $t = 0$, so we assume some

time $\texttt{cost}_{\texttt{split}}(0) + \texttt{cost}_{\texttt{merge}}(0)$, or $n = 1 + r + s$, in which case:

$$\begin{aligned}\texttt{cost}_{\texttt{qs}} \; &= \texttt{cost}_{\texttt{split}}(1 + r + s) + \texttt{cost}_{\texttt{merge}}(r + 1 + s) + \\ &\quad \texttt{cost}_{\texttt{qs}}(r) + \texttt{cost}_{\texttt{qs}}(s)\end{aligned}$$

To complete the cost equation, we need now to relate r and s with the input size n. By assuming that these sizes correspond to a balanced computation, we can then automatically calculate an *amortised* cost. Taking more extreme cases into account would, of course, require further programmer input.

$$\begin{aligned}&\texttt{cost}_{\texttt{qs}}(n) \; = \\ &\quad \texttt{cost}_{\texttt{split}}(1 + r + s) + \texttt{cost}_{\texttt{merge}}(r + 1 + s) + \texttt{cost}_{\texttt{qs}}(r) + \texttt{cost}_{\texttt{qs}}(s) \\ &\quad \text{where } r = (n-1)/2 \text{ and } s = (n-1)/2\end{aligned}$$

We simplify this internally to generate the desired recurrence relation:

$$\begin{aligned}\texttt{cost}_{\texttt{qs}}(0) \; &= \texttt{cost}_{\texttt{split}}(0) + \texttt{cost}_{\texttt{merge}}(0) \\ \texttt{cost}_{\texttt{qs}}(n) \; &= \texttt{cost}_{\texttt{split}}(n) + \texttt{cost}_{\texttt{merge}}(n) + 2 \times \texttt{cost}_{\texttt{qs}}((n-1)/2)\end{aligned}$$

4.3.2 Costing Traces of Parallel Processes

We now describe a systematic way to derive cost equations. We start with a structure C, with some initial empty $\mathcal{P}_i$ and cost c_i. Taking suitably sound simplifications and approximations of $\texttt{time}(\alpha_1, \ldots)$, we then derive an "amortised cost equation" $\texttt{cost}_{\texttt{C}}$ such that for all input l with sized-type $\langle A \rangle^i$ and trace,

$$C(\mathcal{P}_1, \ldots, \mathcal{P}_n)(q_{in} \; \mapsto l, q_{out} \mapsto \langle\rangle) \xrightarrow{\alpha_1, \ldots} C(\mathcal{P}_1, \ldots, \mathcal{P}_n)(q_{in} \; \mapsto \langle\rangle, q_{out} \mapsto l'),$$

then

$$i \times \texttt{cost}_{\texttt{C}}(c_1, \ldots, c_n) \; |A| \; \approx \; \texttt{time}(\alpha_1, \ldots).$$

We differentiate three phases in the execution of a parallel process: an *initialisation phase*, a *steady state*, and a final *flushing phase*. For example, a pipeline of two atomic functions, $w_1 \parallel w_2$, reaches a *steady state* after executing the *initialisation phase* of w_1. At this point, w_2 can run in parallel

with w_1:

$$\left(\begin{bmatrix} q_0 & \mapsto \langle x_1,\ x_2,\ \ldots\rangle, \\ q_1 & \mapsto \langle\rangle, \\ q_2 & \mapsto \langle\rangle \end{bmatrix}, \begin{bmatrix} w_1 & \mapsto \texttt{worker}\,(q_0, f, q_1) \\ w_2 & \mapsto \texttt{worker}\,(q_1, g, q_2) \end{bmatrix}\right)$$

$$\xrightarrow{\mathtt{g}^{w_1} q_0,\ \mathtt{e}^{w_1} x_1,\ \mathtt{p}^{w_1} q_1}$$

$$\left(\begin{bmatrix} q_0 & \mapsto \langle x_2,\ \ldots & \rangle, \\ q_1 & \mapsto \langle f\ x_1\rangle, & \\ q_2 & \mapsto \langle\rangle & \end{bmatrix}, \begin{bmatrix} w_1 & \mapsto \texttt{worker}\,(q_0, f, q_1) \\ w_2 & \mapsto \texttt{worker}\,(q_1, g, q_2) \end{bmatrix}\right)$$

We capture these ideas in the definitions below.

4.3.1 Definition | Steady State. *A parallel process $\mathcal{P}$ is in a steady state,* $\texttt{steady}(\mathcal{P})$*, if for all $w \in \mathcal{P}$,* $\neg\texttt{idle}(w)$*.*

4.3.2 Definition | Initialisation Phase. *The initialisation phase for a parallel process* $\texttt{initial}(\mathcal{P}_1)$ *is the shortest sequence of $a_1, a_2, \ldots, a_n$, such that if $\mathcal{P}_1 \xrightarrow{a_1, a_2, \ldots, a_n} \mathcal{P}_2$, then* $\texttt{steady}(\mathcal{P})$*.*

4.3.3 Definition | Flushing Phase. *The flushing phase for a parallel process $\mathcal{P}_1$ is the sequence of $a_1, a_2, \ldots, a_n$, such that for all $i \in [1 \ldots n]$ and for all $w \in \mathcal{P}_1$, $a_i \neq \texttt{dequeue}\,(q_{in}\,)^w$, and if $\mathcal{P}_1 \xrightarrow{a_1, a_2, \ldots, a_n} \mathcal{P}_2$, then* $\texttt{final}(\mathcal{P}_2)$*.*

Total Cost and Amortised Cost: Given some initial $\mathcal{P}$ and final $\mathcal{P}'$, where $\mathcal{P} \xrightarrow{\alpha_1, \ldots, \alpha_n} \mathcal{P}'$, the total cost of a parallel computation is

$$\texttt{time}(\alpha_1, \ldots, \alpha_n).$$

The cost of each action depends on the worker environment. We calculate the queue contention on each queue by counting the number of workers in which a queue appears in $\mathcal{W}$, similarly to [HBS16]. Then, the cost of the $\mathtt{g}^w$ and $\mathtt{p}^w$ operations is adjusted according to the corresponding overhead. If we split the trace into $\mathcal{P} \xrightarrow{\texttt{init}} \mathcal{P}_1 \xrightarrow{\texttt{steady}} \mathcal{P}_2 \xrightarrow{\texttt{flush}} \mathcal{P}'$, where $\texttt{init} = \alpha_1, \ldots, \alpha_i$, $\texttt{steady} = \alpha_i, \ldots, \alpha_j$, $\texttt{flush} = \alpha_j, \ldots, \alpha_n$ such that $\texttt{initial}(\mathcal{P})$, $\texttt{steady}(\mathcal{P}_1)$, and $\texttt{final}(\mathcal{P}')$ then the total time is:

$$\texttt{time}(\alpha_1, \ldots, \alpha_n) = \texttt{time}(\texttt{init}) + \texttt{time}(\texttt{steady}) + \texttt{time}(\texttt{flush})$$

$$
\begin{aligned}
&\texttt{cost}(\textsc{fun}\ \sigma^{n,m}) && = T_{\text{enqueue}}(n) + \texttt{cost}\, i + T_{\text{dequeue}}(m) \\
&\quad \text{where } |\sigma| = i \to j \mid C && \\
&\texttt{cost}(\textsc{farm}\ n\ \sigma^{i,o}) && = \frac{\texttt{cost}(\sigma^{i \times n, o \times n})}{n} \\
&\texttt{cost}(\sigma_1^{i,j} \parallel \sigma_2^{k,l}) && = \max\left\{\texttt{cost}(\sigma_1^{i,j+k}), \texttt{cost}(\sigma_2^{j+k,l})\right\} \\
&\texttt{cost}(\textsc{fb}\ \sigma^{n,m}) && = \texttt{if}\ (|\sigma| = 0) \\
&&& \quad \texttt{then cost}(\sigma^{n+m,m}) \\
&&& \quad \texttt{else cost}(\sigma^{n+m,m}) \\
&&& \qquad + \texttt{cost}(\textsc{fb}\ (\texttt{resize}(\sigma^{n,m}, |\sigma|))) \\
&\texttt{cost}(\textsc{dc}_{n,F}\ \sigma_1\ \sigma_2) && = \max\left\{\begin{array}{l} \texttt{cost}(\mathcal{D}(F, \sigma_2)), \\ \texttt{cost}(\textsc{dc}_{n-1,F}\ \sigma_1\ \sigma_2), \\ \texttt{cost}(\mathcal{C}(F, \sigma_1)) \end{array}\right\}
\end{aligned}
$$

Figure 4.3: Cost equations.

We can systematically derive cost models for `time(init)` and `time(flush)` using the method shown below.

4.3.3 Deriving Cost Equations from the Operational Semantics

We now show how to systematically derive the cost equations in Figure 4.3 from the operational semantics for skeletons. We base our approach on symbolic execution. The following assumptions are used to automatically derive amortised cost equations for parallel structures.

1. The queues contain enough elements for each *dequeue* to succeed.
2. Each time an element is removed from a queue, we will return a size that is extracted from its type.
3. Evaluating a function on a size immediately returns the size of the output type of that function, plus the set of constraints that relate the input and output sizes.
4. An *enqueue* operation always succeeds, and has no effect on the queues.

5. Any substructure (e.g. a farm worker) produces a trace that can be safely interleaved with the trace that is produced by other substructures (i.e. without modifying the result of the overall computation), and that has a known cost.

Note that assumption 1 holds only for a `steady` structure, so these cost equations would not be useful for estimating run-times of a computation where `time(init)` and/or `time(flush)` dominate.

Worker cost A worker, w, computing $f \ : \ A^i \to B^j \mid C$ in environment $\mathcal{W}[w \mapsto \texttt{worker}\,(q_i, f, q_o)]$ produces the "symbolic trace":

$$\mathtt{g}^w q_i,\ \mathtt{e}^w\ |A^i|,\ \mathtt{p}^w q_o.$$

Assuming there exists some trace $\alpha_1, \ldots, \alpha_n$ that can be safely interleaved with this trace, we want to know the cost of the actions in $\alpha_1, \ldots, \mathtt{g}^w q_i, \ldots, \mathtt{e}^w\ i, \ldots, \mathtt{p}^w q_o, \ldots, \alpha_n$. Here, the cost of $\mathtt{g}^w q_i$ will depend on any other action that is happening in parallel. The only actions that can affect the cost of a queue operation are other queue operations that are acting on the same queue. If n is the number of workers $w_{\text{get}} \in \mathcal{W}$ such that $w_{\text{get}} \neq w_i$ and the number of workers operating on q_o is m, then if we assume that the cost of the *enqueue* and *dequeue* operations is as described by [HBS16], then the cost is:

$$\texttt{time}(\mathtt{g}^w q_i,\ \mathtt{e}^w\ i,\ \mathtt{p}^w q_o)\ =\ T_{\text{enqueue}}(n) + \texttt{cost}_f(i) + T_{\text{dequeue}}(m)$$

We associate this cost with the corresponding high-level structure, so that it can be used by the type-checking algorithm:

$$\texttt{cost}(\textsc{fun}\sigma)\ =\ T_{\text{enqueue}}(n) + \texttt{cost}(\sigma) + T_{\text{dequeue}}(m)$$

The parameters n, m and $\texttt{cost}(\sigma)$ can be instantiated from the context, and added to the structure σ by the type-checking algorithm. We write $\sigma^{n,m}$ for a structure with n contending workers in the input queue, and with m contending workers in the output queue. Note that the real cost, understood as an equation from an input size to an execution time, would be $\texttt{cost}(\textsc{fun}\sigma^{n,m})(T^i\ j)\ =\ i \times (T_{\text{enqueue}}(n) + \texttt{cost}_\sigma(j) + T_{\text{dequeue}}(m))$, if we assume that T contains i elements of size j. Since we can annotate

structures, σ, with sizes, we can abuse our notation for extracting sizes of types, $|\sigma|$, and "overload" the meaning of `cost` to take a structure and yield an amortised cost.

Farm cost A farm, $\mathcal{C}(Q_0, Q_1) \parallel \cdots \parallel \mathcal{C}(Q_0, Q_1)$, consists of a number of parallel processes that are joined using the parallel composition operator, and that share input and output queues. Since we symbolically evaluate the *enqueue* and *dequeue* operations, we can take n arbitrary traces, one for each farm worker:

$$\mathcal{C}(Q_0, Q_1) \xrightarrow{\alpha_1^1, \alpha_2^1, \ldots} \mathcal{P}', \quad \ldots, \quad \mathcal{C}(Q_0, Q_1) \xrightarrow{\alpha_1^n, \alpha_2^n, \ldots} \mathcal{P}'$$

Each trace has cost $c_i, \ldots, c_n$. To obtain amortised costs, we assume that each of these values is equal to some average cost c. Because we assume that these actions can be safely interleaved, and because the cost of the workers already considers the queue contention, we only need to calculate the maximum cost of any worker, assuming that each sub-trace can be performed in parallel:

$$\texttt{time}(\alpha_a^i, \alpha_b^j, \ldots) \;=\; \max \left\{ \begin{array}{l} \texttt{time}(\alpha_1^1, \alpha_2^1, \ldots), \\ \texttt{time}(\ldots), \\ \texttt{time}(\alpha_1^n, \alpha_2^n, \ldots) \end{array} \right\} \;=\; \max\{c_1, \ldots, c_n\} \;=\; c$$

Note that if each $\mathcal{P}$ produces k outputs, then $\texttt{qfarm}(n, \mathcal{P})$ produces $k \times n$ outputs. In order to obtain an amortised cost from costing such traces, we need to divide the total cost by n.

$$\texttt{cost}(\textsc{farm}\ n\ \sigma) \;=\; \frac{\texttt{cost}(\sigma)}{n}$$

Note that, although this cost is clearly correct, the main point is that it was *systematically* derived from the simple queue-based model. It is thus *sound by construction*, and so no longer needs to be a parameter of the type system.

Parallel pipeline A pipeline, $\mathcal{C}_1(Q_0, q) \parallel \mathcal{C}_2(q, Q_1)$, consists of two parallel processes that are joined using the parallel composition operator, and

connected using an intermediate queue q. Again, we take a similar reasoning process:

$$\mathcal{C}_1(Q_0, q) \xrightarrow{\alpha_1^1, \alpha_2^1, \ldots} \mathcal{P}_1 \qquad \mathcal{C}_2(q, Q_1) \xrightarrow{\alpha_1^2, \alpha_2^2, \ldots} \mathcal{P}_2$$

Assume costs c_1,and c_2 for each process:

$$\texttt{time}(\alpha_a^i, \alpha_b^j, \ldots) \;=\; \max \left\{ \begin{array}{l} \texttt{time}(\alpha_1^1, \alpha_2^1, \ldots), \\ \texttt{time}(\alpha_1^2, \alpha_2^2, \ldots) \end{array} \right\} \;=\; \max \{c_1, c_2\}$$

Note that, in order to accurately lift these costs to the high-level structures, we need to consider the size of the output of α_i^1, not the size of the α_j^2 input. We do this by writing $\texttt{resize}(\sigma_2, |\sigma_1|)$, meaning that the input of σ_2 is altered to have the size of the output size in $|\sigma_1|$. We can do this safely since the type-checking algorithm can ensure that sizes meet the necessary constraints. Finally, we associate the cost with the corresponding high-level structure:

$$\texttt{cost}(\sigma_1 \parallel \sigma_2) \;=\; \max \{\texttt{cost}(\sigma_1), \texttt{cost}(\texttt{resize}(\sigma_2, |\sigma_1|))\}$$

Feedback loop A feedback loop requires us to take into consideration that an element may be written back to the input queue, $\mathcal{C}_1(Q_0, Q_0 + Q_1)$. The structure C must compute some function f, and we require it to have type:

$$F^{n \vee 0}\ A \;=\; A^n + B, \quad \text{s.t.} \quad f : A^i \to F\ A^j \mid j < i.$$

Since we require j to be strictly smaller than i, we can estimate the number of steps that are required until $i = 0$ to be n, *i.e.* the average number of times an element will need to be put back into the input queue. Basically, a trace $\mathcal{C}_1(Q_0, Q_0 + Q_1) \xrightarrow{\alpha_1^1, \alpha_2^1, \ldots} \mathcal{P}_1$ will need to be taken n times to ensure that, on average, at least one element is enqueued into output queue Q_1.

$$\mathcal{C}_1(Q_0, Q_0 + Q_1) \xrightarrow{\alpha_1^1, \alpha_2^1, \ldots} \mathcal{P}_1 \xrightarrow{\ldots} \mathcal{P}_n$$

Since this parameter can be estimated from the sized types, we can again assume that a structure σ is parameterised on it, and can use this in the high-level cost models. Again, we need to use `resize` to generate the appropriate cost equation:

$$\begin{array}{lcll} \texttt{cost}(\textsc{fb}\ \sigma) & = & \texttt{if} & (|\sigma| = 0) \\ & & & \texttt{then}\ \texttt{cost}(\sigma) \\ & & & \texttt{else}\ \texttt{cost}(\sigma)\ +\ \texttt{cost}(\textsc{fb}\ (\texttt{resize}(\sigma, \sigma))) \end{array}$$

Divide-and-Conquer The divide-and-conquer skeleton requires a little more work, since we first need to transform the structure so that it matches the skeleton. Since this transformation can be done in a fairly standard way, we initially focus on the cost of the divide-and-conquer skeleton:

$$\begin{array}{lcl}
\texttt{qdc}'(0,\ F,\ g,\ h)(Q_0, Q_1) & = & \texttt{qfun}(\underline{1} + \texttt{hylo}_F g\ h)(Q_0, Q_1) \\
\texttt{qdc}'(n,\ F,\ g,\ h)(Q_0, Q_1) & = & \texttt{qfun}(\mathcal{D}_2(F,h))(Q_0, q_0 \times q_1 \times \cdots \times q_m) \\
& & \parallel \texttt{qdc}'(n-1, F, g, h)(q_1, q_1') \\
& & \parallel \ldots \\
& & \parallel \texttt{qdc}'(n-1, F, g, h)(q_m, q_m') \\
& & \parallel \texttt{qfun}(\mathcal{C}_2(F,g))(q_0 \times q_1' \times \cdots \times q_m', Q_1)
\end{array}$$

Since we define this skeleton inductively for some "depth" n, we start with the base case (depth 0), which is equivalent to the cost of an atomic function:

$$\texttt{cost}(\textsc{dc}_{0,F}\ \sigma_1\ \sigma_2) \ = \ \texttt{cost}(\textsc{fun}\ (\textsc{hylo}_F\ \sigma_1\ \sigma_2))$$

For the recursive case, we assume $2 + m$ traces, one for each "recursive" case, plus the trace of the divide and the trace of the combine parts. Again, we assume that the traces can be safely interleaved, so we can calculate the cost of the total trace as:

$$\texttt{time}(\alpha_1, \ldots) = \max \left\{ \begin{array}{l} \texttt{time}(\alpha^{\text{div}}, \ldots), \\ \texttt{time}(\alpha^{\text{qdc}}, \ldots), \\ \texttt{time}(\ldots), \\ \texttt{time}(\alpha^{\text{qdc}}, \ldots), \\ \texttt{time}(\alpha^{\text{conq}}, \ldots) \end{array} \right\} = \max \left\{ \begin{array}{l} \texttt{time}(\alpha^{\text{div}}, \ldots), \\ \texttt{time}(\alpha^{\text{qdc}}, \ldots), \\ \texttt{time}(\alpha^{\text{conq}}, \ldots) \end{array} \right\}$$

We can then associate this cost equation with the costs of the high-level structures by substituting the cost of the relevant trace. Note that the elements in the queues decrease in size for each level that we descend into the divide-and-conquer structure. Assuming that the structures are annotated with sizes, we need to update the sizes accordingly in the recursive call to the cost of a divide-and-conquer structure, and in the combine part. Although we can once again use `resize`, we assume that the actual sizes have been

correctly updated in the DC structure:

$$\texttt{cost}(\text{DC}_{n,F}\ \sigma_1\ \sigma_2) \ = \ \max \left\{ \begin{array}{l} \texttt{cost}(\mathcal{D}(F, \sigma_2)), \\ \texttt{cost}(\text{DC}_{n-1,F}\ \sigma_1\ \sigma_2), \\ \texttt{cost}(\mathcal{C}(F, \sigma_1)) \end{array} \right\}$$

4.4 Real vs. Predicted Speedups

In order to validate our results, we have run a number of examples to compare the actual speedups against those that are predicted. In Section 4.3.3 we have shown that most of the cost equations can be derived in a systematic way from the operational semantics. These cost equations provide an estimation of the impact of the parallel overhead on the run-times. This section shows how this overhead affects the real run-times, and compare it with the predicted speedups for some of our parallel structures. We use two different real multicore machines: *titanic*, a 800MHz 24-core, AMD Opteron 6176, running Centos Linux 2.6.18-274.e15; and *lovelace*, a 1.4GHz 64-core, AMD Opteron 6376, running GNU/Linux 2.6.32-279.22.1.e16. All speedups shown here were calculated as the mean of ten executions.

Matrix Multiplication

Figure 4.4 shows the real *vs.* predicted speedups for task farms, using a simple matrix multiplication. The matrix multiplication implementation is a divide-and-conquer, and the parallel structure used exploits the *reforestation* of the matrix multiplication algorithm. The input pair of matrices is subdivided until they are smaller than $N/24 \times N/24$. These small matrices are then passed to a task farm, which multiplies them in parallel, and the results are finally recombined. We can see in Figure 4.4 on page 135 that the cost models accurately predict a lower bound on the speedups.

Image Merge

Figure 4.5 on page 136 shows the speedups for the image merge example, parallelised as a farm of a pipeline. Since the worker of the task farm is a

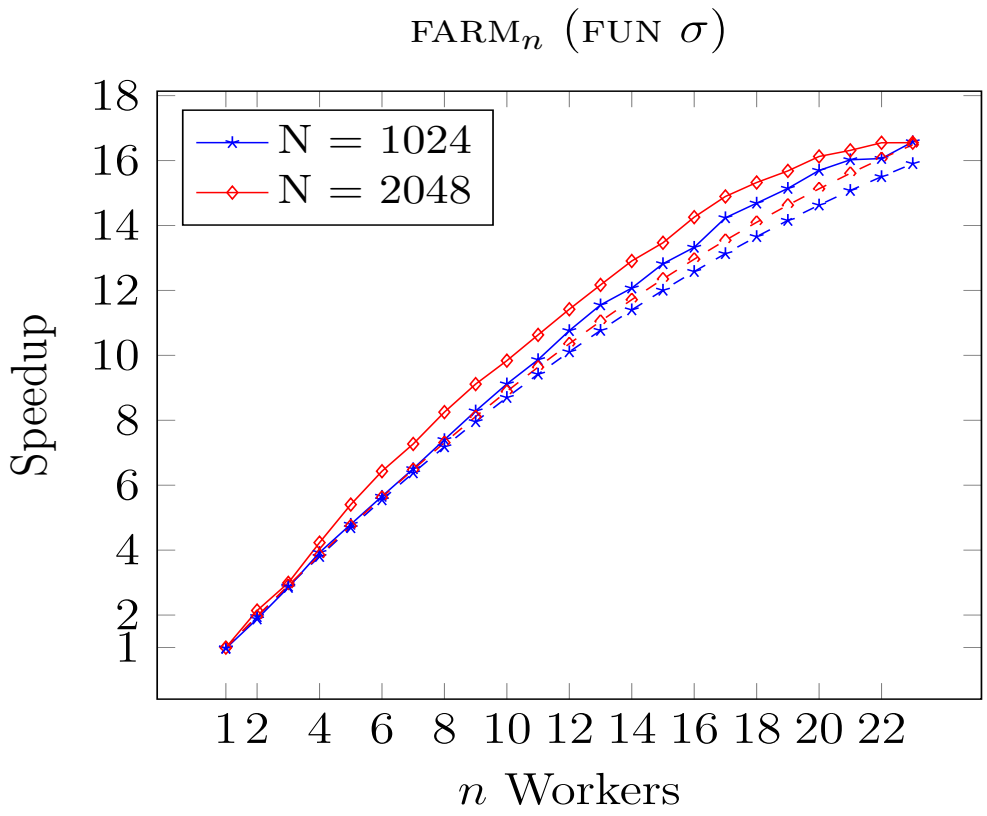


Figure 4.4: Speedup (solid lines) vs prediction (dashed lines). Matrix Multiplication of matrices of sizes N×N (*titanic*).

parallel pipeline, we use a maximum of 12 workers to use the 24 cores of *titanic*. Note that overhead of adding more workers greatly increases with this parallel structure. This is due to the small cost of the computation done at each of the pipeline stages.

Image Convolution

Figure 4.6 on page 137 shows the speedups for the image convolution example. This was originally described as a similar structured expression to image merge, as the composition of two functions, `read` and `process`, but the cost models predict a different optimal parallel structure, a pipeline of two farms. Figure 4.6 shows the real vs. predicted speedups of this parallel structure, with different number of workers in each of the pipeline stages. Note that the cost models (dashed) accurately predict a tight lower bound on the speedups.

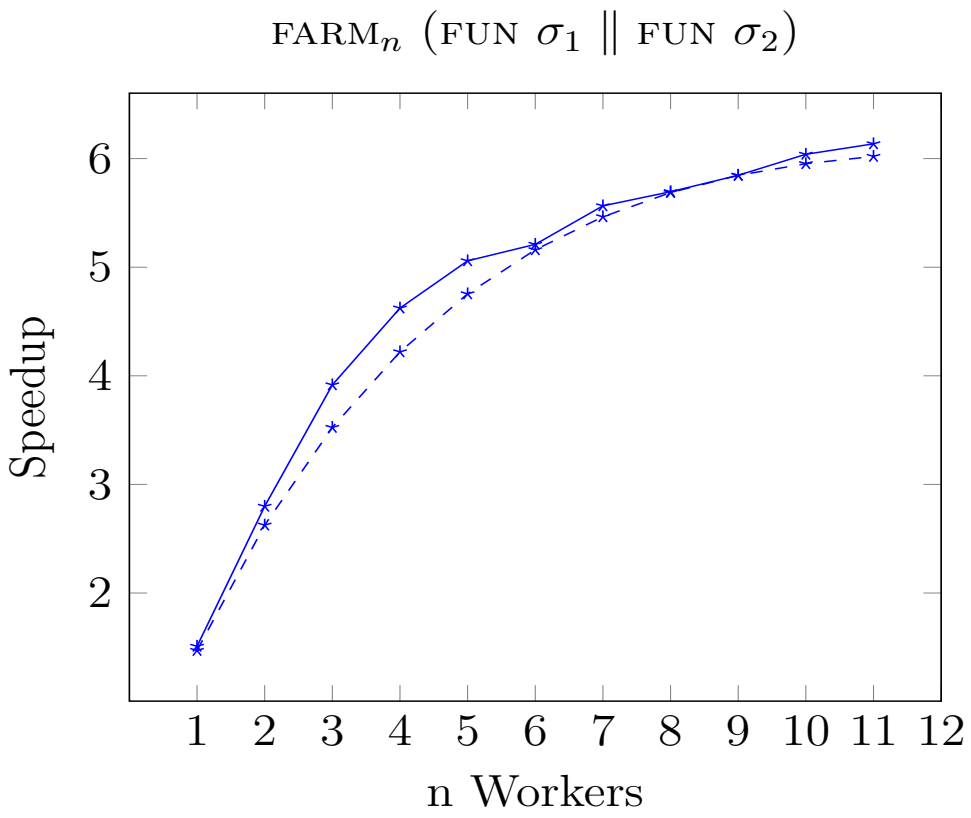


Figure 4.5: Speedup (solid lines) vs prediction (dashed lines). Image Merge, 500 input tasks (*titanic*).

Comparing Functionally Equivalent Parallelisations

Finally, Figures 4.7 and 4.8 on pages 138 and 139 compare different parallel structures in the form of farms and pipelines for the image convolution example. These parallel structures are:

- A single task farm.
- A task farm of a parallel pipeline.
- A pipeline of two task farms.
- A task farm, where each worker is a parallel pipeline with a sequential computation in the first stage, and another task farm in the second stage.

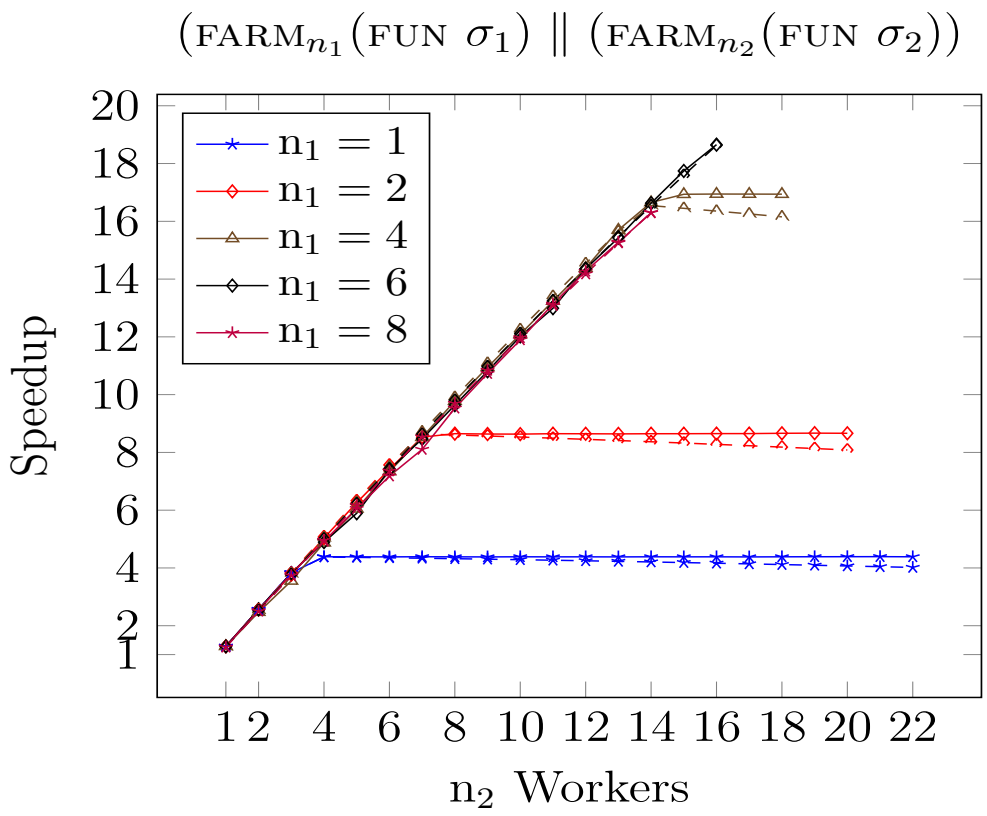


Figure 4.6: Speedup (solid lines) vs prediction (dashed lines). Image Convolution, 500 input tasks (*titanic*).

Note that for comparing the pipeline of two farms, the first stage has a fixed number of workers: 6 on titanic, 12 on lovelace. This implies that running this parallel structure on less than 6 (or 12) cores yields no speedups, and even a slowdown. Note that the cost models produce a tight prediction on both titanic and lovelace. This suggests that the predictability of the cost models scales up to 64 cores. In lovelace, the speedups drop significantly when using 56 to 62 cores, but they get significantly higher when using the full 64 cores. We currently do not have an explanation for this behaviour, but we speculate that it might be related to the frequency scaling. Although further experiments are needed to determine the cause of this behaviour, these issues are not crucial for the cost models that were presented in this chapter, since they rarely happen in our examples.

These examples collectively confirm the experimental results that we showed in [HBS16], and show that our cost models are able to correctly

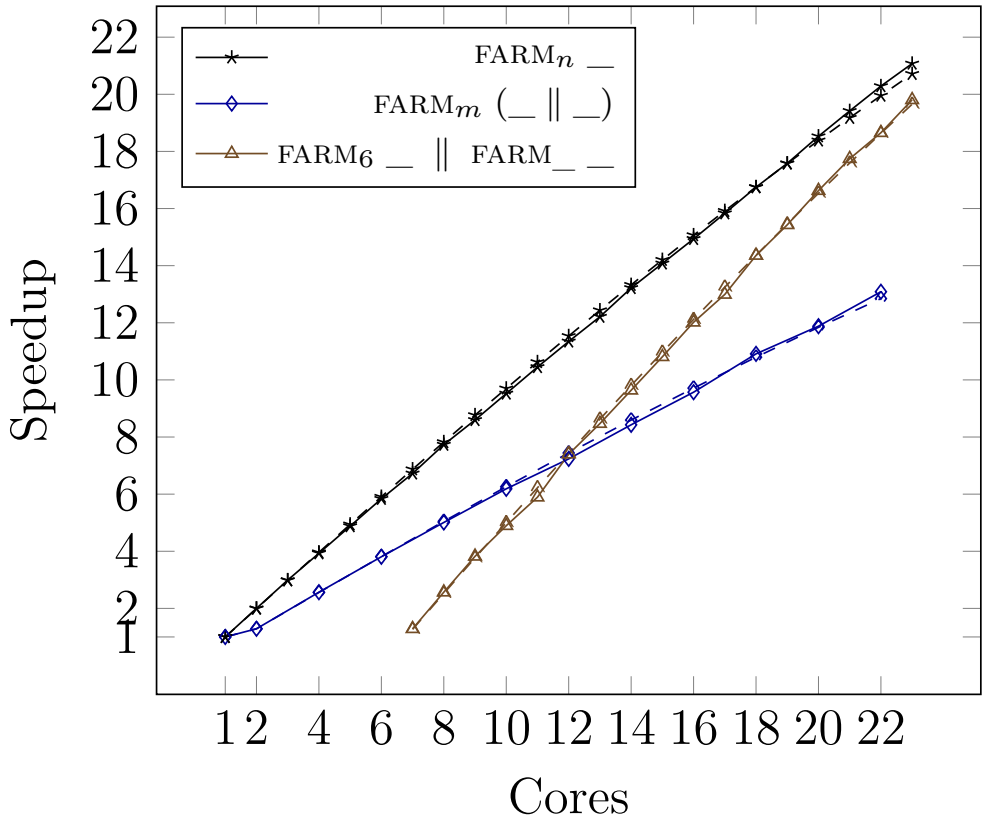


Figure 4.7: Speedup (solid lines) vs predicted (dashed lines). Different Parallel Structures for Image Convolution, 500 Images 1024 * 1024: *titanic*

capture queue contentions. We are thus able to predict a safe, tight upper bound of execution times.

4.5 Discussion

Speedups

The speedups that we showed are meant to illustrate the predictability and scalabiliy of the underlying queue-based model. Although the examples were run in 64 and 24-core machines, there could be some benefit in using this approach on a machine with less cores (e.g. 4). In this case, instead of trying complex nestings of parallel structures, the StA framework could be used to decide *where* to parallelise a program, and automatically rewrite it to avoid potential errors in the parallelisation process. However,

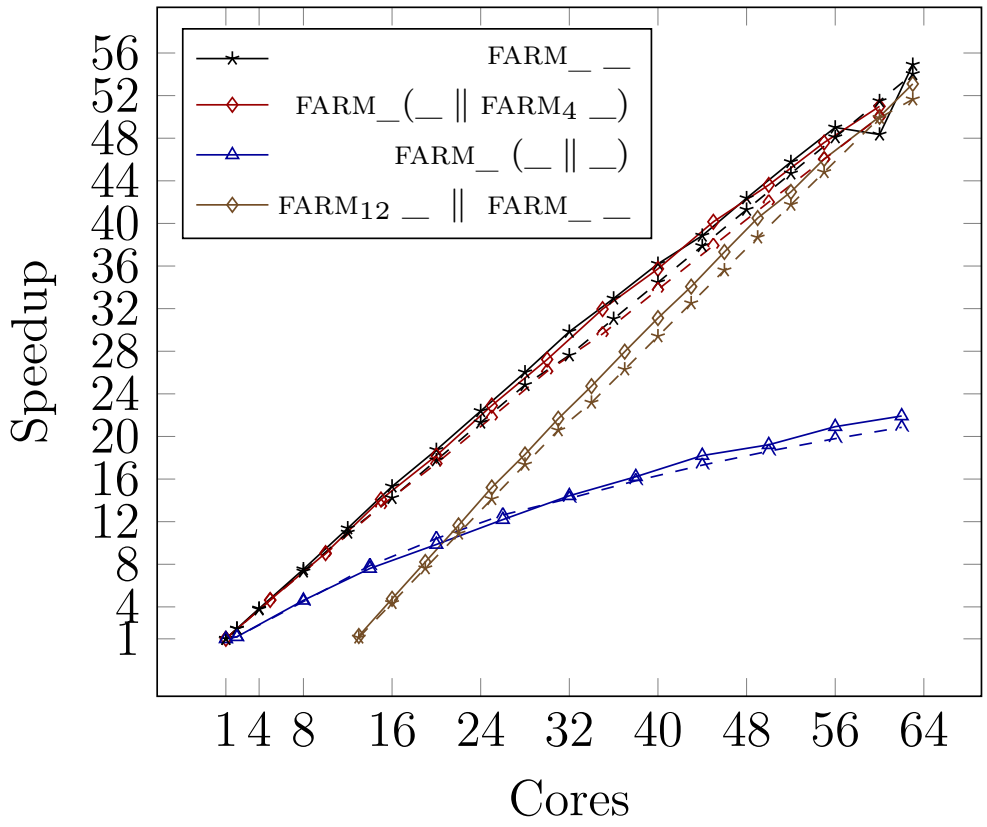


Figure 4.8: Speedup (solid lines) vs predicted (dashed lines). Different Parallel Structures for Image Convolution, 500 Images 1024 * 1024: *lovelace*.

more experiments would be required to determine whether this is useful (i.e. achieves good speedups) on regular machines with less resources.

Expressiveness

In this chapter, a new operational semantics in terms of queue and thread safe queue operations was presented. The queue-based model is general enough to define more complex skeletons. We illustrate this by sketching how a possible Bulk Synchronous Parallel skeleton. A skeleton following the BSP model must be a sequence of supersteps. We can use iteration for this:

$$\begin{aligned}&\texttt{bsp}(n, f, g)(Q_i, Q_o)\\ &= \texttt{superstep}(n, f, g)(Q_i, Q_i + Q_o)\end{aligned}$$

The superstep skeleton needs to enqueue values back in Q_i for n iterations. The queues Q_i and Q_o must be tuples of queues, each corresponding

to a processing unit:

$$
\begin{aligned}
&Q_i = q_{i1} \times \cdots \times q_{im} \\
&Q_o = q_{o1} \times \cdots \times q_{om} \\
&\texttt{superstep}(n, f, g)(Q_i, Q_i + Q_o) \\
&= \texttt{qfun}(f)(q_{i1}, q'_{i1}) \parallel \cdots \parallel \texttt{qfun}(f)(q_{im}, q'_{im}) \\
&\quad \parallel \texttt{qfun}(g')(q_0, q'_{i1} \times \cdots \times q'_{im}, q_0 \times Q_i + Q_o)
\end{aligned}
$$

This skeleton uses an additional queue, q_0, to indicate when the superstep has finished. It originally contains the value n, and is decreased after each superstep. The stage that computes $\texttt{qfun}(g')$ is in charge of the synchronisation. If the value in q_0 is zero, the result is produced to the output Q_o. Otherwise, the operation g is applied to the values in the intermediate queues q'_k, and feeds them back to the input queue. Since there is only one worker in charge of distributing the data to the different Q_i, this acts as a synchronisation stage of the superstep.

Cost Models

The cost models that are derived in this work are reasonably accurate, but simple. Our approach for deriving the cost of hylomorphisms is based on deriving *recurrence equations*. An alternative that is worth exploring is by Hope and Hutton [HH06]. Hope and hutton define a technique for reasoning about the space behaviour of hylomorphisms, based on a series of program transformations that convert the original hylomorphism into a stack-based abstract machine. It would be interesting to study whether this approach can be applied to improve time analysis as well.

Summary The simplicity of the queue-based model makes it sufficiently predictable, so cost models can be defined easily. The main novelty of this semantics is that it provides both a mechanism for compiling high-level skeletal programs to low-level parallel programs in terms of shared queue operations. This queue-based model is also powerful enough to define more complex and general skeletons. Finally, cost models were derived in a systematic way from this operational semantics. This close correspondence between the operational semantics and the cost models has, as advantages,

that the cost models are *sound* by construction, and that an automatic mechanism can be derived. A programmer using this queue-based model to define the operational semantics of new skeletons can derive, for free, both: a compilation process to low-level code, and cost equations that statically predict its run-time behaviour.

Chapter 5

SKI-ing Skeletons: Parallelising Explicit Recursive Functions using Applicative Expressions

I have now presented the full StA framework: Chapter 3 presented a type-and-effect system for a point-free purely functional language with hylomorphisms that can be used to parallelise sequential implementations, and Chapter 4 provides a mechanism for reasoning about the operational behaviour of different parallel structures, including correctness and execution times. However, the StA framework requires programmers to write their implementations in a point-free style, that is still far from a real programming language such as Haskell.

This chapter presents the purely functional language HH, and an extension of its typing system with Structured Arrows. HH is a subset of Haskell, similar to GHC's core language, with a rank-1 polymorphic type system, where recursion is restricted to "regular" recursive functions, i.e. no mutual or nested recursion is allowed. The current prototype implementation in `https://bitbucket.org/david_castro/skel` can be found under the directory *ApplicativeSkel*, and provides an implementation of the core part of the technique, which extracts hylomorphisms out of explicit recursive functions. This extension is a step towards applying the StA framework to a

real functional programming language.

5.1 Parallelising HH Functions

Listing 5.1 on page 145 contains an example of a Discrete Fourier Transform algorithm, the Cooley-Tukey Fast Fourier Transform algorithm [CT65] implementation in HH. The implementation of `fft` is equivalent to a Haskell 98 implementation, with the difference that the variables bound by the **`let-in`** expression must be ordered. For example, in Listing 5.1 on page 145, variable `n` must be defined in Line 6, since it is used in the definition of `ws` in Line 7. In Haskell, however, the order in which those definitions appear could be reversed. The full assumptions and differences between HH and Haskell are detailed later in this section.

There are many sources of parallelism in Listing 5.1. For example, the **`map`** function in Line 5, and the **`zipWith`** functions in Lines 8, 9 and 10. There is, however, another possible source of parallelism that arises from the recursive calls to `fft` in Lines 7 and 8. It is not straightforward to take the recursive structure of `fft` into account for parallelising it. However, if we can rewrite the function `fft` in Hylo, then it can be parallelised using structured arrows;

5.1.1 Example | Parallel FFT. A Cooley-Tukey algorithm implementation in Hylo can be parallelised in a number of alternative ways, as shown below:

$$
\begin{aligned}
& \mathtt{hylo}_T\ \mathtt{fft}^c\ \mathtt{fft}^d \\
\cong\ & \mathrm{DC}_{n,T}\ \mathtt{fft}^c\ \mathtt{fft}^d \\
\cong\ & \mathrm{REDUCE}_{n,T}\ \mathtt{fft}^c \circ \mathsf{ana}_T\ \mathtt{fft}^d \\
\cong\ & \ldots
\end{aligned}
$$

All of the parallelisations in Example 5.1.1 can be derived from a Cooley-Tukey algorithm implementation as a hylomorphism, $\mathtt{hylo}_T\ \mathtt{fft}^c\ \mathtt{fft}^d$. By using the technique described in this chapter, similar *parallel implementations can be derived automatically from the sequential implementation in Listing 5.1.*

Listing 5.1 Cooley-Tukey FFT in HH

```
fft : List Complex → List Complex
fft = λ xs.
  case xs of
       [x] → [x]
       xs  →
          let n      = length xs
              ws     = map (w n) (fromto 0 (minus n 1))
              xs'    = halves xs
              evens' = fft (π1 xs')
              odds'  = zipwith mult ws (fft (π2 xs'))
          in concat (zipwith plus  evens' odds')
                    (zipwith minus evens' odds')
```

Listing 5.2 Cooley-Tukey FFT in Hylo

```
T A B C = A + C × (B × C)

out_fft : List Complex → Complex + List Complex
out_fft = λ xs.
       case xs of
           [x] → inj1 x
           xs  → inj2 xs

fft^d : List Complex → T Complex (List Complex) (List Complex)
fft^d = (+2 id (△2 (π1 ∘ π1) (△2 π2 (π1 ∘ π2))
               ∘ ×2 halves
                    (λ n.map (w n) (fromTo 0 (minus n 1)))
               ∘ △2 id length)) ∘ out_fft

fft^c : T Complex (List Complex) (List Complex) → List Complex
fft^c = ▽2 (λ x. [x]) (uconcat
                         ∘ ×2 (uzipWith plus) (uzipWith minus)
                         ∘ ×2 id (uzipWith mult))

hfft : List Complex → List Complex
hfft = hylo_T fft^c fft^d
```

The function `hfft` in Listing 5.2 on page 145 presents one possible implementation of Listing 5.1 in Hylo. This implementation requires three components: (i) the functor `T` that represents the structure of the recursion, (ii) the "divide" function $\texttt{fft}^d$ that divides the inputs into the structure `T`, and (iii) the "combine" function $\texttt{fft}^c$ that takes an input with structure `T` and returns the output list. The parallelisations shown in Example 5.1.1 can be derived using the technique described in Chapter 3. However, it requires the code in Listing 5.1 to be converted to Listing 5.2.

5.1.1 Translating between HH and Hylo

The aim of this chapter is developing a novel technique for parallelising HH functions, by associating expressions in the language HH to Hylo, via *combinatory logic*. Converting from a point-wise to a point-free representation using hylomorphisms has been previously studied [CPP05, Cun05]. The tool DrHylo [CPP06] implements a translation scheme from typing judgements to a point-free representation of the corresponding point-wise program, based on [HIT96] and [Cun05].

The alternative proposed in this chapter is based on the close relation between λ-calculus and combinatory logic [Sch24, Cur30]. Thanks to this close correspondence, combinatory logic can be used as a "bridge" between a point-wise and point-free representations which is better suited for finding parallel structures. DrHylo relies on a particular interpretation of typing contexts, together with syntactic restrictions on recursive functions, which can be removed by using combinatory logic. The main differences between DrHylo and the applicative approach are:

1. Parts of the typing context can be "ignored" for the point-wise to point-free transformations. This makes it possible to translate to point-free style only the relevant parts of a function definition, leaving the variables that are irrelevant to introduce/change the parallel structure of a program.

2. DrHylo interprets the variables in the context as tuples. This interpretation is not necessary using applicative structures, and whenever

a composition of functions is required, an *uncurrying* transformation can be performed.

3. There is a single syntactic construct associated with application, and it does not rely on exponentials. This simplifies the reverse translation. Exponentials arise naturally from applying the uncurrying transformation to the identity combinator, and they can be avoided by applying a systematic transformation. Avoiding exponentials can be useful to fully determine the functions that are being applied at each stage of the parallel computation.

4. The restriction on recursive functions is placed on the equivalent point-free representations, and not in the syntactic construct for recursive definitions. This restriction will be enforced only where parallelism is required. This adds some flexibility on the kind of recursive functions that can be treated.

5.1.2 Stages of the Translation

The different parts of the translation between HH and Hylo are:

1. Relate the HH expressions to *applicative expressions*, which are expressions derived from combinatory logic. This relation provides a mechanism for doing the conversion in a bidirectional way.

2. Two inference algorithms based on the relation between HH and applicative expressions: (a) an algorithm that infers an associated applicative expression from a HH expression in a context, and (b) an algorithm that infers a HH expression from an applicative expression. By using these inference algorithms, all rewritings done at the applicative level can be applied to the HH level.

3. Relate *applicative expressions* to Hylo. Relating an applicative expression to Hylo is a systematic process.

These different steps in the relation between HH and Hylo are illustrated on a simplification of the FFT example in Listing 5.1. The working example is shown in Example 5.1.2.

5.1.2 Example | A Working FFT example in HH. The example that is going to be used throughout this chapter will be deriving a parallel implementation of the simplified FFT implementation below. In this example, a REDUCE$_k$ skeleton is specified. However, this parallel structure could be automatically derived from the `fft` implementation.

```
fft : List Complex → List Complex
    ∼ REDUCE_k _ ∘ _
fft = λ xs. case xs of
              [x] → [x]
              xs  → let  n    = length xs
                         xs' = halves xs
                    in comb ( genWs n) (fft (π1 xs'))
                                       (fft (π1 xs'))
```

In this example, the `comb` function performs all the **`zipWith`** and append operations in Lines 10, 11 and 12 of Listing 5.1, and `genWs` performs the **`map`** operation done in Line 7 of Listing 5.1. Before explaining the first stage of the translation, an explanation of the HH language and applicative expressions is provided.

5.2 The HH and Applicative Languages

5.2.1 The HH Language

The language HH (see Figure 5.2 on page 150) is a non-strict functional language that can be described as a subset of Haskell 98, with some differences. A program in HH is a sequence of datatype and function definitions, where there is no mutual recursion. Unlike in Haskell, a function or datatype can only be used if it has been previously defined. The type of function definitions must be declared before the body of the definition. The HH expressions are variables, primitive tuple and either type operations, λ-abstractions, application of, `let-in` expressions and `case` expressions. To illustrate a few of the differences between Haskell and HH, consider the code listings in Figure 5.1 on page 149. First, in HH type signatures are described with a

Listing 5.3 Cooley-Tukey FFT in HH

```
1  fft : [Complex] → [Complex]
2  fft = λ xs.
3    case xs of
4         [x] → [x]
5         xs  →
6            let n      = length xs
7                ws     = map (w n) (fromto 0 (minus n 1))
8                xs'    = halves xs
9                evens' = fft (π1 xs')
10               odds'  = zipwith mult ws (fft (π2 xs'))
11           in concat (zipwith plus  evens' odds')
12                     (zipwith minus evens' odds')
```

Listing 5.4 Cooley-Tukey FFT in Haskell

```
1  fft :: [Complex] -> [Complex]
2  fft [x] = [x]
3  fft xs  = zipWith (+) evens' odds' ++ zipWith (-) evens' odds'
4    where
5      (evens, odds) = halves xs
6      evens' = fft evens
7      odds'  = zipWith (*) ws (fft odds)
8      ws     = map (w n) (fromTo 0 (minus n 1))
9      n      = length xs
```

Figure 5.1: Comparison between implementations of Cooley-Tukey FFT in HH and Haskell

single colon, instead of Haskell's double colon. In HH, functions by pattern matching must be defined using λ-expressions and `case`-expressions, unlike in Haskell, where functions can be defined directly by pattern matching, and `let` expressions and λ-abstractions can use pattern matching (irrefutable patterns). There is no `where` clause in HH, and the definitions must be introduced in order, while in Haskell there are `where` clauses. Note that that the code in HH could be directly implemented in Haskell, subject to relatively minor syntactic changes (e.g. changing `:` to `::` in type signatures.

$$
\begin{array}{lcl}
\langle prog \rangle & ::= & \langle def \rangle \text{ ';' } \langle prog \rangle \\
& | & \langle def \rangle \\
\\
\langle def \rangle & ::= & \text{`}\texttt{data}\text{'} \langle uvar \rangle \langle uvars \rangle \text{ '='} \langle data\text{-}alts \rangle \\
& | & \langle var \rangle \text{ ':' } \langle type\text{-}scheme \rangle \\
& | & \langle var \rangle \text{ '='} \langle expr \rangle \\
\\
\langle data\text{-}alts \rangle & ::= & \langle data\text{-}alt \rangle \text{ '|' } \langle data\text{-}alts \rangle \\
& | & \langle data\text{-}alt \rangle \\
\\
\langle data\text{-}alt \rangle & ::= & \langle uvar \rangle \langle types \rangle \\
\\
\langle type\text{-}scheme \rangle & ::= & \forall \langle uvars \rangle \text{'.'} \langle type \rangle \\
& | & \langle type \rangle \\
\\
\langle types \rangle & ::= & \langle type \rangle \langle types \rangle \\
& | & \langle type \rangle \\
\\
\langle type \rangle & ::= & \langle uvar \rangle \\
& | & \langle type \rangle \text{ '}\rightarrow\text{' } \langle type \rangle \\
& | & \langle type \rangle \text{ '+' } \langle type \rangle \\
& | & \langle type \rangle \text{ '}\times\text{' } \langle type \rangle \\
& | & \langle var \rangle \langle types \rangle \\
\\
\langle expr \rangle & ::= & \langle var \rangle \\
& | & \langle prim \rangle \\
& | & \text{'}\lambda\text{'} \langle vars \rangle \text{'.'} \langle expr \rangle \\
& | & \langle expr \rangle \langle expr \rangle \\
& | & \text{`}\texttt{let}\text{'} \langle bndrs \rangle \text{ `}\texttt{in}\text{'} \langle expr \rangle \\
& | & \text{`}\texttt{case}\text{'} \langle expr \rangle \text{ `}\texttt{of}\text{'} \text{ '}\{\text{'} \langle alts \rangle \text{ '}\}\text{'} \\
\\
\langle bndrs \rangle & ::= & \langle bndr \rangle \text{ ';' } \langle bndrs \rangle \\
& | & \langle bndr \rangle \\
\\
\langle bndr \rangle & ::= & \langle uvar \rangle \text{ '='} \langle expr \rangle \\
\\
\langle alts \rangle & ::= & \langle alt \rangle \text{ ';' } \langle alts \rangle \\
& | & \langle alt \rangle \\
\\
\langle alt \rangle & ::= & \langle pat \rangle \text{ '}\rightarrow\text{' } \langle expr \rangle \\
\\
\langle pat \rangle & ::= & \langle var \rangle \\
& | & \langle uvar \rangle \langle pats \rangle \\
\\
\langle pats \rangle & ::= & \langle pat \rangle \langle pats \rangle \\
& | & \langle pat \rangle
\end{array}
$$

Figure 5.2: HH Syntax

5.2.2 Algebraic Data Types

The HH language supports the definition of algebraic datatypes. An algebraic datatype is defined as an upper case variable, possibly followed by a number of upper case variables, and finally by a number of alternatives, separated by the '|' character. Each of the alternatives is defined by a constructor applied to a number of types.

5.2.1 Example | The List Datatype in HH. The list datatype is defined as an empty list, `Nil`, or a cons-cell with an element of type `A` as the head of the list, followed by a tail of type `List A`.

```
data List A = Nil
            | Cons A (List A)
```

5.2.2 Example | The Binary Tree Datatype in HH. The tree datatype is defined in an analogous way.

```
data Tree A = Leaf
            | Node A (Tree A) (Tree A)
```

5.2.3 Syntax of HH

The syntax of HH (see5.2) is a subset of Haskell, with some minor syntactic differences. These are:

1. Type annotations are specified using single colon, :, instead of double colon ::.

2. Type annotations must be provided for top-level definitions.

3. Code blocks and definitions are always separated by a separator character. In the case of data alternatives, it is |. Case blocks, let-in definitions and top-level definitions are separated by a semicolon, ;.

4. Either types are named +, and tuple types by ×. For example, a 3-tuple that contains types `A`, `B` and `C` is represented as `A` × `B` × `C`[1].

HH Type System

The type system of HH is a *predicative rank-1 polymorphic type system* [Pie02].

Rank-1 In a *polymorphic* type system, some types will depend on type variables. This is specified with the ∀ construct. Consider, for example, the type of the `length` function for lists. This type accepts lists that contain elements of any type, so the type annotation is parameterised by a type variable `A`.

```
length : ∀ A. List A → Int
```

Rank-1 restricts the position of the quantifiers to the outermost level of the type. In general, for any fixed k, a *rank-k* polymorphic type system restricts the position of the quantifiers to the left of less than k arrows. A *rank-n* or *higher-rank* type system is one in which quantifiers may occur in unrestricted positions.

5.2.3 Example | A Rank-2 Type. In this example, the position of ∀ appears to the left of one arrow, here marked with the superscript '$*$', $\to^*$, so this type would not be valid in a rank-1 polymorphic type system, but it is a valid rank-2, or higher-rank type.

```
f : (∀ A. (A → A) → A → List A) →* List Int
          └──────────────────┘
              scope of A
```

Predicative Types parameterised by type variables are called *type schemas*. In a predicative type system, type variables cannot be instantiated by *type schemas*.

[1] For presentation purposes, some symbols and keywords are "prettified" in this document. In a real implementation, the symbol '`*`' will be used instead of '×', and the keyword `forall` will be used instead of the symbol ∀.

Kinds *Kinds* can be thought of as *the types of the types*. They are used to statically check that a type is well-formed.

$$\begin{array}{rcl} \langle kind \rangle & ::= & \text{`Type'} \\ & | & \langle kind \rangle \text{ `} \rightarrow \text{ '} \langle kind \rangle \end{array}$$

The kind 'Type' is the kind of the types. Any primitive, constant type has this kind. For example, the fact that "`Int` is a type" can be stated as `Int` : Type. Functions, products, either types have kind 'Type'.

Type constructors have kind '→'. Type constructors take a kind as an argument, and produce a kind as a result. For example, the type constructor `List` has kind Type → Type. The type constructor `List` applied to the type `Int` is a type, `List Int` : Type. However, the type constructor `List` : Type → Type, applied to the type constructor `Tree` : Type → Type is not "well kinded". Every type in the language HH is kind-checked with kind Type. A kind environment is used for kind-checking purposes. A well-kinded data definition `data` $T\ A_1 \cdots A_n = \cdots$ extends the environment with $T : K_1 \rightarrow \cdots K_n \rightarrow$ Type, where K_i are the kinds inferred for the typesA_i in the definition of T. The kind inference rules for HH types and data definitions are entirely standard.

Inference Rules

The set of rules that define the HH type system is given in Figure 5.3 on page 154. These rules are entirely standard for a predicative rank-1 polymorphic type system, and they have the standard soundness property. As usual, the soundness property of this type system is the formalisation of the notion that no type error can happen in run-time. In Section 5.3, these inference rules will be modified slightly so that they can be used to statically check whether a program can be parallelised according to some given parallel structure, and they can be used to systematically explore the space of all possible alternative parallel implementations for a given function in HH.

$$\text{Gen}\frac{\Gamma \vdash M : T \qquad A \notin \mathsf{fv}(T)}{\Gamma \vdash M : \forall A.\ T}$$

$$\text{Spec}\frac{\Gamma \vdash M : \forall A. T_1}{\Gamma \vdash M : T_1[T_2/A]}$$

$$\text{Var}\frac{}{\Gamma, x : T \vdash x : T}$$

$$\text{Abs}\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x.M : T_1 \to T_2}$$

$$\text{App}\frac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash N : T_1}{\Gamma \vdash M\ N : T_2}$$

$$\text{Case}\frac{\begin{array}{c}\Gamma \vdash M : T_1 \\ \Gamma \vdash_a p_1 \to N_1 : T_1 \to T_2 \\ \cdots \\ \Gamma \vdash_a p_k \to N_k : T_1 \to T_2\end{array}}{\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{p_1 \to N_1; \cdots ; p_k \to N_k\}\ : T_2}$$

$$\text{Let}\frac{\Gamma, x : T_1 \vdash M : T_1 \qquad \Gamma, x : T_1 \vdash N : T_2}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : T_2}$$

$$\text{Alt}\frac{\Gamma, \Gamma' \vdash N : T_2 \qquad \Gamma \vdash_p p : T_1; \Gamma'}{\Gamma \vdash_a p \to N : T_1 \to T_2}$$

$$\text{VPat}\frac{}{\Gamma \vdash_p x : T_1; [x : T_1]}$$

$$\text{CPat}\frac{\begin{array}{c}\Gamma \vdash C : T_{11} \to \cdots \to T_{1m} \to T_2 \\ \Gamma \vdash_p p_1 : T_{11}; \Gamma_1 \qquad \cdots \qquad \Gamma \vdash_p p_m : T_{1m}; \Gamma_m\end{array}}{\Gamma \vdash_p C\ p_1 \cdots p_m : T_2; \Gamma_1, \ldots, \Gamma_m}$$

Figure 5.3: Typechecking HH Expressions.

5.2.4 Applicative Structures

The technique that is described in this chapter relies on *combinatory logic*, instead of the more common approach of using *supercombinators* [Hug82]. The reason for this is that, while supercombinators are good for compiling, for finding specific instances of common point-free primitives (e.g. function

composition) is simpler when working with a small, fixed set of combinators.

Applicative operators have been used for concurrency and parallelism before. Marlow et al. [MBCP14] define Haxl, a concurrency abstraction built on top of Haskell Applicative Functors, that allow implicit concurrency to be extracted from Monad and Applicative instances. Unlike their approach, which is based on the implicit parallelism of the `<*>` applicative operator, our approach uses applicative structures to discover potential instances of parallel patterns.

Combinatory Logic

Combinatory logic [Sch24, Cur30] was originally defined as a notation for mathematical logic, and it was later used as a model of computation [Bar84]. Combinatory logic is based on the notion of *combinators*: higher-order functions that are defined in terms of function application, and other previously defined combinators. Common examples of combinators are the `S`, `K` and `I` combinators:

$$\texttt{S}\ f\ g\ x = f\ x\ (g\ x) \qquad \texttt{K}\ x\ y = x \qquad \texttt{I}\ x = x$$

The `S` combinator is a generalised form of application, where `S` f g x applies f to g, after passing x to f and g. The `K` combinator is the *constant* combinator. The constant combinator `K` applied to some x, `K` x, is a function which always returns x, given any input. Finally, the `I` combinator is the *identity* combinator, which always returns the same input unaltered. Note that some combinators could be implemented in terms of other combinators. For example, the `I` combinator is extensionally equal to `S K K` and `S K S`.

Function composition Two additional combinators, `B` and `C`, originally introduced by Schönfinkel [Sch24] and rediscovered by Haskell Curry [Cur30], capture different notions of function composition. The `C` combinator can be thought of as a "swap function", that exchanges the order of the arguments that are passed to f. The argument to `C` f x is passed to f, which is then applied to x:

$$\texttt{C}\ f\ x\ y = f\ y\ x$$

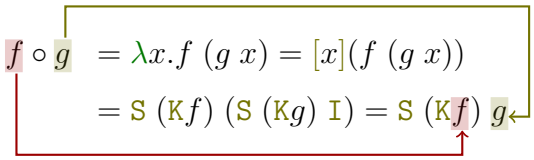


Figure 5.4: Function composition, represented using the S and K combinators, derived using bracket abstraction.

$$\mathtt{C} f\ x = \mathtt{S}\ f\ (\mathtt{K}\ x)$$

It is possible to define C in terms of S and K as well. The resulting expression is, however, more complex than that of the B combinator, and it is not of interest for the purposes of this chapter.

The B combinator is equivalent to the usual notion of function composition, and the technique on this chapter relies on finding instances of this combinator in the source code.

$$f \circ g = \mathtt{B}\ f\ g = \lambda x.f\ (g\ x)$$

Since it can be defined using S and K combinators (see Figure 5.4 on page 156), the technique that we describe in this chapter relies on finding the following instances of the S and K combinators.

$$\boxed{\mathtt{B} = \mathtt{S}\ (\mathtt{KS})\ \mathtt{K}}$$

$$\mathtt{B}\ f\ g = \mathtt{S}\ (\mathtt{KS})\ \mathtt{K}\ f\ g = (\mathtt{KS}\ f)\ (\mathtt{K}\ f)\ g = \mathtt{S}\ (\mathtt{K}\ f)\ g$$

To associate terms in HH, we will rely on standard techniques for translating λ-expressions to combinator expressions. An algorithm for translating from λ-expressions to combinatory logic is generally known as *abstraction elimination* [Bar84]. One of the possibilities for performing abstraction elimination is through *bracket abstraction*, which is a process that takes a variable x, an expression E, and produces an expression, $[x].E$ that is extensionally equal to $\lambda x.E$. An example of this is Turner's bracket abstraction [Tur79]:

$$\boxed{\begin{array}{ll} [x]x & = \mathtt{I} \\ [x]y & = \mathtt{K}\ y \\ [x](E_1\ E_2) & = \mathtt{S}\ [x]E_1\ [x]E_2 \end{array}}$$

Applicative Expressions

Generally, combinator expressions are built by using only a set of predefined combinators and application. Expressions that share this pattern are generally called *applicative*, and this has been captured by the Haskell *Applicative* type class. In this chapter, the term '*applicative expressions*' is used to refer to terms defined using the specific syntax in Definition 5.2.1.

Syntax of Applicative Expressions

The syntax of applicative expressions, with atomic terms $\mathcal{M}$, and primitive operations $\mathcal{P}$, is defined as follows.

5.2.1 Definition | Syntax of Applicative Expressions.

$$\begin{array}{cccccccccc} & & M \in \mathcal{M} & & & & p \in \mathcal{P} & & & \\ a_i \in \mathcal{A}_{\mathcal{M}} & ::= & [M] & \mid & p & \mid & \langle\rangle^n & \mid & @^n_m\ a_1 \cdots a_m & \end{array}$$

Applicative expressions are parameterised by the domain $\mathcal{M}$. If M is some term in domain $\mathcal{M}$, then $[M]$ is an *atomic expression*. If p is in the domain of primitive operations, $p \in \mathcal{P}$, then p is an applicative expression. The applicative expression $\langle\rangle^n$ is equivalent to n applications of the `K` combinator, i.e. $\langle x\rangle^n = \overbrace{\texttt{K}\ (\ldots(\texttt{K}\ x))}^{n \text{ times}}$. In other words, $\langle\rangle^n$ is the composition of n `K` combinators. If $n = 0$, then $\langle\rangle^0 = \texttt{I}$. Finally, the combinator $@^n_m\ a_1 \cdots a_m$ is the application of m terms, in a context of n elements, and is similar to the Φ^n_m defined by Curry et al. [CFC58]. Applying a $@^n_m$ combinator to n inputs is equivalent to applying the n inputs to the $a_{1\cdots m}$ terms, and then applying the resulting $a_{2\cdots m}$ to the resulting a_1. Since we require $@^n_m$ to be fully applied, the subscript is generally omitted.

$$\begin{array}{c} @^0\ (@^n\ a_1 \cdots a_m)\ x_1 \cdots x_n \\ = @^0\ a_1\ x_1 \cdots x_n\ (@^0\ a_2\ x_1 \cdots x_n) \cdots (@^0\ a_m\ x_1 \cdots x_n) \end{array}$$

The $@$ applicative structure can be defined in terms of `S` and `K`. We show below the case for $@^n_2$, since $@^n_m$ can be defined by nesting $@^n_2$:

$$@^n = \begin{cases} \texttt{I} & \text{if } n = 0 \\ \texttt{S} \circ (\texttt{S}\ (\texttt{K}\ @^{n-1})) & \text{if } n > 0 \end{cases}$$

5.2.1 Lemma $\lambda x_1 \cdots x_n.M\ N \approx @^n\ (\lambda x_1 \cdots x_n.M)\ (\lambda x_1 \cdots x_n.N)$

Proof By induction on n.

Case $n = 0$.

$$M\ N = @^0\ M\ N = \mathtt{I}\ M\ N = M\ N.$$

Case $n = m + 1$.

$$\begin{aligned}
& \lambda x\ x_1 \cdots x_m.M\ N \\
\approx\ & \quad \{ \text{ Induction Hypothesis } \} \\
& \lambda x.@^m(\lambda x_1 \cdots x_m.M)\ (\lambda x_1 \cdots x_m.N) \\
\approx\ & \quad \{ \text{ Apply Bracket Abstraction } \} \\
& [x].@^m(\lambda x\ x_1 \cdots x_m.M)\ (\lambda x\ x_1 \cdots x_m.N) \\
\approx\ & \quad \{ \text{ Unfold Bracket Abstraction } \} \\
& \mathtt{S}(\mathtt{S}(\mathtt{K}\ @^m)(\lambda x\ x_1 \cdots x_m.M))\ (\lambda x\ x_1 \cdots x_m.N) \\
\approx\ & \quad \{ \text{ Fold Function Composition } \} \\
& (\mathtt{S} \circ (\mathtt{S}(\mathtt{K}\ @^m)))(\lambda x\ x_1 \cdots x_m.M)\ (\lambda x\ x_1 \cdots x_m.N) \\
\approx\ & \quad \{ \text{ Fold Definition of } @ \} \\
& @^{m+1}(\lambda x\ x_1 \cdots x_m.M)\ (\lambda x\ x_1 \cdots x_m.N)
\end{aligned}$$

□

Notation and Assumptions

The work in this chapter uses the following notation and assumptions. The usual juxtaposition, $a_1\ a_2$, is used instead of explicitly writing $@^0\ a_1\ a_2$. Recall that the composition of n $\mathtt{K}$ combinators is written $\langle\rangle^n$. Applying n $\mathtt{K}$ combinators to an expression, $\langle\rangle^n\ a$, is written $\langle a\rangle^n$. The notation $\langle a\rangle^i_m$ stands for $@^i_{m+1}\ \langle a\rangle^i$. Since the combinators $@$ are required to be fully applied, the subscript is omitted.

$$\langle a\rangle^i\ a_1 \cdots a_m = @^i\ \langle a\rangle^i\ a_1 \cdots a_m$$

The composition operation is defined in terms of @ and $\langle\rangle^1$:

$$a \circ b = \langle a \rangle^1 \ b$$

The work in this chapter is developed modulo associativity of $\circ$:

$$\boxed{a \circ b \circ c} = a \circ (b \circ c) = (a \circ b) \circ c$$
$$= \langle \langle a \rangle^1 \ b \rangle^1 \ c = \langle a \rangle^1 \ (\langle b \rangle^1 \ c)$$

The following notation is used for sum and product types. Recall that $\&_n$ is a n-tuple constructor, i.e. $\&_n \ x_1 \cdots x_n$ is a tuple of n elements, $(x_1, \ldots, x_n)$. The usual product morphism is defined as follows:

$$\triangle_n \ \ a_1 \cdots a_n = \langle \&_n \rangle^1_n \ a_1 \cdots a_n$$

The expression $\triangle_n \ \ a_1 \cdots a_n$ is a function that takes one argument, and returns the n-tuple that results from applying the a_i to the input:

$$(\triangle_n \ \ a_1 \cdots a_n) \ x \ = \ (a_1 \ x, \ldots, a_n \ x)$$

The product functor is defined in terms of the product morphism and projection operations.

$$\times_n \ a_1 \cdots a_n = \triangle_n \ (a_1 \circ \pi_1) \cdots (a_1 \circ \pi_n)$$

The expression $(\times_n \ a_1 \cdots a_n)$ is a function that, when applied to a tuple $(x_1, \ldots, x_n)$, it returns the tuple $(a_1 \ x_1, \ldots, a_n \ x_n)$. Finally, the sum functor is defined in an analogous way to the product functor:

$$+_n \ a_1 \cdots a_n = \nabla_n \ (\mathtt{inj}_1 \circ a_1) \cdots (\mathtt{inj}_n \circ a_n)$$

The expression $(+_n \ \ a_1 \cdots a_n)$ is a function that takes some $\mathtt{inj}_i \ x$, for $i \in [1 \cdots n]$, and returns $\mathtt{inj}_i \ (a_i \ x)$.

Remark By using a Church-encoding of sums and products, primitive operations could be entirely represented using applicative expressions:

$$\&_i = @^{i+1} \ |^i_0 \ |^0_i \cdots |^{i-1}_1 \qquad \nabla_i = @^{i+1} \ |^i_0 \ |^0_i \cdots |^{i-1}_1$$
$$\pi^j_i = @^1 \ \mathtt{I} \ |^i_{j-i} \qquad \mathtt{inj}^j_i = @^{j+1} \ |^i_{j-i} \ |^0_j$$

However, this representation adds no benefit for the purpose of parallelising patterns of recursion. In order to reason about patterns of recursion and parallelism, the usual product and sum combinators, $\triangle_i$ and $\triangledown_i$, expose more structure than directly working with the church encoding of those primitive types. In the rest of this chapter, tuple constructors, projections, sum injections and either combinators will be considered primitive operations, as shown in Figure 2.5 on page 42.

5.3 Structure Checking Relation

This section contains a description of the translation from HH to $\mathcal{A}_{\mathsf{HH}}$, i.e. applicative structures that embed HH terms as atomic expressions. Recall the code of the FFT working example in Example 5.1.2:

```
fft : List Complex → List Complex
      ∼ REDUCE_k _ ∘ _
fft = λ xs. case xs of
              [x] → [x]
              xs  → let n   = length xs
                        xs' = halves xs
                    in comb (genWs n) (fft (π1 xs'))
                                      (fft (π1 xs'))
```

The type of the `fft` function is annotated with the structure $\text{REDUCE}_k\ \_\circ\_$. This structure contains two holes, represented with the underscore character, _. Chapter 3 describes a normalisation process that takes a parallel structure, and turns it into a composition of hylomorphisms, which can then be compared with the program structure. The normalised parallel structure for this example is shown below:

$$\text{REDUCE}_k\ \_\circ\_ \rightsquigarrow^* \text{CATA}\_\ \_\circ\_.$$

This implies that the "program structure" must be equivalent to the composition of a catamorphism and some other function. In order to check this equivalence, a structure must be extracted from `fft`, normalised and then compared with the target structure. Extracting the structure from `fft` is done in two stages:

$$\begin{array}{l}\texttt{fft} \sim \mathtt{D}_{\texttt{fft}}\ \Gamma\ (@^1\quad (\langle\nabla_2\rangle^1\quad (\langle\texttt{Cons}\rangle^2\ |_0^1\ \langle\texttt{Nil}\rangle^2)\\ \qquad\qquad\qquad\qquad\qquad\quad (@^2\quad (@^3\quad (\langle\texttt{comb}\rangle^4\quad (\langle\texttt{genWs}\rangle^4\ |_1^2)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\langle\texttt{fft}\rangle^4\ (\langle\pi_1\rangle^4\ |_0^3))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\langle\texttt{fft}\rangle^4\ (\langle\pi_2\rangle^4\ |_0^3)))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\langle\texttt{halves}\rangle^3\ |_1^1))\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad (\langle\texttt{length}\rangle^2\ |_0^1))))\\ \qquad\qquad\qquad\qquad\quad (\langle\texttt{out}([\texttt{x}],\texttt{xs})\rangle^1\ \mathtt{I}))\end{array}$$

(a) The `fft` (Example 5.1.2, page 148) associated structure.

$$\begin{array}{l}\texttt{fft} \sim \mathtt{D}_{\texttt{fft}}\ \Gamma\quad (\nabla_2\quad (\langle\texttt{Cons}\rangle^1\ \mathtt{I}\ \langle\texttt{Nil}\rangle^1)\\ \qquad\qquad\qquad\qquad (\langle\texttt{comb}\rangle^1\ (\texttt{genWs}\circ\pi_1)\ (\texttt{fft}\circ\pi_{2,1})\ (\texttt{fft}\circ\pi_{2,2})\\ \qquad\qquad\qquad\qquad\quad \circ\ \triangle_2\ \ \pi_2\ (\texttt{halves}\circ\pi_1)\circ\ \triangle_2\ \ \mathtt{I}\ \texttt{length})\\ \qquad\qquad\qquad\quad \circ\ \texttt{out}([\texttt{xs}],\texttt{xs}))\end{array}$$

(b) Simplified `fft` associated structure.

1. Obtain an implementation of `fft` in the $\mathcal{A}$ language of applicative expressions: $\texttt{fft} \sim a$. The applicative expression a is known as the *associated structure* of `fft`.

2. Find a hylomorphism, or composition of hylomorphisms, in the structure of `fft`. This may require rewriting the structure a into an equivalent a'. The equivalent implementation of a' as a composition of hylomorphisms, h, is captured by relation: $\boxed{a' \updownarrow h}$.

Building this technique on applicative expressions has several advantages. First, applicative expressions are abstractions built on top of a well-known theory, combinatory logic, and this has the crucial advantage that the results available for combinatory logic can be reused and extended for rewriting and obtaining a structure from a function. This is illustrated by the definition of the technique for extracting hylomorphisms from applicative structures, in Section 5.5.

This section focuses on the first stage, and presents the novel *associated structure relation* (Definition 5.3.1), that associates HH expressions with semantically equivalent applicative expressions. This relation contains, as

novel features:

1. A compositional approach to point-free program transformations that uses a single construct for representing application. This contrasts the more common usage of exponentials to translate an application of two terms in a context. Exponentials will arise naturally from an uncurrying transformation of the application construct @.

2. A new structure, called `D`, for representing definitions. This structure is used to focus the point-free transformation on the relevant parts of the program, by ignoring variables that are irrelevant to the parallelisation of a function.

The associated structure of the FFT example above is shown in Figure 5.5a on page 161. There are two elements that deserve commenting in this structure. First, it is built entirely using the application construct, @, the constant construct, $\langle\rangle^i$, primitive operations, π_i and ∇, and free variables. These constructs are derived from combinatory logic, as was shown in Section 5.2, so the properties derived from combinatory logic can be applied to systematically simplify it to the structure Figure 5.5b (see Section 5.5).

Extracting a hylomorphism from the simplified structure in Figure 5.5b is shown in Section 5.5. This section contains a presentation on how to derive the structure in Figure 5.5a from the code in Example 5.1.2, by using the associated structure relation.

5.3.1 Associated Applicative Structures

5.3.1 Definition | Associated Structure Relation. *A term M "has associated structure" a in context $\Gamma = [x_1, \ldots, x_n]$,*

$$\boxed{\Gamma \vdash M \sim a}$$

a is a combinator expression such that $a\ \overline{\Gamma} \approx M$, where $\overline{\Gamma} = x_1\ \cdots\ x_n$.

5.3.2 Definition | Structure-Annotated Types. *A type $a \sim A$ is a "struc-*

ture annotated type" such that for any term M and context Γ,

$$\boxed{\Gamma \vdash M : a \sim A} \quad \Leftrightarrow \quad \Gamma \vdash M \sim a \quad \wedge \quad \Gamma \vdash M : A.$$

The type system in Figure 5.3 on page 154 can be extended to annotate types with associated structures, by using the Associated Structure Relation. The full rules for the Associated Structure Relation are given in Figure 5.7 on page 171.

Variables

Consider typechecking a variable in a context:

$$\frac{}{\Gamma, x : A \vdash x : A}$$

Generally, the notation used for stating that a variable x occurs in an environment Γ with type A is $\Gamma, x : A$. However, if Γ is defined as an ordered sequence of pairs variable-type, then variable x can occur in any position k, $1 \leq k \leq n$, inside $\Gamma = x_1 : T_1, \ldots, x_n : T_n$.

$$\frac{}{x_1 : T_1,\ \ldots\ x_{k-1} : T_{k-1},\ x : A,\ x_k : T_k,\ \ldots\ x_n : T_n \vdash x : A}$$

The associated structure of variables can be obtained from the context, using bracket abstraction:

$$[x_1].\ \ldots [x_{k-1}].\ [x].\ [x_k].\ \ldots [x_n].\ x$$

Assuming that none of the x_i, with $i > k$, are equal to x, the following simplification is applied.

$$\begin{aligned}&[x_1].\ \ldots [x_{k-1}].\ [x].\ \overbrace{\mathtt{K}\ (\ldots(\mathtt{K}\ x)\ldots)}^{n-k \text{ times}}\\ &= [x_1].\ \ldots [x_{k-1}].\ [x].\ \langle x \rangle^{n-k} = [x_1].\ \ldots [x_{k-1}].\ \langle\rangle^{n-k}\end{aligned}$$

Finally, since none of the remaining x_i occur free in the RHS of the equation, the remaining square abstractions can be simplified to $k-1$ applications to the $\mathtt{K}$ combinator.

$$[x_1].\ \ldots [x_{k-1}].\ [x].\ [x_k].\ \ldots [x_n].\ x = \langle\langle\rangle^{n-k}\rangle^{k-1}$$

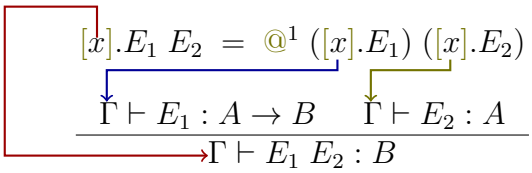


Figure 5.6: Relation between bracket abstraction and application typechecking.

The following notation is used:

$$|_{n-k}^{k-1} = \langle\langle\rangle^{n-k}\rangle^{k-1}$$

The structure-annotated inference rule for variables is show below. The notation $\Gamma(x) = k, n$ is used o represent that variable x is in Γ in position k of a context of size n.

$$\text{Var}\frac{\Gamma(x) = k, n}{\Gamma \vdash x \sim |_{n-k}^{k-1}}$$

Abstraction

Abstraction is handled by introducing variables in the context. Figure 5.6 illustrates this. By introducing variable x in the context Γ, the application of E_1 to E_2 can be translated to an instance of $@^1$. Introducing a variable in a context is handled by the rule for lambda abstraction. The rule is:

$$\text{Abs}\frac{\Gamma, x \vdash M \sim a}{\Gamma \vdash \lambda x.M \sim a}$$

Application

The HH application is translated to application nodes $@^i$, where i is the size of the context in which the application occurs. Recall that $M \sim a$ in context Γ means that $a\ \overline{\Gamma} \approx M$. Using bracket abstraction, the application $M\ N$ in context Γ can be rewritten as follows:

$$M\ N = (\lambda\overline{\Gamma}.M\ N)\ \overline{\Gamma}$$

By using the property of applicative application, this is rewritten to:

$$M\ N = (\lambda\overline{\Gamma}.M\ N)\ \overline{\Gamma} = (@^i\ (\lambda\overline{\Gamma}.M)\ (\lambda\overline{\Gamma}.N))\ \overline{\Gamma}$$

The final rule is, therefore:

$$\text{App}\frac{\Gamma \vdash M \sim a_1 \qquad \Gamma \vdash N \sim a_2 \qquad n = |\Gamma|}{\Gamma \vdash M\ N \sim @^n\ a_1\ a_2}$$

Case Expressions

The only combinator that deals with "alternatives" is the *either* combinator in $\mathcal{P}$, ∇_i. Case expressions must therefore use this combinator. However, pattern matching alternatives use algebraic datatypes. For example,

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
...
... case t of {
      ; Node a (Node b c) d -> f a b c d
      ; Node a Leaf       b -> g a b
      ; Leaf                -> z
      }
...
```

These alternatives need to be converted to a sum type so that they can be handled by the ∇_i construct. This is done by the `out` function. The `out` function converts a list of possibly nested pattern matching alternatives into a list of alternatives of an equivalent either type.

$$\texttt{out}\,(p_1, \ldots, p_k) = \begin{cases} \lambda x.\texttt{case}\ x\ \texttt{of} \\ \quad p_1 \to \texttt{inj}_1\ \texttt{TP}[\![p_1]\!] \\ \quad \cdots \\ \quad p_k \to \texttt{inj}_k\ \texttt{TP}[\![p_k]\!] \end{cases}$$

The translation $\texttt{TP}[\![p]\!]$ stands for "tuple pattern", and is a tuple of all variables bound by pattern p_1.

$$\begin{aligned} \texttt{TP}[\![x]\!] &= x \\ \texttt{TP}[\![C\ p_1 \cdots p_n]\!] &= (\texttt{TP}[\![p_1]\!], \cdots, \texttt{TP}[\![p_n]\!]) \end{aligned}$$

In the example above,

```
out (Node a (Node b c) d, Node a Leaf , Leaf) =
  λ x. case x of {
```

```
      ; Node a (Node b c) d -> inj_1 (a,(b,c),d)
      ; Node a Leaf       b -> inj_2 (a,b)
      ; Leaf                -> inj_3 ()
    }
...
```

Note that the `out` function is a closed term. This implies that square abstraction can be used in any applicative expression that embeds the `out` function as atomic structure. Using this out function, the original case expression is equivalent to:

```
case t of {
   ; Node a (Node b c) d -> f a b c d
   ; Node a Leaf       b -> g a b
   ; Leaf                -> z
   }
   ≈ (▽_3 f' g' z')
        (out(Node a (Node b c) d, Node a Leaf, Leaf) t)
```

In the above example, the functions `f'` and `g'` are used instead of `f` and `g`. These functions roughly correspond to the uncurrying of `f` and `g`:

$\texttt{f'} = \langle \texttt{f} \rangle^1 \ \pi_1 \ (\pi_1 \circ \pi_2) \ (\pi_2 \circ \pi_2) \ \pi_3$

$\texttt{g'} = \langle \texttt{g} \rangle^1 \ \pi_1 \ \pi_2$

$\texttt{z'} = \langle \texttt{z} \rangle^1$

This translation is captured by the 'Case' rule below.

$$\text{Case}\frac{\Gamma \vdash M \sim a_M T \qquad n = |\Gamma| \qquad \Gamma \vdash_a p_1 \to N_1 \sim a_1 \quad \cdots \quad \Gamma \vdash_a p_k \to N_k \sim a_k}{\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{p_1 \to N_1; \cdots; p_k \to N_k\} \sim \langle \nabla_k \rangle^n \ a_1 \cdots a_k \ (\langle \{\texttt{out}(p_1, \ldots, p_k)\} \rangle^n \ a_M)}$$

The case rule uses a different rule for case alternatives. The rules $\vdash_a$ associate a case alternative $p \to N$, with an applicative structure that takes a tuple with all variables bound by the pattern p, and returns N:

$$\text{Alt}\frac{\Gamma, x \vdash N[p \downarrow x] \sim a \qquad x \notin \Gamma \wedge x \notin \text{fv}(N)}{\Gamma \vdash_a p \to N \sim a}$$

Pattern-variable substitution is denoted by $N[p \downarrow x]$, and corresponds to applying a substitution that turns any variable in p by a projection from

the tuple $\texttt{TP}[\![p]\!]$ in term N.

$$\begin{aligned}[x \downarrow y] &= [y/x] \\ [C\ p_1 \cdots p_n \downarrow y] &= [p_1 \downarrow \pi_1\ y] \cup \cdots \cup [p_n \downarrow \pi_n\ y]\end{aligned}$$

5.3.1 Lemma | Empty Pattern Substitution. *For all pattern p, the substitution $[p \downarrow \texttt{TP}[\![p]\!]]$ is equivalent to an empty substitution. I.e.*

$$[p \downarrow \texttt{TP}[\![p]\!]] = [].$$

Proof By induction on p,
Case / $p = x$. $[x \downarrow \texttt{TP}[\![x]\!]] = [x \downarrow x] = [x/x] = []$
Case / $p = C\ p_1 \cdots p_n$.

$$\begin{aligned}& [C\ p_1 \cdots p_n \downarrow \texttt{TP}[\![C\ p_1 \cdots p_n]\!]] && \{\text{definition of } \texttt{TP}\} \\ =\ & [C\ p_1 \cdots p_n \downarrow (\texttt{TP}[\![p_1]\!], \cdots, \texttt{TP}[\![p_n]\!])] && \{\text{definition of } \downarrow\} \\ =\ & [p_1/\texttt{TP}[\![p_1]\!]] \cup \cdots \cup [p_n/\texttt{TP}[\![p_n]\!]] && \{\text{induction hypothesis}\} \\ =\ & [] \cup \cdots \cup [] = [] &&\end{aligned}$$

5.3.2 Lemma | Pattern η-equivalence. *For all expression M, variable x and pattern p,*

$$M = (\lambda x, M[p \downarrow x])\ \texttt{TP}[\![p]\!].$$

Proof

$$\begin{aligned}& (\lambda x, M[p \downarrow x])\ \texttt{TP}[\![p]\!] && \\ =\ & && \{\beta\text{-reduction}\} \\ & M[p \downarrow \texttt{TP}[\![p]\!]] && \\ =\ & && \{\text{Lemma 5.3.1}\} \\ & M &&\end{aligned}$$

□

Let-in Expressions

A let-in expression is semantically equivalent to a λ-abstraction followed by an application:

$$\texttt{let}\ x = M\ \texttt{in}\ N \approx (\lambda x.N)\ M$$

One possible way to assign a combinator structure to a let-in expression is, therefore, to use a structure equivalent to an abstraction followed by an application.

$$\text{Let}\frac{\Gamma, x \vdash N \sim a_1 \qquad \Gamma \vdash M \sim a_2}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N \sim @^n\ a_1\ a_2}$$

For treating recursive definitions, this structure can use the $\texttt{Y}$ fixpoint combinator:

$$\text{Let-rec}\frac{\Gamma, x \vdash N \sim a_1 \qquad \Gamma, x \vdash M \sim a_2}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N \sim @^n\ a_1\ (\langle \texttt{Y} \rangle^n\ a_2)}$$

This approach has a shortcoming: it does not differentiate any arguments to M from variables that are already in the context. This makes makes it difficult to reason about the parallelisation of let-in expressions. Parallelising M requires knowing how M handles its input, not how the variables in the context are used. For example, consider a list sorting function, where the comparison operator is in the context:

$$\begin{aligned}&\texttt{let cmp} = \lambda x.\lambda y.M \texttt{in}\\ &\texttt{let sort} = \lambda x.\texttt{case} \ldots \texttt{filter (le cmp } h\texttt{) } t \ldots\end{aligned}$$

The whole structure for this expression would be $@^n\ a_1\ a_2$, where a_1 would be the structure of `sort`, and a_2 the structure of `cmp`. Moreover, the structure for the filter subexpression, inside a_1, would have the following shape:

$$@^n\ |^i_j\ (@^n\ |^k_l\ |^r_s\ |^u_v)\ |^x_y$$

This contains too much information: the precise location in the context of every single definition and variable used in this expression. This makes handling these cases cumbersome. For parallelising function definitions, it is enough to know how it handles its inputs, leaving the other definitions as free variables:

$$\langle \texttt{filter} \rangle^n\ (\langle \texttt{le cmp} \rangle^n\ |^u_v)\ |^x_y$$

In the above structure, it is clear that the inputs $|^u_v$ and $|^x_y$ are passed to `filter`, `le` and `cmp`.

To simplify the treatment of `let-in` definitions, we define the new structure $\texttt{D}$. $\texttt{D}_x\ \Gamma\ a$ denotes the definition of x, in a context Γ, with structure a, where the variables in Γ may be used as free variables by a. For structure checking definitions, the context Γ is split in two parts: $\Delta; \Gamma$. The variables in Δ are not going to be used to obtain a point-free representation

of a program, i.e. they do not affect the potential parallelisations of the definition.

$$\text{Let}\ \frac{\Delta; \Gamma, x \vdash N \sim a_1 \qquad \Delta +\!\!+ \Gamma, x; [] \vdash M \sim a_2' \qquad a_2 = \mathtt{D}_x\ \Gamma\ a_2'}{\Delta; \Gamma \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N \sim @^n\ a_1\ a_2}$$

5.3.3 Definition | Recursive Definitions. *The structure* $\mathtt{D}_x\ \Gamma\ a$ *represents a definition, in context* Γ*, with structure* a*. This structure is equivalent to:*

$$\mathtt{D}_x\ \Gamma\ a \stackrel{\mathsf{def}}{=} [\overline{\Gamma}]\ (\mathtt{Y}\ ([x]\ a)).$$

Note that if $x \notin \mathrm{fv}(a)$*, then*

$$\mathtt{D}_x\ \Gamma\ a = [\Gamma]a.$$

The rules for variables need to be updated due to this new environment Δ. The rest of the associated structure rules remain the same. For variables in Γ, the rule does not change:

$$\text{Var}\ \frac{\Gamma(x) = k, n}{\Delta; \Gamma \vdash x \sim |_{n-k}^{k-1}}$$

The rule for variables occurring in Δ is as follows:

$$\text{GVar}\ \frac{x \notin \Gamma \qquad n = |\Gamma|}{\Delta, x; \Gamma \vdash x \sim \langle x \rangle^n}$$

User defined data constructors, and any function that is assumed will be introduced to Δ.

Inferring Structure-Annotated Types The compositional rules of the associated structure relation presented in this section can be implemented as a modification of a type-checking algorithm. The changes required to merge the typing rules in Figure 5.3 with the rules in Figure 5.7 mainly consist on taking into account the split context. This implies that inferring the associated applicative structures can be done during type-checking with little overhead. The properties of applicative expressions, derived from combinatory logic, can be applied to rewrite programs in HH, by using their associated structures. However, for this approach to work, there are two major requirements:

1. there must be formal guarantees that HH expressions and their associated structures are semantically equivalent; and,
2. it must be possible to do a reverse translation from applicative structures to HH.

The proof of soundness and the reverse translation are discussed in the next section.

5.4 Properties of the Associated Applicative Structure Relation

The associated structure relation provides a mechanism to derive that the code shown in Example 5.1.2 on page 148 has the associated structure shown in Figure 5.5a on page 161. Before simplifying and extracting a hylomorphism structure from Figure 5.5a on page 161, there must be strong static guarantees that this structure is semantically equivalent to the original HH term. For all expression M, if it has structure a in context $\Delta;\Gamma$, then a must be "functionally equivalent" M. For example, given a context $\Gamma = [y]$, and $f, g \in \Delta$, and

$$\Delta;\Gamma \vdash \lambda x.\ f\ (g\ x)\ y \sim \langle f\rangle^2\ (\langle g\rangle^2\ |^1_0)\ |^0_1,$$

the applicative expression $\langle f\rangle^2\ (\langle g\rangle^2\ |^1_0)\ |^0_1$ must be functionally equivalent to $\lambda x.\ f\ (g\ x)\ y$ in Γ. One approach is to derive the original term from $\langle f\rangle^2\ (\langle g\rangle^2\ |^1_0)\ |^0_1$, by using a small set of well-known rules: η and β conversion. The structure $\langle f\rangle^2\ (\langle g\rangle^2\ |^1_0)\ |^0_1$ can be η-expanded using two fresh variables y_1, y_2:

$$\lambda y_1\ y_2.\ (\langle f\rangle^2\ (\langle g\rangle^2\ |^1_0)\ |^0_1)\ y_1\ y_2$$

The body of the abstraction can be β-reduced according to the semantics of applicative expressions.

$$\begin{aligned}
&\lambda y_1\ y_2.\ (\langle f\rangle^1\ (\langle g\rangle^1\ (|^1_0\ y_1))\ (|^0_1\ y_1))\ y_2\\
&\rightsquigarrow \lambda y_1\ y_2.\ (\langle f\rangle^1\ (\langle g\rangle^1\ |^0_0)\ \langle y_1\rangle^1)\ y_2\\
&\rightsquigarrow \lambda y_1\ y_2.\ \langle f\rangle^0\ (\langle g\rangle^0\ (|^0_0\ y_2))\ (\langle y_1\rangle^1\ y_2)\\
&\rightsquigarrow \lambda y_1\ y_2.\ f\ (g\ y_2)\ \langle y_1\rangle^0\\
&\rightsquigarrow \lambda y_1\ y_2.\ f\ (g\ y_2)\ y_1
\end{aligned}$$

$$\text{GVar}\frac{x \notin \Gamma \qquad n = |\Gamma|}{\Delta, x; \Gamma \vdash x \sim \langle x \rangle^n} \qquad \text{Var}\frac{\Gamma(x) = k, n}{\Delta; \Gamma \vdash x \sim |_{n-k}^{k-1}}$$

$$\text{Abs}\frac{\Delta; \Gamma, x \vdash M \sim a}{\Delta; \Gamma \vdash \lambda x.M \sim a}$$

$$\text{App}\frac{\Delta; \Gamma \vdash M \sim a_1 \qquad \Delta; \Gamma \vdash N \sim a_2 \qquad n = |\Gamma|}{\Delta; \Gamma \vdash M\ N \sim @^n\ a_1\ a_2}$$

$$\text{Case}\frac{\begin{array}{c}\Delta; \Gamma \vdash M \sim a_M T \qquad\qquad n = |\Gamma| \\ \Delta; \Gamma \vdash_a p_1 \to N_1 \sim a_1 \quad \cdots \quad \Delta; \Gamma \vdash_a p_k \to N_k \sim a_k\end{array}}{\begin{array}{l}\Delta; \Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{p_1 \to N_1; \cdots; p_k \to N_k\} \\ \qquad \sim \langle \nabla_k \rangle^n\ a_1 \cdots a_k\ (\langle\{\texttt{out}(p_1, \ldots, p_k)\}\rangle^n\ a_M)\end{array}}$$

$$\text{Alt}\frac{\Gamma, x \vdash N[p \downarrow x] \sim a \qquad x \notin \Gamma \wedge x \notin \text{fv}(N)}{\Gamma \vdash_a p \to N \sim a}$$

$$\text{Let}\frac{\Delta; \Gamma, x \vdash N \sim a_1 \qquad \Delta +\!\!+ \Gamma, x; [] \vdash M \sim a_2' \qquad a_2 = \texttt{D}_x\ \Gamma\ a_2'}{\Delta; \Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N \sim @^n\ a_1\ a_2}$$

(a) Structure-checking Rules.

$$\begin{array}{ll}[x \downarrow y] & = [y/x] \\ [C\ p_1 \cdots p_n \downarrow y] & = [p_1 \downarrow \pi_1\ y] \cup \cdots \cup [p_n \downarrow \pi_n\ y]\end{array}$$

(b) Variable-pattern substitution.

$$\texttt{out}\ (p_1, \ldots, p_k) = \begin{cases} \lambda x.\texttt{case}\ x\ \texttt{of} \\ \quad p_1 \to \texttt{inj}_1\ \texttt{TP}[\![p_1]\!] \\ \quad \cdots \\ \quad p_k \to \texttt{inj}_k\ \texttt{TP}[\![p_k]\!] \end{cases}$$

$$\begin{array}{ll}\texttt{TP}[\![x]\!] & = x \\ \texttt{TP}[\![C\ p_1 \cdots p_n]\!] & = (\texttt{TP}[\![p_1]\!], \ldots, \texttt{TP}[\![p_n]\!])\end{array}$$

(c) The out Function.

Figure 5.7: Associated Structure Relation.

Applying α-renaming produces $\lambda y\ x.\ f\ (g\ x)\ y$, which is the original term. This is the approach that is followed for proving that HH expressions are semantically equivalent to their associated applicative structures. This approach has, as a side benefit, the definition of] a "reverse translation" that can be used to recover a λ-expression from an applicative expression.

5.4.1 Semantic Equivalence of λ-expressions

The meaning of being "functionally equivalent" that is used in this section is derived from the notion of $\beta\eta$-equality [Gha95], with an additional rule: the `case-case` rule. This rule is considered separately for simplicity, but can be derived using $\beta\eta$-equality.

5.4.1 Definition | `case`-`case` rule.

$$\begin{aligned}&\texttt{case}\ (\texttt{case}\ N\ \texttt{of}\ \{p_1 \to C_1\ N_1; \cdots p_k \to C_k\ N_k\})\ \texttt{of}\\ &\qquad\qquad \{C_1\ v_1 \to M_1; \cdots C_k\ v_k \to M_k\}\\ &= \texttt{case}\ N\ \texttt{of}\ \{p_1 \to (\lambda v_1.M_1)\ N_1; \cdots p_k \to (\lambda v_k.M_k)\ N_k\}\end{aligned}$$

This rule states that a case statement applied to the result of another case statement can be flattened into a single case statement. Note that the primitive *either* combinator, $\triangledown$, is equivalent to a case statement on a sum-type. This implies that `case-case` can be used to relate the primitive *either* combinator with case statements.

5.4.2 Definition | $\beta\eta$-equality of expressions. *The notation*

$$M \approx N$$

is used to represent that terms M and N are equal up to $\alpha/\beta/\eta$-conversion, and the `case-case` *rule.*

In other words, two terms M and N are equivalent, $M \approx N$, if M and N can be rewritten to the same form, by using only β, η and `case-case` equivalences. Since applicative expressions can be represented in the HH language itself, the same symbol $\approx$ will be uses to state the equivalences of HH terms and applicative structures as well.

5.4.2 Soundness

A simplified form of reverse translation from applicative expressions to λ-expressions is defined for each syntactic construct following a set of rules. The reason for defining this reverse translation is verifying that the associated structures are semantically equivalent to the corresponding HH expressions. The translation proceeds as follows:

1. If the structure represents a function taking n arguments, i.e. it is of the form $@^n$, the structure is η-expanded with $\lambda y_1 \cdots y_n$, where all y_i are free.

2. Any structure applied to a sufficient number of arguments is β-reduced.

3. Nested case expressions, or nested case and either combinator expressions are merged using `case-case`.

This is equivalent to first applying the translation in Definition 5.4.3 on page 173, followed by a number of β-reductions. The resulting term is, therefore semantically equivalent to the original term, up to $\beta\eta$ equality.

5.4.3 Definition | Reverse Translation. *The reverse translation of an applicative structure to a λ-expression is defined recursively:*

$$\lambda[\![@^n\ a_1\ a_2 \cdots a_k]\!] \stackrel{\text{def}}{=} \lambda\overline{y}.\lambda[\![a_1]\!]\ \overline{y}\ (\lambda[\![a_2]\!]\ \overline{y}) \cdots (\lambda[\![a_k]\!]\ \overline{y}) \quad (5.1)$$

$$\text{where } \overline{y} = y_1 \cdots y_n$$

$$\lambda[\![\langle\rangle^n]\!] \stackrel{\text{def}}{=} \lambda x\ x_1 \cdots x_n.x \quad (5.2)$$

$$\lambda[\![\mathtt{D}_x\ \Gamma\ a]\!] \stackrel{\text{def}}{=} \lambda\Gamma.\mathtt{Y}(\lambda x.\lambda[\![a]\!]) \quad (5.3)$$

$$\lambda[\![P]\!] \stackrel{\text{def}}{=} P \quad (5.4)$$

$$\lambda[\![[M]]\!] \stackrel{\text{def}}{=} M \quad (5.5)$$

The soundness of the associated structure relation is defined in terms of Definition 5.4.3. The soundness property states that if any two HH and applicative expressions, M and a, are associated in context Γ, this implies that both $\lambda\overline{\Gamma}.M$ and a are equivalent. Or, in other words a applied to the variables in Γ would yield a term that is equivalent to M.

5.4.1 Theorem | Soundness of Associated Structure Relation.
For all context $\Delta;\Gamma$, λ-expression M, and applicative structure a; if M has structure a, then the M and a are equivalent modulo β, η and α equivalence.

$$\Delta;\Gamma \vdash M \sim a \Longrightarrow \lambda\overline{\Gamma}.M \approx \lambda[\![a]\!].$$

Proof Sketch. By induction on the structure of the derivation $\Delta;\Gamma \vdash M \sim a$, and applying $\alpha/\beta/\eta$ and `case-case` equivalences between $\lambda\overline{\Gamma}.M$ and $\lambda[\![a]\!]$. The full proof can be found in Appendix B.

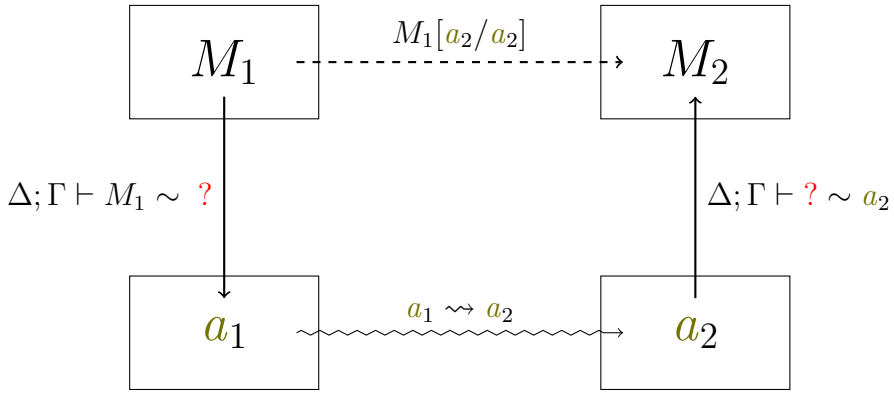


Figure 5.8: Structure Rewriting: A λ-expression M_1 can be rewritten to M_2, if M_1 has structure a_1, M_2 has structure a_2, and a_1 can be rewritten to a_2. This rewriting can be automated by solving the structure-inference problem for M_1, and the term-inference problem for a_2. The soundness of associated structures ensures that M_1 and M_2 are semantically equivalent, provided that the rewriting of a_1 into a_2 preserves the semantics.

5.4.3 Structure Rewriting

Note that the structure annotation rules in Figure 5.7 on page 171 are syntax-directed, which implies that rewriting M into a and viceversa can be reduced to an inference problem. Given $\Delta;\Gamma$ and M, solving the inference problem

$$\Delta;\Gamma \vdash M \sim ?,$$

results in the associated structure a. If a is shown to be equivalent to a', denoted by $a \cong a'$, then, using the same context $\Delta;\Gamma$, solving the inference

problem

$$\Delta; \Gamma \vdash ? \sim a',$$

provides a term M' that has associated structure a' in the context $\Delta; \Gamma$. Note that given the inference rules in Figure 5.7a on page 171, there may be many M' for a given a'. To illustrate this, consider the following `let` and λ expressions: `let x = M in N` | `(λ x. N) M` If $\Gamma, x \vdash N \sim a_2$, $\Gamma \vdash M \sim a_1$, and $x \notin M$, then both of those expressions have structure $@^n\ a_1\ a_2$. To solve these ambiguities, the *Let* and *Case* rules are prioritised, and the *App* rule is applied only if *Let* and *Case* fail. However, note that this choice is arbitrary, and any other would be equally valid, but would yield different, but equivalent HH terms for the same applicative structure.

Inferring HH expressions Some of the rewritings that will be applied to applicative structures in Section 5.5 yield valid applicative structures that have no associated HH expression. An example of this is the following rewriting:

$$@^{i+1}\ \langle a \rangle^1\ \langle b \rangle^1 \rightsquigarrow \langle @^i\ a\ b \rangle^1$$

Both terms are still semantically equivalent. If they are applied to $i+1$ elements, they return the same result:

$$@^{i+1}\ \langle a \rangle^1\ \langle b \rangle^1\ x\ x_1 \cdots x_i = a\ x_1 \cdots x_i (b\ x_1 \cdots x_i) = \langle @^i\ a\ b \rangle^1\ x\ x_1 \cdots x_i$$

However, the term $\langle @^i\ a\ b \rangle^1$ does not have a direct HH associated expression, by using only the structure checking rules in Figure 5.7. Section 5.5 discuss how to rewrite any term that does not match the applicative structure rules, so that a HH expression can always be found. However, for the sake of completeness, the following rule is introduced:

$$\frac{\Delta; \Gamma_1 \vdash M \sim a \qquad i = |a| = |\Gamma_1|}{\Delta; \Gamma_1, \Gamma_2 \vdash M\ \overline{\Gamma}_2 \sim a}$$

This rule states that if there are more elements in the context than required by applicative structure a, the HH expression is obtained using only the necessary elements from the context. Note that the opposite rule is not required. Whenever there are not enough elements in the context, these

can be generated using the *Abs* rule of Figure 5.7. As an example, consider again the term $\langle @^i\ a\ b\rangle^1$, in some context Γ. This term is notation for

$$@^0\ \langle\rangle^1\ (@^i\ a\ b)$$

Since this structure is an application that requires 0 elements from the context, the two sub-problems are generated:

$$\boxed{\Delta;[] \vdash ?_1 \sim \langle\rangle^1} \qquad \boxed{\Delta;[] \vdash ?_2 \sim (@^i\ a\ b)}$$

Assume that the term M is inferred for $@^i\ a\ b$. The term $\langle\rangle^1$ requires two elements in the context, so they are introduced using *Abs* twice:

$$\frac{\Delta;[x,y] \vdash ?_1 \sim \langle\rangle^1 \qquad x,y \text{ free}}{\Delta;[] \vdash \lambda xy.?_1 \sim \langle\rangle^1}$$

Since $\langle\rangle^1 = |^0_1$, the *Var* rule is applied:

$$\frac{\Delta;[x,y] \vdash x \sim \langle\rangle^1 \qquad x,y \text{ free}}{\Delta;[] \vdash \lambda xy.x \sim \langle\rangle^1}$$

The resulting term is $(\lambda xy.x)\ M\ \overline{\Gamma}$, which can be beta reduced to $(\lambda y.M)\ \overline{\Gamma}$. A final remark is that this term is correct, provided that y does not occur free in M. If y does not occur free in M, then the resulting term has the expected behaviour: it "drops" the first argument, and applies the rest of the elements in $\overline{\Gamma}$ to M. Since y was generated for applying the *Abs* rule, the inference algorithm ensures that y is a fresh free variable.

Since it is possible to convert between HH terms and $\mathcal{A}$ terms, then it is possible to rewrite HH terms according to rewritings defined at the $\mathcal{A}$ level. The notation $M_1[a_2/a_1]$ is introduced to denote the rewriting of M_1 to some M_2 with associated structure a_2, provided that the associated structure of M_1 is equivalent to a_2.

5.4.4 Definition | Structure Rewriting. *Given a context $\Delta;\Gamma$, rewriting a term M_1 with structure a_1 to a term with structure a_2, denoted*

$$M_1[a_2/a_1],$$

is defined as: (1) solving the structure-inference problem $\Delta;\Gamma \vdash M \sim\ ?$,

(2) rewriting a_1 to a_2, and finally (3) solving the term-inference problem $\Delta;\Gamma \vdash\ ?\ \sim a_2$, as illustrated in Figure 5.8 on page 174.

Remark The reverse translation used for the proof of soundness provides another mechanism for recovering λ-expressions from applicative expressions where the context is unknown. This mechanism consists of applying the reverse translation, followed by any possible $\beta\eta$ and `case-case` reductions. For any applicative structure a, the equivalent λ-term arising from structure a is known as $\lambda[\![a]\!]_{\beta\eta}$. However, note that it is not necessarily true that

$$\Delta;\Gamma \vdash \lambda[\![a]\!]_{\beta\eta} \sim a.$$

The reason for this is that the structure a may be derived from terms that contain $\beta\eta$-redexes, which $\lambda[\![a]\!]_{\beta\eta}$ will optimise away. For example, consider the following expression and associated structure:

$$\lambda x.(\lambda y.y\ x)\ f \sim @^1\ (@^2\ |^1_0\ |^0_1)\ \langle f\rangle^1.$$

The equivalent λ-expression of the associated structure is:

$$\begin{aligned}
&\lambda[\![@^1\ (@^2\ |^1_0\ |^0_1)\ \langle f\rangle^1]\!]_{\beta\eta}\\
&\quad\approx\quad \{\ \text{By (5.1)}\ \}\\
&\lambda x_1.(\lambda y_1\ y_2.((\lambda a_1\ a_2.a_2)\ y_1\ y_2)\ ((\lambda b_1\ b_2.b_1)\ y_1\ y_2))\ x_1\ ((\lambda z_1.f)\ x_1)\\
&\quad\approx\quad \{\ \text{By } \beta\text{-reduction, until } \beta\text{-normal form.}\ \}\\
&f
\end{aligned}$$

It is clear that the associated structure of f is different from the associated structure of $\lambda x.(\lambda y.y\ x)\ f$, i.e. $\Delta;\Gamma \not\vdash f \sim @^1\ (@^2\ |^1_0\ |^0_1)\ \langle f\rangle^1$.

5.5 Converting $\mathcal{A}$ to Hylo

This section presents a mechanism for extracting hylomorphisms from definition structures `D`. In the applicative form, a hylomorphism structure can be derived, provided that the applicative expression is in the right form. This form is what we call a *flat* representation. Intuitively, a flat applicative structure is one that corresponds to a function with no let-in definitions and lambda-abstractions. This implies that all subexpressions occur at the

same "level", i.e. under application nodes within the same context size n, $@^n$. An example of flat structure and hylomorphism extraction is shown below. Consider the quicksort function definition in with the quicksort function in Listing 5.5. The associated structure of the `qsort` function is shown below:

$$\begin{aligned}\mathtt{D}\ \Gamma\ (\langle\nabla_2\rangle^2\ \ & \langle\mathtt{Nil}\rangle^3 \\ & (\langle\mathtt{app}\rangle^3\ (\langle\mathtt{qsort}\rangle^3\ \ldots)\ (\langle\mathtt{Cons}\rangle^3\ (\langle\pi_1\rangle^3\ |_0^2)\ (\langle\mathtt{qsort}\rangle^3\ \ldots))) \\ & (\langle\mathtt{out_L}\rangle^2\ |_0^1)\end{aligned}$$

We omit the substructures corresponding to `le d x xs` and `gt d x xs` since they will not affect the hylomorphism derivation. Note that in a flat structure, all $\langle a\rangle^i$ nodes occur at the same level i, with the exception of the case branches, that add an element to the context, and therefore occur at level $i+1$.

This structure can be translated to a hylomorphism by choosing as the divide function one that replaces any "either" node by a map node on sum types, and any application node that does not correspond to the recursive call by a tupling operation. The recursive calls are dropped, since they are going to be handled by the hylomorphism. In the quicksort case, the divide function corresponds to the following structure:

$$\begin{aligned}\mathtt{qdiv} = (\langle+_2\rangle^2\ \ & \langle\&_0\rangle^3 \\ & (\langle\&_2\rangle^3\ (\langle\&_2\rangle^3\ \ldots)\ (\langle\&_2\rangle^3\ (\langle\pi_1\rangle^3\ |_0^2)\ (\langle\&_2\rangle^3\ \ldots))) \\ & (\langle\mathtt{out_L}\rangle^2\ |_0^1)\end{aligned}$$

This structure represents a function that first does pattern matching according to the original function definition, in this case represented by $\mathtt{out}_L$, and then creates a tuple that contains the arguments for the recursive calls. Note that the subexpression $\langle\pi_1\rangle^3\ |_0^2$ is left unmodified in the divide function. The reason for this is purely arbitrary: any application of a primitive function to an expression is left unmodified in the divide part. However, a structure $\langle\ \Delta_1\ \rangle^3\ |_0^2 = |_0^2$ would have been equally valid, provided that the "combine" part applies π_1 to this.

The combine part is obtained by applying *uncurried* versions of the functions that were replaced by the tuple constructor, $\&_i$, in the divide

Listing 5.5 HH Quicksort

```
1 qsort : Ord a → [a] → [a]
2 qsort = λ d xs.
3             case xs of {
4               Nil         -> Nil;
5               (Cons x xs) -> app
6                                (qsort d (le d x xs))
7                                (Cons x (qsort d (gt d x xs))) }
```

function:

$$\texttt{qcomb} = \nabla_2 \;\; \langle \texttt{Nil} \rangle^1 \;\; (\langle \texttt{app} \rangle^1 \; \pi_1 \; (\langle \texttt{Cons} \rangle^1 \; \pi_{2,1} \; \pi_{2,2}))$$

Finally, a traversal on the divide function can be used to obtain the corresponding functor. Occurrences of $+$ correspond to a sum type, and occurrences of the recursive call are replaced by a type variable A, and for each other subexpression in the resulting structure, an extra type variable is added to the functor argument list. In the case of the quicksort, this results in the bi-functor:

$$T \; B \; A = 1 + A \times (B \times A)$$

This results in the expected hylomorphism, which can be parallelised using the StA framework:

$$\text{HYLO}_{T\;A} \; \texttt{qcomb} \; \texttt{qdiv}$$

We write $\updownarrow_x$ for this translation, where the subscript x is the recursive call. However, note that the structures generated by this transformation only derive high-level hylomorphism definitions. To expose more structure, it will be convenient to expose function compositions. Recall that the `B` combinator is $a \circ b = \langle a \rangle^1 \; b$. The following transformation *lifts* occurrences of $\langle\rangle^1$ to expose (some) function compositions:

$$\langle a \rangle^i \; \langle b \rangle^j \; \langle c \rangle^k \approx \langle \langle a \rangle^{i - \min\; i\; j\; k} \; \langle b \rangle^{j - \min\; i\; j\; k} \; \langle c \rangle^{k - \min\; i\; j\; k} \rangle^{\min\; i\; j\; k}$$

This transformation is applied to the resulting hylomorphism to expose function compositions. When this results in a composition of *map* operations, these will be used as potential instantiations of *pipeline* skeletons.

For example, suppose that after a number of transformations the following code is derived:

$$\langle \text{MAP}_T \rangle^1 \ (@^2 \ (\ @^2 \ \langle f \rangle^2 \ \langle g \rangle^1) \ |_0^1)$$

By lifting applications of the const function, we end up with the following:

$$\langle \text{MAP}_T \ (\langle \langle f \rangle^1 \ g \rangle^1 \ \mathtt{I}) \rangle^1 = \langle \text{MAP}_T \ (f \circ g) \rangle^1$$

This structure can be parallelised using a combination of farms and pipelines, as we showed in Chapter 3. It remains, however, how the instances of MAP_T are derived.

5.5.1 Deriving Hylomorphisms

A proposition of the form $\mathtt{D}_x \ \Gamma \ a \updownarrow_x h$ states that the definition x is equivalent to the composition of hylomorphisms h. The relation $\updownarrow_x$ assumes that that the structure a is in a precise shape, and it is defined in three parts. The recursive structure of the hylomorphism, i.e. the functor F that defines the structure of the hylomorphism, is derived by using the ϕ relation that is shown in Figure 5.9a. The superscript of the ϕ relation is used to ensure that the structure is in this precise shape, called *flat*, i.e. that no elements are introduced to the context, except by the either combinator. It is also important to note that the sums and products of (multi-) functors are used, here represented by $\hat{\times}$ and $\hat{+}$. These operations are different to the standard definitions in the literature. Given two multifunctors F and G:

$$\begin{aligned}
&F = \Lambda A_1 \cdots A_n \ B, \ T_1 \\
&G = \Lambda C_1 \cdots C_m \ D, \ T_2 \\
&F \hat{\times} G = \Lambda A_1 \cdots A_n \ C_1 \cdots C_m \ B, \ F \ A_1 \cdots A_n B \times G \ C_1 \cdots C_m \ B
\end{aligned}$$

The intuition behind these definitions is that the arguments A_i and C_j will correspond to the non-recursive parts of the function definition, and the last argument, B in F and D in G represent the recursive calls.

Extracting the divide and combine parts from a definition follow the same pattern, and it is shown in Figures 5.9b and 5.9c. Note, however, that the rules for extracting a divide function from an applicative structure derive functions that may take multiple arguments. In the quicksort example, this

function has type:

$$\texttt{qdiv} : \texttt{Ord}\ A \to \texttt{List}\ A \to F\ A\ (\texttt{Ord}\ A, \texttt{List}\ A)$$

An *uncurrying* transformation rewrites `div` to the necessary shape. We write $\texttt{U}$ for the uncurrying transformation, and it is explained in Section 5.5.2.

$$\texttt{U}_2\ \texttt{qdiv} : \texttt{Ord}\ A \times \texttt{List}\ A \to F\ A\ (\texttt{Ord}\ A, \texttt{List}\ A)$$

Together, the rules in Figure 5.9, and the uncurrying transformation allow us to define the hylomorphism extraction rule.

5.5.1 Definition | Hylomorphism Equivalent. *We say that a definition* $\texttt{D}_x\ \Gamma\ a$ *has a hylomorphism equivalent* $\textsc{hylo}_F\ b\ c$*, written*

$$\texttt{D}\ \Gamma\ a \updownarrow_x \textsc{hylo}_F\ b\ c,$$

if $\phi_x(a, F)$*,* $[] \vdash a \downarrow_x^n b$*,* $a \uparrow_x^n c'$ *and* $c = \texttt{U}_{|a|}$*, where the subscript* $|a|$ *is the number of arguments that receives the definition* x*.*

Note that due to the uncurrying transformation, if a definition $\texttt{D}_x\ \Gamma\ da$ has a hylomorphism equivalent h, then they curried/uncurried versions are equivalent:

$$\texttt{U}_n\ (\texttt{D}_x\ \Gamma\ a) \approx h \qquad \wedge \qquad \texttt{D}_x\ \Gamma\ a \approx \texttt{C}_n\ h$$

The rules in Figure 5.9, however, are not powerful enough to handle the FFT working example in Example 5.1.2 on page 148:

```
fft : List Complex → List Complex
      ∼ REDUCE_k _ ∘ _
fft = λ xs. case xs of
              [x] → [x]
              xs  → let n    = length xs
                        xs'  = halves xs
                    in comb (genWs n) (fft (π1 xs'))
                                      (fft (π1 xs'))
```

Note that on its applicative structure, shown in Figure 5.5a on page 161, the following pattern occurs:

$$@^n\ (@^{n+1}\ \ldots)\ a$$

These structures arise from the structures of let-in expressions, and they need to be treated before deriving the hylomorphism equivalent. This, however, can be handled by extending the hylomorphism equivalent rules, and the currying/uncurrying transformation.

5.5.2 Currying-Uncurrying

Currying and uncurrying are well-known transformations that consist on converting a function that takes an input tuple with n arguments into a *higher-order function* that takes n arguments, and viceversa:

$$\begin{aligned}
&\mathsf{curry}_n : ((A_1, \ldots, A_n) \to B) \to A_1 \to \cdots \to A_n \to B \\
&\mathsf{curry}_n\ f\ = \lambda x_1 \cdots x_n . f\ (x_1, \ldots, x_n) \\
&\mathsf{uncurry}_n : (A_1 \to \cdots \to A_n \to B) \to (A_1, \ldots, A_n) \to B \\
&\mathsf{uncurry}_n\ f\ = \lambda x . f\ (\pi_1\ x) \cdots (\pi_n\ x)
\end{aligned}$$

The currying and uncurrying transformations can be derived from their applicative structures:

$$\begin{aligned}
&\mathsf{curry}_n \quad f\ \sim \langle f \rangle^n\ (\langle \&_n \rangle^n\ |_{n-1}^0 \cdots |_0^{n-1}) \\
&\mathsf{uncurry}_n \quad f\ \sim \langle f \rangle^1\ \pi_1 \cdots \pi_n
\end{aligned}$$

It is easy to show using equational reasoning that curry/uncurry structures are indeed inverses:

5.5.1 Lemma | Curry/Uncurry inverses. *The associated structures of* $\mathtt{U}_i$ *and* $\mathtt{C}_i$ *are mutual inverses:*

$$\mathtt{U}_i\ (\mathtt{C}_i\ a) \approx a \qquad \mathtt{C}_i\ (\mathtt{U}_i\ a) \approx a.$$

Currying and uncurrying transformations have useful properties on applicative structures. Since applicative structures are functions that pass a context, or extract elements from the context, currying and uncurrying transformations can be defined in a compositional way for each syntactic construct in $\mathcal{A}$.

5.5.2 Definition | (Un-)Currying Transformation. *Given the structures* a, $a_1, \ldots, a_j$, *and natural number* n *such that* $n \leq i$, *the currying/uncurrying*

$$\frac{x \notin a \qquad a \not\approx \langle a' \rangle^n}{\phi^n_x(a, \Lambda AB, A)}$$

$$\frac{x \notin a_n \qquad n \in [1 \ldots j]}{\phi^n_x(\langle x \rangle^n \ a_1 \cdots a_j, \Lambda AB, B)}$$

$$\frac{x \in a_m \qquad m \in [1 \ldots j] \qquad \phi^n_x(a_1, F_1) \qquad \cdots \qquad \phi^n_x(a_j, F_j)}{\phi^n_x(\langle a \rangle^n \ a_1 \cdots a_j, F_1 \hat{\times} \cdots \hat{\times} F_j)}$$

$$\frac{x \notin a_{j+1} \qquad x \in a_m \qquad m \in [1..j] \qquad \phi^{n+1}_x(a_1, F_1) \qquad \cdots \qquad \phi^{n+1}_x(a_j, F_j)}{\phi^n_x(\langle \nabla_j \rangle^n \ a_1 \cdots a_j \ a_{j+1}, F_1 \hat{+} \cdots \hat{+} F_j)}$$

(a) Deriving the structure of the hylomorphism.

$$\frac{x \notin a \qquad a \not\approx \langle a' \rangle^n}{a \uparrow^n_x a}$$

$$\frac{x \notin a_n \qquad n \in [1 \ldots j]}{\langle x \rangle^n \ a_1 \cdots a_j \uparrow^n_x \langle \&_j \rangle^n \ a_1 \cdots a_j}$$

$$\frac{x \in a_m \qquad m \in [1 \ldots j] \qquad a_1 \uparrow^n_x b_1 \qquad \cdots \qquad a_j \uparrow^n_x b_j}{\langle a \rangle^n \ a_1 \cdots a_j \uparrow^n_x \langle \& \rangle^n \ b_1 \cdots b_j}$$

$$\frac{x \notin a_{j+1} \qquad x \in a_m \qquad m \in [1..j] \qquad a_1 \uparrow^{n+1}_x b_1 \qquad \cdots \qquad a_j \uparrow^{n+1}_x b_j}{\langle \nabla_j \rangle^n \ a_1 \cdots a_j \ a_{j+1} \uparrow^n_x \langle + _j \rangle^n \ b_1 \cdots b_j \ a_{j+1}}$$

(b) Deriving the divide part of a hylomorphism.

$$\frac{x \notin a \qquad a \not\approx \langle a' \rangle^n}{m \vdash a \downarrow^n_x \pi_m}$$

$$\frac{x \notin a_n \qquad n \in [1 \ldots j]}{m \vdash \langle x \rangle^n \ a_1 \cdots a_j \downarrow^n_x \pi_m}$$

$$\frac{x \in a_m \qquad m \in [1 \ldots j] \qquad 1, m \vdash a_1 \downarrow^n_x b_1 \qquad \cdots \qquad j, m \vdash a_j \downarrow^n_x b_j}{m \vdash \langle a \rangle^n \ a_1 \cdots a_j \downarrow^n_x \langle a \rangle^n \ b_1 \cdots b_j}$$

$$\frac{x \notin a_{j+1} \qquad x \in a_r \qquad r \in [1..j] \qquad m \vdash a_1 \downarrow^{n+1}_x b_1 \qquad \cdots \qquad m \vdash a_j \downarrow^{n+1}_x b_j}{m \vdash \langle \nabla_j \rangle^n \ a_1 \cdots a_j \ a_{j+1} \downarrow^n_x \langle \nabla_j \rangle^n \ b_1 \cdots b_j}$$

(c) Deriving the combine part of a hylomorphism.

Figure 5.9: Hylomorphism-derivation Rules.

transformations are defined by repeatedly applying the following properties:

$$\mathtt{U}_n\ (\langle a\rangle^i\ a_1 \cdots a_j) \approx \langle a\rangle^{i-n+1}\ (\mathtt{U}_n\ a_1) \cdots (\mathtt{U}_n\ a_j)$$

$$\mathtt{C}_n\ (\langle a\rangle^{i-n+1}\ a_1 \cdots a_j) \approx \langle a\rangle^i\ (\mathtt{C}_n\ a_1) \cdots (\mathtt{C}_n\ a_j)$$

Finally, uncurrying variables is done by replacing them by the equivalent tuple-projections. Given natural numbers n, i and j, if $n = i + j + 1$, then:

$$\mathtt{U}_n\ |^i_j \ \approx\ \pi_{i+1} \qquad |^i_j \ \approx\ \mathtt{C}_n\ \pi_{i+1}$$

As examples of this transformation, consider first the variable case. The associated structure of a variable that occurs in a context with four elements is $|^2_1$, with an uncurried version π_3:

$$\begin{array}{ll} |^2_1 \quad x\ y\ z\ t = \langle\langle\rangle^1\rangle^2\ x\ y\ z\ t = \langle\langle\rangle^1\rangle^1\ y\ z\ t = \langle\langle\rangle^1\rangle^0\ z\ t = \langle z\rangle^1\ t = \langle z\rangle^0 & = z \\ \pi_3 \quad (x, y, z, t) & = z \end{array}$$

For an application node, in a context of for elements, $\mathtt{U}_4$ would rewrite the following expressions to their uncurried equivalents:

$$\mathtt{U}_4\ (@^4\ |^2_1\ |^0_3\ |^1_2) = @^1\ \pi_3\ \pi_1\ \pi_2$$

It is easy to check that both expressions are equivalent. Given a sequence of variables $\overline{x} = x_1\ x_2\ x_3\ x_4$,

$$\begin{array}{ll} @^0\ (@^4\ |^2_1\ |^0_3\ |^1_2)\ \overline{x} = @^0\ (|^2_1\ \overline{x})\ (|^0_3\ \overline{x})\ (|^1_2\ \overline{x}) & = x_3\ x_1\ x_1 \\ @^0\ (@^1\ \pi_3\ \pi_1\ \pi_2)\ \overline{x} & = x_3\ x_1\ x_1 \end{array}$$

The uncurrying transformation applied to the associated structures of case expressions need to be treated slightly differently. The reason for this is that the ∇_i combinator introduces one variable in the context, with the result of the pattern matching on a value of primitive sum type. Given two structures a and b, the uncurrying transformation would proceed as follows:

$$\mathtt{U}_2\ (\langle\nabla_2\rangle^2\ a\ b\ c) = \langle\nabla_2\rangle^1\ (\mathtt{U}_2\ da)\ (\mathtt{U}\ b)\ (\mathtt{U}\ c).$$

However, this would leave in the branches of this expression two substructures that correspond to functions of (at least) two arguments: the input to

the combinator, and the result of the "out" function c. As a special case of the uncurrying transformation, either combinators are treated as follows:

$$\begin{aligned}&\mathtt{U}_n\ \langle\nabla_i\rangle^j\ a_1 \cdots\ a_i\ b\\ &= \langle\langle\nabla_i\rangle^{j-n}\ (\mathtt{U}_{n+1}\ a_1) \cdots\ (\mathtt{U}_{n+1}\ a_i)\rangle^1\ (\mathtt{distr}_n^i\ \circ\ (\pi_1 \times \cdots \times \pi_n \times \mathtt{U}_n\ b))\end{aligned}$$

Note that the whole branches of the either combinator are wrapped under a const combinator and that the resulting either combinator is $\langle\nabla\rangle^{j-n}$. The reason for this is that the environment is extended using the `distrib` function, so the first argument can be safely dropped. This has more opportunities of exposing function compositions. We will see this in the quicksort example, and later in the final FFT example.

The function $\mathtt{distr}_n^i$ takes a tuple of $n+1$ elements as input, where the last element is of sum type, with i branches, and distributes the product over the sum, as indicated by the type:

$$\mathtt{distr}_j^i : A_1 \times \cdots \times A_j \times (B_1 + \cdots + B_i) \to A_1 \times \cdots \times A_j \times B_1 + \cdots + A_1 \times \cdots \times A_j \times B_i.$$

As an example of the uncurrying transformation, consider again the quicksort example. Since quicksort takes two arguments, the `Ord` dictionary for comparing elements of the list, and the input list, we will apply $\mathtt{U}_2$. The top-level structure is:

$$\begin{aligned}&\mathtt{U}_2\ (\langle\nabla_2\rangle^2\ \mathtt{anil}\ \mathtt{acons}\ (\langle\mathtt{out_L}\rangle^2\ |_0^1))\\ \rightsquigarrow\ &\langle\nabla_2\ (\mathtt{U}_3\ \mathtt{anil})\ (\mathtt{U}_3\ \mathtt{acons})\rangle^1\ (\mathtt{distr}_2^2\ \circ\ (\pi_1 \times \pi_2 \times \langle\mathtt{out_L}\rangle^1\ \pi_2))\\ \rightsquigarrow\ &\nabla_2\ \mathtt{anil}'\ \mathtt{acons}'\ \circ\ (\mathtt{distr}_2^2\ \circ\ (\pi_1 \times \pi_2 \times \langle\mathtt{out_L}\rangle^1\ \pi_2))\end{aligned}$$

The substructure `qnil` is uncurried as follows, into `qnil'`:

$$\mathtt{U}_3\ \langle\&_0\rangle^3 \rightsquigarrow \langle\&_0\rangle^1$$

The substructure `qcons` is uncurried to `qcons'` as is shown below:

$$\begin{aligned}&\mathtt{U}_3\ (\langle\&_2\rangle^3\ (\langle\&_2\rangle^3\ \ldots)\ (\langle\&_2\rangle^3\ (\langle\pi_1\rangle^3\ |_0^2)\ (\langle\&_2\rangle^3\ \ldots)))\\ \rightsquigarrow\ &\langle\&_2\rangle^1\ (\langle\&_2\rangle^1\ \ldots)\ (\langle\&_2\rangle^1\ (\langle\pi_1\rangle^1\ \pi_3)\ (\langle\&_2\rangle^1\ \ldots))\\ \rightsquigarrow\ &\Delta_2\ \ (\Delta_2\ \ldots)\ (\Delta_2\ \ (\pi_1 \circ \pi_3)\ (\Delta_2\ \ldots))\end{aligned}$$

The final structure using the curried and uncurried versions of the divide and combine functions is a valid Hylo structure, and can therefore be parallelised using the rules of the StA framework that were defined in Chapter 3.

5.5.3 Flattening Transformation

As we mentioned previously in this section, these transformations are not enough to handle many cases, as it was illustrated by the case of the Cooley-Tukey FFT algorithm. The structures that cannot be handled have the following shape:

$$@^i\ (@^{i+1}\ a_1 \cdots a_n)\ b.$$

A pre-processing step is carried to these structures to ensure that they are in the correct shape. We quickly cover these transformations as rewriting steps. Note that in order to reverse these translations, we must keep a list of all the rewriting steps done to a substructure.

Applicative Eta-expansion. The equivalent to eta-expansion can be performed to applicative expressions as we explain below. Particularly, whenever an expression such as

$$(\lambda x_1 \cdots x_n.M)\ N$$

is used, an applicative structure of the form $@^i\ (@^{i+n}\ \ldots)\ldots$ will be generated. However, note that since abstraction is not used in applicative expressions, any $@^i$ can be safely expanded to $@^{i+k}$ provided that the inner variables are modified. The transformation proceeds as follows:

$$@^i\ a_1 \cdots a_n \rightsquigarrow @^{i+k}\ (\langle\langle\rangle^k\rangle^i\ a_1) \cdots (\langle\langle\rangle^k\rangle^i\ a_n)\ |^i_{k-1} \cdots |^{i+k-1}_0$$

Note that the expression $\langle\langle\rangle^k\rangle^i$ can be "pushed down" in the structure of a_i. When an $\langle\rangle^i$ is found, it is converted to $\langle\rangle^{i+k}$. For variables, $|^n_m$ such that $n + m = i - 1$, they are converted to $|^n_{m+k}$. This transformation is applied systematically until all structures have the shape

$$@^i\ (@^{i+n}\ \ldots)\ a_1 \cdots a_n$$

Const-Lifting and Structure-Swapping So far, we have assumed that $\langle\langle\rangle^i\rangle^j\ a$ can be applied to arbitrary structures. The relation that defines this transformation is shown in Figure 5.10, and shows precisely to which structures it can be applied. In this figure $a \Uparrow^n_m b$ states that $b \approx \langle\langle\rangle^m\rangle^n\ a$, and adds the necessary conditions where this can be applied. Note that if a

$$\frac{i+j+1=n}{|^i_j \Uparrow^n_m |^i_{j+m}} \qquad \frac{n \leq i \leq m}{\langle a \rangle^i \Uparrow^n_m \langle a \rangle^{n-i+m}}$$

$$\frac{a_1 \Uparrow^n_m a'_1 \quad \cdots a_i \Uparrow^n_m a'_i}{@^n\ a_1 \cdots a_i \Uparrow^n_m @^{n+m}\ a'_1 \cdots a'_i}$$

Figure 5.10: Const-lifting relation.

structure is equivalent to some $\langle\langle\rangle^j\rangle^i\ a$, then it implies that it will take the first i arguments, and then ignore the rest j arguments. This fact can be used to define a *structure swapping* transformation. Given the structures a, b_1, b_2 and c, if $b_1 \Uparrow^i_j b_2$, then

$$@^i(@^{i+j}\ a\ b_1)\ c \rightsquigarrow @^i a\ c\ b_2.$$

This can be generalised to multiple arguments to $@$ and $\langle a \rangle^i$ structures. This rewriting provides us with a mechanism to *lift* structures to group them in levels that do not depend on each other (e.g. c and b_2 in the example above).

Function inlining. Sometimes, the recursive calls will not occur fully applied in the definition of a function. Take for example the quicksort definition below.

```
qsort : Ord a → [a] → [a]
qsort = λ d xs.
          let qd = qsort d
          in case xs of {
               Nil         -> Nil;
               (Cons x xs) -> app
                                (qd (le d x xs))
                                (Cons x (qd (gt d x xs))) }
```

In this definition, the recursive call occurs partially applied. These cases cannot be treated easily with the hylomorphism derivation rules that we defined. However, note that these cases will result in a structure of the following form:

$$@^2\ a\ (\langle \mathtt{qd} \rangle^2\ |^0_1).$$

In these cases, however, note that $\langle \texttt{qd} \rangle^2$ can be inlined further down, in an expression equivalent to:

```
qsort : Ord a → [a] → [a]
qsort = λ d xs.
          let qd_arg = d
              ...
```

Where `qd` is replaced by `qsort qd_arg` in the remaining expression. Note that although keeping the let definition and the argument is unnecessary in this case, this is not true for the general case, since some useful computation might be done to the partially applied argument. This transformation can be described in the applicative framework as follows. If $b \Uparrow_n^i b'$, and $c[n \mapsto a]$ replaces the structure where variable n occurs by structure a, with the necessary applications of the const combinator in order to keep the application context sizes consistent.

$$@^i\ b\ (\langle x \rangle^i\ a_1 \cdots a_n) \rightsquigarrow @^i((\langle\langle\rangle^n\rangle^i\ b)[i+1 \mapsto \langle x \rangle^{i+n}\ |_{n-1}^i \cdots |_0^{i+n-1}])\ a_1 \cdots a_n$$

Collectively, these transformations perform a *flattening* transformation that broadens the scope of the hylomorphism derivation mechanism. The following rules for deriving divide and combine functions for hylomorphisms apply to cases where the structures are not flat:

$$\frac{x \notin a_s \quad \forall s \in [1 \cdots m] \quad x \in b_r \qquad b_1 \uparrow_x^{i+n} b'_1 \quad \cdots \quad b_n \uparrow_x^{i+n} b'_n}{@^i\ (\langle a \rangle^{i+n}\ a_1 \cdots a_m)\ b_1 \cdots b_n \uparrow_x^i \langle \&_n \rangle^i\ |_{i-1}^0 \cdots |_0^{i-1}\ b'_1 \cdots b'_n}$$

$$\frac{x \in a_s \quad \exists s \in [1 \cdots m] \quad x \notin b_r \quad \forall r \in [1 \cdots n] \qquad a_1 \uparrow_x^{i+n} a'_1 \quad \cdots \quad a_m \uparrow_x^{i+n} a'_m}{@^i\ (\langle a \rangle^{i+n}\ a_1 \cdots a_m)\ b_1 \cdots b_n \uparrow_x @^i\ (\langle \&_m \rangle^{i+n}\ a'_1 \cdots a'_m)\ b_1 \cdots b_n}$$

$$\frac{x \notin a_s \quad \forall s \in [1 \cdots m] \quad x \in b_r \qquad 1, c \vdash b_1 \downarrow_x^{i+n} b'_1 \quad \cdots \quad n, c \vdash b_n \downarrow_x^{i+n} b'_n}{c \vdash @^i\ (\langle a \rangle^{j+n}\ a_1 \cdots a_m)\ b_1 \cdots b_n \downarrow_x^i \mathtt{U}_{j+n}\ (\langle a \rangle^{j+n}\ a_1 \cdots a_m)}$$

$$\frac{x \in a_s \quad \exists s \in [1 \cdots m] \quad x \notin b_r \quad \forall r \in [1 \cdots n] \qquad 1, c \vdash a_1 \downarrow_x^{i+n} a'_1 \quad \cdots \quad 1, c \vdash a_m \uparrow_x a'_m}{c \vdash @^i\ (\langle a \rangle^{j+n}\ a_1 \cdots a_m)\ b_1 \cdots b_n \downarrow_x^i \langle a \rangle^1\ \pi_1 \cdots \pi_m}$$

The next section shows examples of how the result of applying these rules on the examples that were used in Chapter 3. We will show that, by using this extension of the structure-annotated arrows with applicative structures, parallel structures can be safely introduced to programs written in HH.

5.6 Examples

This section illustrates the extension of the StA framework with applicative structures and HH expression, by using a number of examples. The first example is the Cooley-Tukey FFT algorithm. Then, we will review the examples shown in Chapter 3, and show how their implementation in HH lead to similar parallelisations. Since we have already illustrated how these examples can be parallelised in the StA framework, when implemented directly as hylomorphisms, we will just show the derived HYLO structures for them. Note that re-running these experiments would require a full back-end for HH, which is not yet implemented. The examples shown below are implementations using code that tries to mimic idiomatic Haskell code.

Cooley-Tukey FFT Algorithm

We review now the Cooley-Tukey FFT algorithm. Recall that the implementation in HH is as follows:

```
fft : List Complex → List Complex
      ∼ REDUCE_k _ ∘ _
fft = λ xs. case xs of
              [x] → [x]
              xs  → let n   = length xs
                        xs' = halves xs
                    in comb (genWs n) (fft (π1 xs'))
                                      (fft (π1 xs'))
```

The `fft` associated structure is shown below:

$$\begin{array}{l}\texttt{fft} \sim \mathtt{D}_{\texttt{fft}}\ \Gamma\ (@^1\quad (\langle\nabla_2\rangle^1\quad (\langle\texttt{Cons}\rangle^2\ |_0^1\ \langle\texttt{Nil}\rangle^2)\\ \qquad\qquad\qquad\qquad\qquad\qquad (@^2\quad (@^3\quad (\langle\texttt{comb}\rangle^4\quad (\langle\texttt{genWs}\rangle^4\ |_1^2)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\langle\texttt{fft}\rangle^4\ (\langle\pi_1\rangle^4\ |_0^3))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\langle\texttt{fft}\rangle^4\ (\langle\pi_2\rangle^4\ |_0^3)))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\langle\texttt{halves}\rangle^3\ |_1^1))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\langle\texttt{length}\rangle^2\ |_0^1)))\\ \qquad\qquad\qquad\qquad\qquad\quad (\langle\texttt{out}([\texttt{x}],\texttt{xs})\rangle^1\ \mathtt{I}))\end{array}$$

Note that all calls to `fft` occur fully applied, under a context of size four. This implies that the extended rules for hylomorphism derivation can be applied. First, the hylomorphism structure derivation generates the functor that represents the structure of the hylomorphism:

$$\phi(\mathtt{D}_{\texttt{fft}}\ldots, F = \Lambda A\ B\ C,\ A + B \times C \times C)$$

The type-checking algorithm can derive that this structure must be sectioned with the types

F **`(List Complex) (List Complex)`**

for generating the hylomorphism structure. This corresponds to the types of the non-recursive parts of the function definition, namely `[x]` and `genWs n`. Then, the divide part of the hylomorphism is derived. The derived structure is as follows:

$$\begin{array}{l}\mathtt{D}_{\texttt{fft}} \uparrow_{\texttt{fft}} \sim @^1\quad (\langle+_2\rangle^1\quad (\langle\texttt{Cons}\rangle^2\ |_0^1\ \langle\texttt{Nil}\rangle^2)\\ \qquad\qquad\qquad\qquad\qquad\qquad (@^2\quad (@^3\quad (\langle\&_3\rangle^4\quad (\langle\texttt{genWs}\rangle^4\ |_1^2)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\langle\pi_1\rangle^4\ |_0^3)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\langle\pi_2\rangle^4\ |_0^3))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\langle\texttt{halves}\rangle^3\ |_1^1))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\langle\texttt{length}\rangle^2\ |_0^1)))\\ \qquad\qquad\qquad\qquad\quad \texttt{out}([\texttt{x}],\texttt{xs})\end{array}$$

Since this structure takes only one argument, it does not need to be further uncurried. However, the flattening transformations can be performed to expose further structure in this. Since all application nodes are of the form $@^i\ (@^{i+1}\ldots)\ \ldots$, and the recursive calls occur at the same level, no structure

reordering is necessary. However, a cons-lifting, and systematic uncurrying exposes function compositions, as we show below:

$$\begin{aligned}
\mathsf{D}_{\mathtt{fft}}\uparrow_{\mathtt{fft}}\sim \;\; +_2 \;\; & (\langle \mathtt{Cons}\rangle^1 \;\mathsf{I}\; \langle \mathtt{Nil}\rangle^1) \\
& (\;\; (\;\; (\triangle_3 \;\; (\mathtt{genWs} \circ \pi_1) \\
& \qquad\qquad (\pi_1 \circ \pi_2) \\
& \qquad\qquad (\pi_2 \circ \pi_2)) \\
& \qquad \circ (\triangle_2 \;\; \pi_2 \; (\mathtt{halves} \circ \pi_1))) \\
& \quad (\mathsf{I} \times (\mathtt{length} \circ \pi_2))) \\
\circ \mathtt{out}([\mathtt{x}], \mathtt{xs})
\end{aligned}$$

Finally, the combine part is derived by the $\downarrow_{\mathtt{fft}}$ rules. The following structure is generated:

$$\mathsf{D}_{\mathtt{fft}} \;\downarrow_{\mathtt{fft}} \;\; \nabla_2 \; \mathsf{I} \; (\langle \mathtt{comb}\rangle^1 \; \pi_1 \; \pi_2 \; \pi_3)$$

Note that the types of these functions, named $\mathtt{fft}_{\mathtt{div}}$ and $\mathtt{fft}_{\mathtt{comb}}$, are as follows:

```
fft_div : List Complex
      → F (List Complex) (List Complex) (List Complex)
fft_comb : F (List Complex) (List Complex) (List Complex)
      → List Complex
```

However, since the function `genWs` is applied to a tuple component that is generated in the divide function, it can also be applied in the combine part. In that case, the structures would be as follows:

$$\begin{aligned}
\mathsf{D}_{\mathtt{fft}}\uparrow_{\mathtt{fft}}\sim \;\; +_2 \;\; & (\langle \mathtt{Cons}\rangle^1 \;\mathsf{I}\; \langle \mathtt{Nil}\rangle^1) \\
& (\;\; (\;\; (\triangle_3 \;\; (\pi_1) \\
& \qquad\qquad (\pi_1 \circ \pi_2) \\
& \qquad\qquad (\pi_2 \circ \pi_2)) \\
& \qquad \circ (\triangle_2 \;\; \pi_2 \; (\mathtt{halves} \circ \pi_1))) \\
& \quad (\mathsf{I} \times (\mathtt{length} \circ \pi_2))) \\
\circ \mathtt{out}([\mathtt{x}], \mathtt{xs})
\end{aligned}$$

$$\mathsf{D}_{\mathtt{fft}} \;\downarrow_{\mathtt{fft}} \;\; \nabla_2 \; \mathsf{I} \; (\langle \mathtt{comb}\rangle^1 \; (\mathtt{genWs} \circ \pi_1) \; \pi_2 \; \pi_3)$$

Note that these alternative structures can be obtained by applying equational reasoning, since `genWs` occurs in a function composition that is applied to a tuple component that corresponds to a non-recursive argument. In these cases, the types would change as follows:

```
fft_div : List Complex
      → F Int (List Complex) (List Complex)
fft_comb : F Int (List Complex) (List Complex)
      → List Complex
```

This, however, does not have a great impact in the parallelisation of this function. Most of the work is done in the combine part, and this means that it can be parallelised using a reduce structure. The derived structure,

$$\text{HYLO}_{F\,(ListComplex)\,(ListComplex)}\ \texttt{fft}_{\texttt{comb}}\ ((F\ \texttt{genWs}\ \texttt{id})\ \circ\ \texttt{fft}_{\texttt{div}})$$

unifies with $\text{REDUCE}_k\ \_ \circ \_$, by instantiating the first hole to:

$$\texttt{fft}_{\texttt{comb}},$$

and the second hole to an anamorphism

$$\text{ANA}_{F\,(ListComplex)\,(ListComplex)}\ ((F\ \texttt{genWs}\ \texttt{id})\ \circ\ \texttt{fft}_{\texttt{div}}).$$

Image Merge

The image merge example composes two functions, `mark` ad `merge`. The function `mark` computes a threshold in the original pair of images. This threshold is used by the `merge` function to perform the actual merging of the pair of images.

```
imgMerge : List (Img × Img) → List Img
         ∼ PAR_L (_ ‖ FARM n _)
imgMerge = map (λ x, let m = mark x in merge m  x)
```

The structure annotation of `imgMerge` specifies that this function is to be parallelised with a combination of farms and pipelines. The functions **`map`**, `mark` and `merge` must be in the global environment Δ when typechecking this definition. The associated structure of this functions is shown below:

$$\texttt{D}_{\texttt{imgMerge}}\ \Gamma\ (\langle\texttt{map}\rangle^0\ (@^2\ (\texttt{merge}\ |^1_0\ |^0_1)\ (\langle\texttt{mark}\rangle^1\ |^0_0)))$$

Since `imgMerge` does not occur in the body of the applicative structure, the only hylomorphism is the `map` function used in this definition. Since `map` is in the global environment, it can be inlined:

$$\text{MAP}_{L\,(\texttt{Img}\times\texttt{Img})}\ (@^2\ (\texttt{merge}\ |^1_0\ |^0_1)\ (\langle\texttt{mark}\rangle^1\ |^0_0))$$

Finally, an uncurrying transformation exposes the function compositions:

$$\text{MAP}_{L\,(\texttt{Img}\times\texttt{Img})}\ ((\langle\texttt{merge}\rangle^1\ \pi_2\ \pi_1)\circ(\texttt{I}\times\texttt{mark})))$$

Note that this structure is equivalent to the one shown in Section 3.5 on page 96. This structure unifies with the desired parallel structure, with a number of alternative structures that can be disambiguated, given suitable cost models.

$$\begin{aligned}
\delta_1 = \{\ & m_1 \sim \text{FARM}\ n_1\ (\text{FUN}\ (\langle\texttt{merge}\rangle^1\ \pi_2\ \pi_1)),\\
& m_2 \sim \text{FARM}\ n_2\ (\text{FUN}\ (\texttt{I}\times\texttt{mark}))\}\\
\delta_2 = \{\ & m_1 \sim \text{FUN}\ (\langle\texttt{merge}\rangle^1\ \pi_2\ \pi_1),\\
& m_2 \sim \text{FARM}\ n_2\ (\text{FUN}\ (\texttt{I}\times\texttt{mark}))\}\\
\delta_3 = \{\ & m_1 \sim \text{FARM}\ n_1\ (\text{FUN}\ (\langle\texttt{merge}\rangle^1\ \pi_2\ \pi_1)),\\
& m_2 \sim \text{FUN}\ (\texttt{I}\times\texttt{mark})\}\\
\delta_4 = \{\ & m_1 \sim \text{FUN}\ (\langle\texttt{merge}\rangle^1\ \pi_2\ \pi_1),\\
& m_2 \sim \text{FUN}\ (\texttt{I}\times\texttt{mark})\}
\end{aligned}$$

Quicksort

The quicksort example in Section 3.5 on page 96 assumed a fixed type A, together with functions for filtering the elements of the list as required. The quicksort example that we show here takes as an argument the `Ord` instance dictionary for type a.

```
qsort : Ord a → [a] → [a]
qsort = λ d xs.
          case xs of {
            Nil         -> Nil;
            (Cons x xs) -> app
                             (qsort d (le d x xs))
                             (Cons x (qsort d (gt d x xs))) }
```

The associated structure of quicksort is:

$$
\begin{aligned}
&\mathtt{D_{qsort}}\ \Gamma\ (\langle\nabla_2\rangle^2\ \ \langle\mathtt{Nil}\rangle^3 \\
&\qquad\qquad (\langle\mathtt{app}\rangle^3 \\
&\qquad\qquad\quad (\langle\mathtt{qsort}\rangle^3\ |_2^0\ (\langle\mathtt{le}\rangle^3\ |_2^0\ (\langle\pi_1\rangle^3\ |_0^2)(\langle\pi_2\rangle^3\ |_0^2))) \\
&\qquad\qquad\quad (\langle\mathtt{Cons}\rangle^3\ (\langle\pi_1\rangle^3\ |_0^2) \\
&\qquad\qquad\qquad (\langle\mathtt{qsort}\rangle^3\ |_2^0\ (\langle\mathtt{gt}\rangle^3\ |_2^0\ (\langle\pi_1\rangle^3\ |_0^2)(\langle\pi_2\rangle^3\ |_0^2)))))) \\
&\qquad\qquad (\langle\mathtt{out_L}\rangle^2\ |_0^1)
\end{aligned}
$$

The hylomorphism derivation, plus the systematic uncurrying of the resulting structures to expose function compositions produces the following hylomorphism.

$$T\ B\ A = 1 + A \times (B \times A)$$

$$
\begin{aligned}
&\textsc{hylo}_{T\,A}\ \ (\nabla\ \langle Nil\rangle^1\ (\langle\mathtt{app}\rangle^1\ \pi_1\ (\langle\mathtt{Cons}\rangle^1\ \pi_2\ \pi_3))) \\
&\qquad\qquad (+_2\ \ \triangle_0\ \ (\triangle_2\ \ \pi_1\ (\langle\mathtt{le}\rangle^1\ \pi_1\ (\pi_1 \circ \pi_3)(\pi_1 \circ \pi_3))) \\
&\qquad\qquad\qquad\qquad (\triangle_2\ \ (\pi_1 \circ \pi_3) \\
&\qquad\qquad\qquad\qquad\qquad (\triangle_2\ \ \pi_1\ (\langle\mathtt{gt}\rangle^1\ \pi_1\ (\pi_1 \circ \pi_3)(\pi_2 \circ \pi_3))))) \\
&\qquad\qquad\qquad \circ(\mathtt{distr}_2^2\ \circ\ (\pi_1 \times \pi_2 \times \langle\mathtt{out_L}\rangle^1\ \pi_2)))
\end{aligned}
$$

The alternative representation of quicksort as a hylomorphism makes it possible to parallelise it using the rules of the StA framework. For example, in order to parallelise it using a divide and conquer skeleton, the following rule can be applied.

$$\textsc{map}_L(\textsc{hylo}_F\ \mathrm{A}\ \mathrm{A}) \cong \textsc{par}_L\ (\textsc{dc}_{n,F}\ \mathrm{A}\ \mathrm{A})$$

It suffices to annotate the type with the corresponding structure as we show below:

$$
\begin{aligned}
&\mathtt{map\ qsort\ :\ Ord}\ a \rightarrow \mathtt{List\ (List}\ a\mathtt{)} \rightarrow \mathtt{List\ (List}\ a\mathtt{)} \\
&\qquad\qquad \sim\ \textsc{par}_L\ (\textsc{dc}_{n,F}\ \mathtt{A\ A})
\end{aligned}
$$

Barnes-Hut N-Body Simulation

N-Body simulations are widely used in astrophysics. They comprise a simulation of a dynamic system of particles, usually under the influence of physical forces. The Barnes-Hut simulation recursively divides the n bodies

storing them in an `Octree`, where each node in the tree represents a region of the space. The topmost node represents the whole space and the eight children, the octants. The leaves of the tree contain the bodies. The algorithm continues by calculating the cumulative mass and centre of mass of each region of the space. Finally, the algorithm calculates the net force on each particular body by traversing the tree, and updates its velocity and position. This process is repeated for a number of iterations. The high-level implementation details of Barnes-Hut in the language HH is shown below:

```
buildTree : Area → List Particle → Tree

calcForces : Tree → List Particle → List Force
calcForces t ps = map (calc t) ps

move : List Particle → List Force → List Particle
move ps fs =
  case (ps, fs) of {
     (Nil, fs) → Nil
     (ps, Nil) → Nil
     (Cons x xs, Cons f fs) → Cons (moveP x f) (move xs fs)
  }

step  : List Particle → List Particle
step ps =    let tree = buildTree initArea ps
          in let fs = calcForces tree ps
          in move ps fs

nbody : Int → List Particle → List Particle
nbody n ps = case n <= 0 of {
                True  → ps ;
                False → nbody (n-1) (step ps)
             }
```

We focus on the structure of `step` and `move`, since they illustrate how finding hylomorphisms can be used to parallelise this code. The associated

structure of `move` is show below:

$$\begin{array}{lll}\mathtt{D}_{\mathtt{move}}\ \Gamma\ ((\langle\nabla_3\rangle^1 & \langle\mathtt{Nil}\rangle^2 & \\ & \langle\mathtt{Nil}\rangle^2 & \\ & (\langle\mathtt{Cons}\rangle^2 & (\langle\mathtt{moveP}\rangle^1\ (\langle\pi_1\circ\pi_1\rangle^2\ |_0^1)(\langle\pi_1\circ\pi_2\rangle^2\ |_0^1)) \\ & & (\langle\mathtt{move}\rangle^2\ (\langle\pi_2\circ\pi_1\rangle^2\ |_0^1)(\langle\pi_2\circ\pi_2\rangle^2\ |_0^1)))) \\ & \mathtt{mout})\end{array}$$

This structure is flattened after doing a const-lifting. Note that all the substructures occur under $\langle\rangle^1$. This means that the argument is discarded by all substructures, except for `mout`.

$$\begin{array}{lll}\mathtt{D}_{\mathtt{move}}\ \Gamma\ ((\nabla_3 & \langle\mathtt{Nil}\rangle^1 & \\ & \langle\mathtt{Nil}\rangle^1 & \\ & (\langle\mathtt{Cons}\rangle^1 & (\langle\mathtt{moveP}\rangle^1\ (\pi_1\circ\pi_1)(\pi_1\circ\pi_2)) \\ & & (\langle\mathtt{move}\rangle^1\ (\pi_2\circ\pi_1)(\pi_2\circ\pi_2)))) \\ & \circ\ \mathtt{mout})\end{array}$$

The divide and combine functions that are extracted from this structure are shown below:

$$\begin{array}{lll}\mathtt{move}_{\mathtt{div}}\ ((+_3 & \triangle_0 & \\ & \triangle_0 & \\ & (\triangle_2 & (\triangle_2\ (\pi_1\circ\pi_1)(\pi_1\circ\pi_2)) \\ & & (\triangle_2\ (\pi_2\circ\pi_1)(\pi_2\circ\pi_2)))) \\ & \circ\ \mathtt{mout})\end{array}$$

$$\begin{array}{lll}\mathtt{move}_{\mathtt{comb}}\ (\nabla_3 & \langle\mathtt{Nil}\rangle^1 & \\ & \langle\mathtt{Nil}\rangle^1 & \\ & (\langle\mathtt{Cons}\rangle^1 & (\langle\mathtt{moveP}\rangle^1\ (\pi_1\circ\pi_1)(\pi_2\circ\pi_1)) \\ & & \pi_2))\end{array}$$

The structure of the `move` hylomorphism corresponds almost exactly to the list base functor,

$$\Lambda B\ A,\ 1+1+B\times A,$$

With minimal rewriting for merging branches that result in the same value, the `move` function can be turned into an anamorphism that is semantically

equivalent to a `zip`, followed by a `map` that applies a function equivalent to the structure that is equivalent to uncurrying the `moveP` function:

$$\langle \texttt{moveP} \rangle^1 \ \pi_1 \ \pi_2$$

Note that the function `move` is applied in function `step` to the result of `calcForces`, which is another map. This means that the StA framework can derive a suitable parallelisation for the `step` function, in terms of farms and pipelines:

```
step  : List Particle → List Particle ∼ _ ‖ _
```

Iterative Convolution

Finally, we briefly discuss the parallelisation of an algorithm for iterative convolution. This algorithm applies a convolution algorithm to a list of images. Each convolution, implemented by function `iterconv`, consists of applying a kernel to an image in a divide and conquer way. This is done until a dynamic condition is met, which is tested by function `finished`.

```
kern : Kern → Img → Img
kern k i = case split i of {
                Left x                  → apply k x ;
                Right (x1, x2, x3, x4) → combine (kern k x1)
                                                 (kern k x2)
                                                 (kern k x3)
                                                 (kern k x4)
           }

iterconv : Kern → Img → Img
iterconv k x = case (finished x) of {
                      True  → x ;
                      False → conv k (kern k x)
                }


conv : Kern → List Img → List Img
conv k = map (iterconv k)
```

The function `kern` can be parallelised using any parallel structure that is equivalent to hylomorphism with a quad-tree structure:

$\texttt{kern} : \texttt{Kern} \to \texttt{Img} \to \texttt{Img} \sim \textsc{hylo}\ \_\ \_$

The function `conv` is a map, which means that it can be parallelised using a task farm. However, if we want to expose more parallelism, we need to split the function `iterconv` into smaller components. The structure that is derived for `iterconv` is the one that is shown below:

$$\mathsf{D}_{\texttt{iterconv}}\ \Gamma\ (\langle\nabla_2\rangle^2\ |_1^1\ (\langle\texttt{conv}\rangle^3\ |_2^0\ (\langle\texttt{kern}\rangle^3\ |_2^0\ |_1^1))\ (\langle\texttt{out}_{\texttt{Bool}}\rangle^2\ (\langle\texttt{finished}\rangle^2\ |_1^1)))$$

The divide structure that is extracted from this is the following:

$$(+_2\ \pi_2\ (\triangle_2\ \ \pi_1\ (\triangle_2\ \ \pi_1\ \pi_2)))\ \circ\ \texttt{distr}_2^2\ (\triangle_3\ \ \pi_1\ \pi_2\ (\texttt{out}_{\texttt{Bool}} \circ \texttt{finished} \circ \pi_2))$$

This structure has type

$$\texttt{Kern} \times \texttt{Img} \to \texttt{Img} + \texttt{Kern} \times \texttt{Img},$$

and corresponds to the anamorphism part of an ITER structure, provided that the branches are reversed both in the out function and case statement. After rewriting the function as required, the function `conv` can be parallelised using a task farm and feedback loops, as was shown in Section 3.5 on page 96:

$\texttt{conv} : \texttt{Kern} \to \texttt{List Img} \to \texttt{List Img} \sim \textsc{farm}\ (\textsc{fb}\ \_)$

5.7 Discussion

This chapter has presented a techique for annotating types of a functional language, HH, with structures that can be used for reasoning about the introduction of parallelism. This language is a subset of Haskell 98, and illustrates how this technique can be applied to a real functional language. This technique relies on combinatory logic, as a *bridge* between a pointed functional language and algorithmic skeletons. This mechanism allows us to rewrite programs in HH, according to rules defined at the Hylo and applicative levels. Moreover, the connection between algorithmic skeletons and

combinatory logic is a novel feature of the technique that was described in this chapter, and it is worth exploring further. For example, a representation of algorithmic skeletons directly as applicative expressions can be used to find similar structures in programs. A technique that is worth exploring for this is described is *antiunification*, a technique explored in Barwell's forthcoming thesis [Bar17].

Chapter 6

Conclusions and Future Work

The main goal of this thesis was to develop novel state-of-the-art mechanisms for reasoning simultaneously about the run-time performance of, and the functional equivalences between parallel programs. These mechanisms are aimed at systematically exploring the space of functionally equivalent parallel implementations. They provide a general framework for parallel programming in which a programmer can write a program once, and then parallelise it by providing type annotations that are parameterised by cost information. To achieve this goal, three main techniques were used:

Functional programming. The absence of side effects has allowed Structured Arrows to exploit equational reasoning in a way that would be much more restricted in the presence of unrestricted side effects.

Hylomorphisms. Hylomorphisms have been used to capture the semantics of parallel programs using a single unifying construct. A canonical representation of programs as a composition of hylomorphisms has been used to systematically explore the space of equivalent parallel implementations.

Type Systems. Types provided a suitable framework to encode the static analyses that has been developed for this thesis. The compositional approach that was described throughout this thesis has allowed the Structured Arrows framework to be realised as an extension of standard type checking, inference and unification algorithms.

This thesis has focused on answering three main questions:

1) *How can a program structure be extracted?* Extracting a program structure means identifying the different components of a program that can be parallelised.

2) *What are all the different ways in which a program structure can be rewritten into functionally equivalent forms?* Systematically exploring how a program structure can be rewritten provides a mechanism to explore the space of all possible parallel implementations of a program.

3) *How can a program's run-time behaviour be statically predicted, based on its structure?* Statically predicting the run-time behaviour of alternative implementations provides a mechanism for selecting a suitable parallel implementation for a program.

Collectively answering these three questions has brought several contributions in the field of structured parallelism, by opening the way to a new automatic parallelisation process, guided by type annotations that are used to reason about structured parallel processes. The answer to these questions provided a novel type-based framework, *Structured Arrows*, that serves the purpose of statically reasoning about *how* to parallelise a program by: (a) reasoning about *how* a program can be parallelised; and (b) statically predicting *what* parallel implementation will achieve the best possible speedups. This framework provides strong static guarantees that a parallel structure introduced to a program does not change its functional behaviour. Moreover, as part of the *Structured Arrows* framework, a procedure for deciding semantic equivalences of parallel programs has been developed, as well as a systematic process to explore all possible alternative parallel implementations of a program. Program structures have been coupled with cost models that are formally derived from their operational semantics. By combining the rewritings for program structures with these cost models, the StA framework can be used to derive *provably optimal* parallel programs, within the model of the operational semantics of the queue-based language in which algorithmic skeletons are defined. This framework, therefore, satisfies the goal of finding a common, general framework for reasoning simultaneously about program rewritings and performance.

How can a program structure be extracted?

Chapter 3 presents a new type-based framework, *Structured Arrows*, that consists of a type-and-effect system for a point-free functional programming language with hylomorphisms, Hylo. The program structure is extracted as an abstraction of the program's AST, and consists of a composition of functions, where recursion is represented by the type of the underlying hylomorphism. The type-annotations are extended with common algorithmic skeletons, which can then be used to specify in the function types the intended parallel strategy for the respective functions. These type-annotations correctly split the different components of the underlying function that can be parallelised.

Chapter 5 described an extension of the *Structured Arrows* framework that extends the expression language to the purely functional language HH. The language HH is a subset of the Haskell programming language. The extension to the type-system of HH extracts the underlying program structure as a hylomorphism. To achieve this, it uses the connection between combinatory logic and λ-calculus. The extended StA framework uses a mechanism for coverting between point-free programs in applicative form and Hylo.

The use of hylomorphisms in both Chapters 3 and 5 provides, therefore, a mechanism for extracting a program structure, i.e. a decomposition of the input program into the different components that can be parallelised.

What are all the different ways in which a program structure can be rewritten into functionally equivalent forms?

Chapter 3 formally defined a set of equivalences between programs in Hylo, and a decision procedure based on this set of equivalences. The different program structures in Hylo can be compared by rewriting them into a *canonical representation*. The same process can be followed in reverse order, and therefore used to explore all possible ways in which a function can be rewritten, up to the set of equivalences considered. This is explored by the unification algorithm in Chapter 3, which considers program structures with holes that need to be inferred, i.e. parts of the structure that are not specified. The unification algorithm takes a program structure, and a target structure with holes, and returns the set of all possible ways of instan-

tiating the holes of the target structure, so that the resulting structure is equivalent to the program structure. By extending the number of program equivalences, a larger set of alternative functionally equivalent parallel programs can be compared. The extension of the *Structured Arrow* framework in Chapter 5 extends the equivalences considered to pointed programs in the programming language HH.

Essentially, the canonical representation of Hylo provides both a way to compare the functionality of different parallel programs, but also to systematically explore all possible alternative parallelisations of a program, up to the set of equivalences considered.

How can a program's run-time behaviour be statically predicted?

Chapter 4 presented an operational semantics for a set of algorithmic skeletons. This operational semantics is defined in terms of a queue-based language that consists of three primitives: `enqueue`, `dequeue` and `eval`. The parallel composition of workers defined in terms of these three primitives can be used to define a number of common algorithmic skeletons. The main novelty of this operational semantics is that it provides at the same time both a translation scheme to generate low-level skeletal code, and a mechanism for systematically deriving cost models from the operational semantics. These cost models can therefore be used to generate provably optimal parallel programs. This approach is extensible: new parallel structures can be defined, and considering more complex cost models can be done by underlying queue-based language.

6.1 Contributions of this Thesis

As we mentioned in the Chapter 1, this thesis has made the following main contributions.

1. Structure-Annotated Arrows (StA).

The novel Structured Arrows type-based framework has been developed for this thesis (Chapter 3). This framework is a type-and-effect system that annotates function types of a point-free programming language with

hylomorphisms, Hylo, with the underlying *program structure*. The program structure was defined as the combination of hylomorphisms and algorithmic skeletons that are used in the implementation of a function. This abstraction of the program structure can be used to reason about possible ways of rewriting between functionally equivalent forms. The soundness of the approach has been proved, and its usage illustrated with a number of examples. Throughout the rest of the thesis, in Chapter 4 and Chapter 5, the framework of Structured Arrows was extended with: (a) an operational semantics of common representative algorithmic skeletons; and (b) a broader, more expressive expression language. These extensions are discussed later in this section, since they are contributions on their own. However, the extensibility of the Structured Arrows framework is illustrated thanks to those extensions.

As discussed at the end of Chapter 3, this type-and-effect system can be considered a form of *behavioural types*, and its usage is not restricted to algorithmic skeletons. StA can potentially be used for more general program optimisations, such as substituting particular instances of hylomorphisms by optimised low-level implementations, or synthesizing hardware.

The StA framework uses the first known representation of the parallel structure of a program as a type. Thanks to this type-based approach, most of the techniques described in this thesis were implemented as extensions of standard type checking, inference and unification techniques.

The main novelty of StA is the full separation between the notions of *what* is a program computing, and *how* is a program performing the computation. There is a major advantage in doing so, since programmers can then focus on each of these concerns separately. Moreover, the mechanisms for reasoning about how to parallelise a program consider, simultaneously, cost and correctness. This implies that the Structured Arrows framework can potentially lead to the automatic provably optimal parallelisation of functions.

2. A denotational semantics for common algorithmic skeletons in terms of hylomorphisms.

Chapter 3 presented a denotational semantics of common algorithmic skeletons in terms of a single unifying construct: *hylomorphisms*. Although hylomorphisms have been previously used to explain the semantics of certain algorithmic skeletons, as has been discussed in Chapter 2, this is the first attempt at using them as a single, unifying construct. Using hylomorphisms as a single, unifying construct reduces the problem of rewriting a parallel program into a different parallel form, to applying well-known and well-understood hylomorphism laws. Common approaches in rewriting algorithmic skeletons generally use ad-hoc rewriting rules that are derived implicitly from the semantics of the algorithmic skeletons. A common example is considering explicitly the pipeline associativity as a rewriting rule. In the Structured Arrows framework, thanks to the definition of pipelines as the composition of two hylomorphisms, pipeline associativity is no longer required: this property is derived from the denotational semantics of pipelines *for free*.

The novelty of considering hylomorphisms as a single unifying construct opens many possibilities for reasoning about functional equivalences of parallel programs. The definition of *a decision procedure for the functional equivalence of alternative parallel implementation*, discussed later in this section, was possible thanks to defining a canonical representation of programs in terms of hylomorphisms. Essentially, when reasoning about semantic equivalences of alternative parallelisations, one only needs to answer the question: *what is the (composition of) hylomorphisms that represents this program?*

3. An operational semantics of common algorithmic skeletons, in terms of a small and predictable queue-based language, with a precise and well-known operational semantics.

In Chapter 4, a novel operational semantics of common, representative algorithmic skeletons is defined, in terms of a queue-based language. This queue-based language contains three primitives: `enqueue`, `dequeue`, and `eval`. These primitives are used to describe *workers* of a parallel process,

which run in a loop a sequence of dequeue operations, followed by an `eval` operation, and finally a sequence of `enqueue` operations. These workers are composed in parallel, and by carefully connecting their shared queues, the semantics of different algorithmic skeletons can be defined.

This operational semantics provides a mechanism to generate *predictable* machine code from a high-level implementation, provided that the `enqueue`, `dequeue` and `eval` operations can be implemented in the target architecture, satisfying the necessary assumptions. This is illustrated in Chapter 4 using a number of examples.

Finally, the main novelty of the operational semantics is that it ties program structures with cost models. The operational semantics in terms of the queue-based language provides a way to reason about soundness and performance of algorithmic skeletons. Cost equations can be derived systematically from their operational behaviour. This means that the operational semantics can, *for free*, generate cost models that will then be used by the StA framework.

4. A systematic mechanism for deriving cost models from the operational semantics of algorithmic skeletons.

As was previously discussed, that the operational semantics of algorithmic skeletons was defined so that it has a predictable run-time. See Contribution 3 for more details. This predictability is what makes it possible to derive cost equations directly from the operational semantics. This is illustrated in Chapter 4 with a number of common, representative algorithmic skeletons.

Basically, from a specification of the operational semantics of an algorithmic skeleton, we can derive (almost) *for free*: (a) a translation scheme that can be used to compile a structured parallel program to low-level predictable code, and (b) cost models that can be used to statically predict the run-time behaviour of a program. This is very powerful, since it implies that whenever a parallel structure is defined in this language, a cost equation will be derived that is tied to its operational semantics.

Finally, the use of these cost equations within the Structured Arrows framework was discussed. The type-based approach made it possible to es-

timate the input and output size tasks by implementing a variant of *sized types*. This was applied to a number of examples, to compare real vs predicted speedups of parallel programs within this framework, providing evidence that the resulting parallel implementations was predictable.

In summary, the main novelty of this systematic process for deriving cost models from an operational semantics is that it provides a way to derive *provably optimal* parallel implementations, with respect to the operational semantics. This is ensured thanks to the fact that the cost models are abstractions of the operational semantics. Since the operational semantics is also used to generate low-level parallel code, we can ensure that the cost models accurately capture the cost of parallel programs within this model.

5. An extension of the Structured Arrows framework to deal not just with point-free programs, but also with pointed programs that are written using explicit recursion.

Chapter 5 presented a relation between terms in a functional language, the subset of Haskell here called HH, and their possible parallelisations using algorithmic skeletons. This relation exploits and extends the Structured Arrows framework. In order to relate arbitrary terms in HH with its parallelisations, first a hylomorphism structure is extracted from programs. This is done by exploiting the well-known correspondence between λ-calculus and combinatory logic. A λ-term can be converted to combinatory logic using a compositional approach. A different notion of applicative terms is defined in Chapter 5, that correspond to a point-free representation of terms in HH derived from combinatory logic. A series of systematic rewritings is applied to applicative terms in order to derive the hylomorphism structure. This contribution illustrates how the Structured Arrows framework can be applied to a real functional language such as Haskell.

The main novelty of this extension is, therefore, the usage of *combinatory logic* to find program structures. This connection between combinatory logic and hylomorphisms represents a technique for finding parallel structures in arbitrary HH terms. Moreover, the usage of combinatory logic opens the possibility of using well-understood techniques in the field of applicative expressions to do program rewritings. As a final remark, the translation

between HH terms and Hylo was done in a very easy way, as an extension of a standard type-checking algorithm, thanks to the use of combinatory logic as an intermediate step.

6. A novel decision procedure for the functional equivalence of alternative parallel implementation.

The main novelty of this decision procedure is the usage of hylomorphisms to find a canonical representation of programs. This canonical representation consists on a "reforestation" process that splits a program into the smallest possible components, which can later be reassembled into different parallel structures. This decision procedure was presented in Section 3.4 on page 86, and it provides a mechanism for deciding whether to different parallel programs are functionally equivalent. Extending the semantic equivalences that we considered broadens the amount of parallel programs that can be compared. This decision procedure also opens up the possibility of verifying parallel programs, by adapting it to different parallel programming frameworks. The verification of parallel programs would then proceed by first rewriting a parallel program into a canonical representation, and then comparing the canonical representation with a reference sequential implementation.

7. A prototype implementation of the Structured Arrows framework.

We provide a prototype implementation of the type-system described in Chapter 3 and Chapter 4, which can be found at `https://bitbucket.org/david_castro/skel`. These prototypes shows that the approach that has defined in this thesis can be easily implemented as a functional language, and opens many possibilities, such as: (a) building a full compiler for structured parallel programs; (b) using the Structured Arrows framework as a rewriting engine for parallelism; and (c) studying more general optimisations and program rewritings within the Structured Arrows framework. The novelty of this prototype implementation is, therefore, that it illustrates how the Structured Arrows framework can be implemented as

part of a real functional language, by a set of modifications to the type system.

6.2 Limitations

Most of the limitations of this work are imposed by the theoretical frameworks that have been used for structured parallelism, for the operational semantics of parallel skeletons, and for structured patterns of recursion. Many of these limitations can be tackled by extending the approach with the results of more general theoretical frameworks. The following limitations are worth mentioning:

• *No side effects.* The assumption of a pure functional setting means that functions are not allowed side effects. This limitation, however, is beneficial since it is what allows the Structured Arrows framework to use extensively equational reasoning to rewrite a program into an equivalent parallel structure. The usage of equational reasoning would be greatly limited by allowing the usage of impure functions.

• *We did not consider more complex algorithmic skeletons.* Complex algorithmic skeletons, such as algorithmic skeletons that capture the *Bulk Synchronous Parallel* model, have not yet been fully considered. Section 4.5 on page 138 sketches how a Bulk Synchronous Parallel skeleton could be implemented in a queue-based model. However, there are other parallel patterns that are yet to be considered, such as stencil computations, gather-scatter, etc. Since the Structured Arrows framework was designed to be easily extensible, this is not a major limitation. We conjecture that such skeletons will easily fit within our general scheme. The algorithmic skeletons that were chosen for this thesis were selected from a list of common representative skeletons [DT13] that capture a wide variety of parallel algorithms.

• *We did not consider more complex patterns of recursion.* Hylomorphisms are Turing universal, as discussed in Chapter 2. However, they do not capture patterns of recursion such as mutually recursive functions or nested recursion. This means that the recursive functions that were captured by

the languages in Chapter 5 and Chapter 3 could not be defined using mutual or nested recursion, and these functions need to be rewritten so that they could be captured by a hylomorphism.

• *We did not consider more complex back-ends.* The operational semantics in Chapter 4 is deliberately simple enough to make it predictable. This implies that many issues, such as *bounded queues* were not considered in this thesis, since they would complicate predicting the run-time behaviour of programs.

• *The cost models rely on many assumptions.* Low-level details such as memory accesses were not taken into account for generating the cost equations. These assumptions were crucial for making the computation of the static run-time estimations feasible, although they add potential sources of imprecision.

6.3 Further Work

Based on the limitations, and the current trends in parallel programming languages, frameworks and theories, there are several ways in which this thesis can be improved.

Combine the static approach with dynamic approaches.

The Structured Arrows framework relies completely on a static approach. This, however, requires a full knowledge of the sizes of the inputs for a program, and this approach would not work whenever a high variability in the inputs is expected. Several dynamic approaches can be applied to solve the problem of parallelising irregular applications. Examples of this include the use of *adaptive skeletons* [MMT16, BK95], or the use of advanced scheduling algorithms [JH10].

The static approach can determine when a dynamic approach is preferred to a rigid, fixed parallel structure. This would require extending the sized types approach used to estimate task sizes to incorporate information about the *range of expected sizes*, or a *distribution* of expected task sizes. The cost

equations derived from the operational semantics would need then to take into account this variability in a systematic way.

If those problems were solved, then a whole new range of dynamic approaches would become available to the Structured Arrows framework.

Allow certain kinds of side effects for writing parallel programs.

The idea of allowing uncontrolled side effects when writing a parallel application would remove the possibility of using equational reasoning for code rewriting. However, there are certain approaches, such as *deterministic parallelism* [MNPJ11, KRBJ12] that allow this limitation to be relaxed slightly, provided that a parallel program can be proven to be *deterministic*, i.e. that it returns the same result regardless of the execution order of the parallel components. Some side effects could be potentially allowed, if they are shown not to affect the overall result of the program.

This extension would require studying the interactions between side effects and hylomorphisms, and might limit slightly the kind of equational reasoning that can be performed to parallelise programs. However, this would open the possibility to extend the StA framework to (certain kinds of) imperative programs.

Use more complex, more general recursion patterns.

Hylomorphisms were used in this thesis as a general unifying construct to compare alternative parallelisations of functions, and to explore how to automatically parallelise sequential functions that can be represented as a composition of hylomorphisms.

There are more general recursion patterns that can be used for the same purpose, which would open the possibility of exploring e.g. mutually recursive definitions or nested recursion. These recursion patterns include *paramorphisms*, *mutumorphisms*, or even adjoint folds [Hin10] and conjugate hylomorphisms [HWG15].

Introduce more complex skeletons in the *Structured Arrows* framework.

Although the set of algorithmic skeletons that were considered in this thesis were a representative set, that is suitable for parallelising a wide range of applications [DT13], it would be interesting to consider more complex patterns of recursion, such as a Bulk Synchronous Parallel skeleton [Val90]. These algorithmic skeletons could help speed up further some parallel applications developed in the Structured Arrows framework.

Study the relation between *Structured Arrows* and other, more common forms of behavioural types, such as session types.

The work described in this thesis was inspired by the field of *behavioural types* [ABB+16]. Studying the interactions between the StA framework and other approaches in behavioural types could bring the benefits of applying the results developed in the context of behavioural types to the Structured Arrows framework. For example, the representation of algorithmic skeletons in terms of *session types* [HVK98] could be helpful to provide a mechanism for defining further algorithmic skeletons in the Structured Arrows framework. Moreover, we could add much more reasoning power to StA if we could annotate sequential implementations with *session types*. The main challenge would be to determine the possible ways in which a sequential implemntation could be realised as a parallel program that follows the given protocol description.

Extend the queue-based language with more complex, low-level information.

The queue-based language of Chapter 4 can be extended with further low-level information that can be used to better predict the run-time performance of parallel programs. For example, implementating algorithmic skeletons does not necessarily require the use of unbounded queues for communication. Actually, it is probably worth to use bounded queues, and temporarily halting the execution of workers writing to full queues, until the consumers of this queue have removed some of the tasks that are contained

in it. These features, however, would complicate the cost models, and issues such as a *back-pressure* algorithm [NU09] would need to be considered carefully.

An additional feature that may be worth extending is memory access. Although workers compute pure functions, there is a "hidden contention" between the different workers of a parallel program: memory accesses. In memory-intensive computations, this might have an important effect on the total run-times of parallel applications, and it is therefore a problem worth looking into.

Consider heterogeneous architectures.

In this thesis, we assume a queue-based model. This queue-based model can be realised in multiple architectures, so in that sense the approach of Structured Arrows is architecture-independent. However, this model relies on the assumption that the architecture in which the queue-based model is realised is not heterogeneous. Considering a heterogeneous architecture would require not only modelling the parallel composition of the workers, but also modelling the mapping of workers to the different execution units of the target hardware architecture. It would not, however represent a fundamental change to the techniques that are described in this thesis.

Statically predicting energy consumption

The cost models that are derived in Chapter 4 are concerned only with execution times. However, statically predicting the energy consumption of programs in energy-bound devices is gradually capturing more attention [LGG13, KE15, GGP+15, LGK+15, GKCE17]. Since the StA framework accepts any alternative model for the operational semantics and cost models of the parallel constructs, it would not represent a major change to the StA framework to consider energy consumption as well as time, and it would increase its reasoning power about non-functional properties.

6.4 Summary

To conclude, this thesis has tackled the problem of reasoning simultaneously about: (a) the safe parallelisation of sequential functions; and, (b) the run-time performance of parallel programs. By achieving the goals of this thesis, a new general type-based framework for parallel programming has been developed, *Structured Arrows (StA)*. This framework represents a significant step towards the automatic and provably optimal parallelisation by inserting the appropriate type annotations to programs. Structure and functionality are completely separated in the Structured Arrows framework. The code for a parallel program is written *just once* in the Structured Arrows framework, and cost and type annotations determine how it is parallelised. Parallel structure becomes a first-class construct that can be applied to different functions. This greatly eases the task of developing a parallel program, since it completely separates the major questions of writing parallel software: *what* is the program computing, and *how* is it doing it.

Appendix A

Reforestation Confluence

This appendix shows the core part of the proof that the rewriting systems $\rightsquigarrow_{\mathsf{p}}$ and $\rightsquigarrow_{\mathsf{s}}$ are confluent. We name the rewriting rules using the name of the equivalence rules from which they are derived. For example, FARM-EQUIV is used for the rewriting $\text{FARM}_n\ \sigma \rightsquigarrow \sigma$.

A.0.1 Lemma *Both sides of all critical pairs of $\rightsquigarrow_{\mathsf{p}}$ can reduce to the same form.*

Proof We show for each pair of rules all possible structures that yield a critical pair, and give a justification when there is none.
Case FARM-EQUIV, PIPE-EQUIV. $\sigma = \text{FARM}_n\ (\text{FUN}\ \sigma_1 \parallel \text{FUN}\ \sigma_2)$.

Case 1. By PIPE-EQUIV, $\sigma \rightsquigarrow \text{FARM}_n\ (\text{FUN}\ (\sigma_2 \circ \sigma_1))$. By applying FARM-EQUIV, σ reduces to $\text{FUN}\ (\sigma_2 \circ \sigma_2)$

Case 2. By FARM-EQUIV, $\sigma \rightsquigarrow (\text{FUN}\ \sigma_1 \parallel \text{FUN}\ \sigma_2)$. By PIPE-EQUIV, σ reduces to $\text{FUN}\ (\sigma_2 \circ \sigma_2)$.

Case FARM-EQUIV, DC-EQUIV. $\sigma = \text{FARM}_n\ (\text{DC}_{m,F}\ \sigma_1\ \sigma_2)$.

Case 1. By applying first FARM-EQUIV and then DC-EQUIV,

$$\text{FARM}_n\ (\text{DC}_{m,F}\ \sigma_1\ \sigma_2) \rightsquigarrow \text{DC}_{m,F}\ \sigma_1\ \sigma_2 \rightsquigarrow \text{FUN}\ (\text{HYLO}_F\ \sigma_1\ \sigma_2).$$

*Case 2.*By applying first DC-EQUIV and then FARM-EQUIV,

$$\text{FARM}_n\ (\text{DC}_{m,F}\ \sigma_1\ \sigma_2) \rightsquigarrow \text{FARM}_n\ (\text{FUN}\ (\text{HYLO}_F\ \sigma_1\ \sigma_2)) \rightsquigarrow \text{FUN}\ (\text{HYLO}_F\ \sigma_1\ \sigma_2).$$

Case FARM-EQUIV, FB-EQUIV. $\sigma = \textsc{farm}_n\ (\textsc{fb}\ (\textsc{fun}\ \sigma))$

Case 1. FARM-EQUIV, then FB-EQUIV.

$$\textsc{farm}_n\ (\textsc{fb}\ (\textsc{fun}\ \sigma)) \rightsquigarrow \textsc{fb}\ (\textsc{fun}\ \sigma) \rightsquigarrow \textsc{fun}\ (\textsc{hylo}_{(+B)}\ (\textsc{id}\triangledown\textsc{id})\ \sigma)$$

Case 2. FB-EQUIV, then FARM-EQUIV.

$$\textsc{farm}_n\ (\textsc{fb}\ (\textsc{fun}\ \sigma)) \rightsquigarrow \textsc{farm}_n\ (\textsc{fun}\ (\textsc{hylo}_{(+B)}\ (\textsc{id}\triangledown\textsc{id})\ \sigma)) \rightsquigarrow \textsc{fun}\ (\textsc{hylo}_{(+B)}\ (\textsc{id}\triangledown\textsc{id})\ \sigma)$$

Case FARM-EQUIV, *with a* $\textsc{par}_T$ *structure.* There are no critical pairs, since a $\textsc{farm}_n$ is in Σ_{p}, and $\textsc{par}_T$ in Σ. If $\sigma = \textsc{par}_T\ (\textsc{farm}_n\ (\textsc{fun}\ \sigma))$, then the only possible order is FARM-EQUIV, and then the PAR-MAP rule.

Case PIPE-EQUIV, *and* DC-EQUIV. There are no critical pairs, since PIPE-EQUIV requires both sides to be a FUN, which implies that DC-EQUIV must always be applied first.

Case PIPE-EQUIV, *and* FB-EQUIV. There are no critical pairs, since PIPE-EQUIV requires both sides to be a FUN, which implies that FB-EQUIV must always be applied first.

Case PIPE-EQUIV, *with a* $\textsc{par}_T$ *structure.* The rule for $\textsc{par}\,T$ requires a FUN, so there are no critical pairs.

Case DC-EQUIV *and* FB-EQUIV. There are no critical pairs, since FB-EQUIV requires its structure to be a FUN, which implies that DC-EQUIV must always be applied first.

Case DC-EQUIV *and* $\textsc{par}_T$. There are no critical pairs, since the rule for $\textsc{par}_T$ requires a FUN, so DC-EQUIV must be applied first.
Case FB-EQUIV *and* $\textsc{par}_T$. There are no critical pairs, since the rule for $\textsc{par}_T$ requires a FUN, so FB-EQUIV must be applied first. □

A.0.2 Lemma *All critical pairs that arise from the rules* $\rightsquigarrow_{\mathsf{s}}$ *can reduce to the same form.*

Proof The proof follows the same structure as the one for $\rightsquigarrow_{\mathsf{p}}$.

Case ID-CANCEL-L *and* ID-CANCEL-R. $\sigma_1 = \text{ID} \circ \text{ID}$ and $\sigma_2 = \text{ID} \circ \sigma \circ \text{ID}$. Trivial. $\sigma_1 \rightsquigarrow^* \text{ID}$ and $\sigma_2 \rightsquigarrow^* \sigma$.

Case ID-CANCEL-L/R *and* INVERSES-CANCEL. $\sigma_1 = \text{ID} \circ \sigma \circ \sigma^{-1}$ and $\sigma_2 = \sigma \circ \sigma^{-1} \circ \text{ID}$. Trivial. $\sigma_1 \rightsquigarrow^* \text{ID}$ and $\sigma_2 \rightsquigarrow^* \text{ID}$, either by first applying ID-CANCEL and then INVERSES-CANCEL or the other way around.

Case ID-CANCEL-L/R *and* HYLO-CANCEL. $\sigma_1 = \text{ID} \circ \text{HYLO}_F\ \text{IN}\ \text{OUT}$ and $\sigma_2 = \text{HYLO}_F\ \text{IN}\ \text{OUT} \circ \text{ID}$. Trivial. $\sigma_1 \rightsquigarrow^* \text{ID}$ and $\sigma_2 \rightsquigarrow^* \text{ID}$.

Case ID-CANCEL-L/R *and* F-CANCEL. We do not show the symmetric case. Trivial. $\sigma_1 = \text{ID} \circ F\ \text{ID}$ reduces to ID.
Case ID-CANCEL-L/R *and* F-SPLIT.

> *Case 1.* $\sigma_1 = F\ (\text{ID} \circ \sigma)$ and $\sigma_2 = F\ (\sigma \circ \text{ID})$. We solve σ_1, the other case is symmetric. By applying F-SPLIT, then F-ID-CANCEL-CANCEL, then ID-CANCEL-L, we get $F\ (\text{ID} \circ \sigma) \rightsquigarrow F\ \text{ID} \circ F\ \sigma \rightsquigarrow \text{ID} \circ F\ \sigma \rightsquigarrow F\ \sigma$. By applying ID-CANCEL-L, we get $F\ (\text{ID} \circ \sigma) \rightsquigarrow F\ \sigma$.

> *Case 2.* $\sigma_1 = \text{ID} \circ F\ (\sigma \circ \sigma')$ and $\sigma_2 = F\ (\sigma \circ \sigma') \circ \text{ID}$. Trivial. $\sigma_1 \rightsquigarrow F\ \sigma \circ F\ \sigma'$ and $\sigma_2 \rightsquigarrow F\ \sigma \circ F\ \sigma'$.

Case ID-CANCEL-L/R *and* ANA-MAP. Trivial. $\sigma_1 = \text{ANA}_F\ (F\ (\text{ID} \circ \sigma) \circ \text{OUT})$, $\sigma_2 = \text{ANA}_F\ (F\ (\sigma \circ F) \circ \text{OUT})$, $\sigma_3 = \text{ID} \circ \text{ANA}_F\ (F\ \sigma \circ \text{OUT})$, $\sigma_4 = \text{ANA}_F\ (F\ \sigma \circ \text{OUT}) \circ \text{ID}$. All of these trivially reduce to $\text{MAP}_F\ \sigma$.

Case ID-CANCEL-L/R *and* HYLO-SPLIT. Trivial. For simplicity, we don not show the symmetric cases. $\sigma_1 = \text{HYLO}_F\ (\text{ID} \circ \sigma)\ \sigma'$, $\sigma_2 = \text{HYLO}_F\ \sigma\ (\text{ID} \circ \sigma')$ and $\sigma_3 = \text{ID} \circ \text{HYLO}_F\ \sigma\ \sigma'$. All of these reduce to $\text{HYLO}_F\ \sigma\ \sigma'$.

Case ID-CANCEL-L/R *and* CATA-SPLIT. Trivial. Same as HYLO-SPLIT.

Case ID-CANCEL-L/R *and* ANA-SPLIT. Trivial. Same as HYLO-SPLIT.

Cases arising from INVERSES-CANCEL. Trivial, similar to the ID-CANCEL-L/R cases. The only exception is the critical pair arising from F-SPLIT, $\sigma = F\ (\sigma \circ \sigma^{-1})$. By INVERSES-CANCEL, this reduces to ID. By F-SPLIT, we have to use the rule that $F\ \sigma \circ F\ \sigma^{-1} \rightsquigarrow \text{ID}$. Since $F\ \sigma$ and $F\ \sigma^{-1}$ are inverses, this reduces to ID.

Case F-SPLIT *and* ANA-MAP. $\sigma = \text{ANA}_F\ (F\ (\sigma_1 \circ \sigma_2) \circ \text{OUT})$.

> By F-SPLIT, $\sigma \rightsquigarrow \text{ANA}_F\ (F\ \sigma_1 \circ F\ \sigma_2 \circ \text{OUT})$. We proceed by ANA-SPLIT and ANA-MAP: $\text{ANA}_F\ (F\ \sigma_1 \circ F\ \sigma_2 \circ \text{OUT}) \rightsquigarrow \text{MAP}_F\ \sigma_1 \circ \text{ANA}_F\ (F\ \sigma_2 \circ \text{OUT}) \rightsquigarrow \text{MAP}_F\ \sigma_1 \circ \text{MAP}_F\ \sigma_2$.
>
> By ANA-MAP, $\sigma \rightsquigarrow \text{MAP}_F\ (\sigma_1 \circ \sigma_2)$. We proceed by CATA-SPLIT, $\text{MAP}_F\ (\sigma_1 \circ \sigma_2) \rightsquigarrow \text{MAP}_F\ \sigma_1 \circ \text{MAP}_F\ \sigma_2$.

Case F-SPLIT *and* CATA-SPLIT. $\sigma = \text{CATA}_F\ (\sigma_1 \circ F\ (\sigma_2 \circ \sigma_3))$.

> By F-SPLIT, $\sigma \rightsquigarrow \text{CATA}_F\ (\sigma_1 \circ F\ \sigma_2 \circ F\ \sigma_3)$. We proceed by CATA-SPLIT: $\text{CATA}_F\ (\sigma_1 \circ F\ \sigma_2 \circ F\ \sigma_3) \rightsquigarrow \text{CATA}_F\ (\sigma_1 \circ F\ \sigma_2) \circ \text{MAP}_F\sigma_3$. If $\sigma_1 = \text{IN}$, then we have finished with $\text{MAP}_F\ \sigma_2 \circ \text{MAP}_F\sigma_3$, otherwise we perform another CATA-SPLIT $\text{CATA}_F\ \sigma_1 \circ \text{MAP}_F\ \sigma_2 \circ \text{MAP}_F\sigma_3$.
>
> By CATA-SPLIT, $\sigma \rightsquigarrow \text{CATA}_F\ \sigma_1 \circ \text{MAP}_F\ (\sigma_2 \circ \sigma_3)$. This implies that σ_1 cannot be IN. We finish by applying F-SPLIT and CATA-SPLIT. $\text{MAP}_F\ (\sigma_2 \circ \sigma_3) \rightsquigarrow \text{MAP}_F\ \sigma_2 \circ \text{MAP}_F\ \sigma_3$

Case F-SPLIT *and* ANA-SPLIT. $\sigma = \text{ANA}_F\ (F\ (\sigma_2 \circ \sigma_3) \circ \sigma_1)$.

> By F-SPLIT, $\sigma \rightsquigarrow \text{ANA}_F\ (F\ \sigma_2 \circ F\ \sigma_3 \circ \sigma_1)$. We proceed by ANA-SPLIT: $\sigma \rightsquigarrow^*\ \text{MAP}_F\ \sigma_2 \circ \text{MAP}_F\sigma_3 \circ \text{ANA}_F\ \sigma_1$.
>
> By ANA-SPLIT, $\sigma \rightsquigarrow \text{MAP}_F\ (\sigma_2 \circ \sigma_3) \circ \text{ANA}_F\ \sigma_1$. By applying F-SPLIT and CATA-SPLIT. $\text{MAP}_F\ (\sigma_2 \circ \sigma_3) \circ \text{ANA}_F\ \sigma_1 \rightsquigarrow \text{MAP}_F\ \sigma_2 \circ \text{MAP}_F\ \sigma_3 \circ \text{ANA}_F\ \sigma_1$.

Case F-SPLIT *and* HYLO-SPLIT. The critical pairs are obtained from $\sigma = \text{HYLO}_F\ (\sigma_1 \circ F\ (\sigma_2 \circ \sigma_3))\ \sigma_4$ and $\sigma' = \text{HYLO}_F\ \sigma_1\ (F\ (\sigma_2 \circ \sigma_3) \circ \sigma_4)$. We only show the case for σ, since σ' can be derived in an analogous way.

> By first F-SPLIT and then HYLO-SPLIT, $\sigma \rightsquigarrow \text{HYLO}_F\ (\sigma_1 \circ F\ \sigma_2 \circ F\ \sigma_3)\ \sigma_4 \rightsquigarrow \text{CATA}_F\ (\sigma_1 \circ F\ \sigma_2 \circ F\ \sigma_3) \circ \text{ANA}_F\ \sigma_4$
>
> By first HYLO-SPLIT and then F-SPLIT, $\sigma \rightsquigarrow \text{CATA}_F\ (\sigma_1 \circ F\ (\sigma_2 \circ \sigma_3)) \circ \text{ANA}_F\ \sigma_4 \rightsquigarrow \text{CATA}_F\ (\sigma_1 \circ F\ \sigma_2 \circ F\ \sigma_3) \circ \text{ANA}_F\ \sigma_4$.

Cases arising from HYLO-CANCEL. Trivial due to the preconditions of the rules HYLO-SPLIT, CATA-SPLIT and ANA-SPLIT.

Case F-ID-CANCEL *and* ANA-MAP. $\sigma = \text{ANA}_F\ (F\ \text{ID} \circ \text{OUT})$

> By F-ID-CANCEL , $\text{ANA}_F\ (F\ \text{ID} \circ \text{OUT}) \rightsquigarrow \text{ANA}_F\ \text{OUT}$. Then, by HYLO-CANCEL, $\text{ANA}_F\ \text{OUT} \rightsquigarrow \text{ID}$.
>
> By ANA-MAP, $\text{ANA}_F\ (F\ \text{ID} \circ \text{OUT}) \rightsquigarrow \text{MAP}_F\ \text{ID}$. MAP_F is a synonym to a hylomorphism, so $\text{MAP}_F\ \text{ID}$ is the following term: $\text{MAP}_F\ \text{ID} = \text{HYLO}_F\ (\text{IN} \circ F\ \text{ID})\ \text{OUT}$. By F-ID-CANCEL, following by ID-CANCEL-L and HYLO-CANCEL, $\text{MAP}_F\ ID \rightsquigarrow \text{ID}$.

Case F-ID-CANCEL *and* CATA-SPLIT. $\sigma = \text{CATA}_F\ (\sigma_1 \circ F\ \text{ID})$

> By F-ID-CANCEL, $\sigma \rightsquigarrow \text{CATA}_F\ \sigma_1$
>
> By CATA-SPLIT, $\sigma \rightsquigarrow \text{CATA}_F\ \sigma_1 \circ \text{MAP}_F\ \text{ID}$. By F-ID-CANCEL, following by ID-CANCEL-R and HYLO-CANCEL, $\text{CATA}_F\ \sigma_1 \circ \text{MAP}_F\ \text{ID} \rightsquigarrow^*\ \text{CATA}_F\ \sigma_1$.

Case F-ID-CANCEL *and* ANA-SPLIT. $\sigma = \text{ANA}_F\ (F\ \text{ID} \circ \sigma_1)$

> By F-ID-CANCEL, $\sigma \rightsquigarrow \text{ANA}_F\ \sigma_1$
>
> By ANA-SPLIT, $\sigma \rightsquigarrow \text{MAP}_F\ \text{ID} \circ \text{ANA}_F\ \sigma_1$. By F-ID-CANCEL, following by ID-CANCEL-L and HYLO-CANCEL, $\text{MAP}_F\ \text{ID} \circ \text{ANA}_F\ \sigma_1 \rightsquigarrow^*\ \text{ANA}_F\ \sigma_1$.

Case ANA-MAP. No critical pairs. The only possibility would be HYLO/CATA/ANA-SPLIT. However, the preconditions of the rules, specifically $\sigma_1 \neq \text{IN}$ in HYLO-SPLIT/CATA-SPLIT and $\sigma_2 \neq \text{OUT}$ in ANA-SPLIT prevent their application.

Case HYLO-SPLIT *and* CATA-SPLIT. No critical pairs, due to the preconditions if the rules $\sigma_1 \neq$ IN.

Case HYLO-SPLIT *and* ANA-SPLIT. No critical pairs, due to the preconditions if the rules $\sigma_2 \neq$ OUT.

Case CATA-SPLIT *and* ANA-SPLIT. No critical pairs.

□

Appendix B

Soundness of the Structure-Checking Relation

This is a proof of Theorem 5.4.1 on page 174. By induction on the structure of the derivation $\Delta; \Gamma^n \vdash M \sim a$.

Case / GVar: $M = x$. The premise is $\Delta; \Gamma^n \vdash x \sim \langle [x] \rangle^n$. The goal is $\lambda\overline{\Gamma}.x \approx \lambda[\![\langle [x] \rangle^n]\!]$. By unfolding $\lambda[\![\cdot]\!]$,

$$\lambda[\![\langle [x] \rangle^n]\!] = \lambda y_1 \cdots y_n.x$$

By α-renaming y_i to $\overline{\Gamma}$,

$$\lambda[\![\langle [x] \rangle^n]\!] \approx_\alpha \lambda\overline{\Gamma}.x$$

Case / LVar: $M = x$.

The premise is $\Gamma^n \vdash x \sim \langle \mathtt{I} \rangle^{i+j+1} \; |^i_j$. The goal is $\lambda\overline{\Gamma}.x =_\alpha \lambda[\![\langle\rangle^{i+j+1} \; |^i_j]\!]$. The initial assumption implies that $\Gamma^n(x) = A^k$, which in turn implies $k \leq n$ and that $i = k - 1$ and $j = n - k$. By expanding $\lambda[\![\langle\rangle^{i+j+1} \; |^i_j]\!]$:

$$\begin{aligned}
&\lambda[\![\langle\rangle^{i+j+1} \; |^i_j]\!] \\
&= \lambda[\![\langle\langle\rangle^j\rangle^i_0]\!] \\
&= \lambda y_1 \cdots y_i.\lambda[\![\langle\langle\rangle^j\rangle^0_0]\!] \\
&= \lambda y_1 \cdots y_i.\lambda[\![\langle\rangle^j]\!] \\
&= \lambda y_1 \cdots y_i \; y_{i+1}.\lambda[\![\langle [y_{i+1}] \rangle^j_0]\!] \\
&= \lambda y_1 \cdots y_i \; y_{i+1} \; y_{i+2} \cdots y_{i+1+j}.\lambda[\![\langle [y_{i+1}] \rangle^0_0]\!] \\
&= \lambda y_1 \cdots y_i \; y_{i+1} \; y_{i+2} \cdots y_{i+1+j}.\lambda[\![[y_{i+1}]]\!] \\
&= \lambda y_1 \cdots y_i \; y_{i+1} \; y_{i+2} \cdots y_{i+1+j}.y_{i+1}
\end{aligned}$$

Note that $i + 1 + j = (k - 1) + 1 + (n - k) = n$. By expanding Γ,

$$\Gamma = [x_1 : A_1 \cdots x_i : A_i, x : A, z_1 : B_1 \cdots z_j B_j].$$

The variables $y_1 \cdots y_i$ are α-renamed to x_i, y_{i+1} to x, and $y_{i+2} \cdots y_n$ to $z_1 \cdots z_j$:

$$\begin{array}{rcl} \lambda[\![\langle\rangle^{i+j+1} \mid_j^i]\!] & = & \lambda y_1 \cdots y_i \; y_{i+1} \; y_{i+2} \cdots y_n . y_{i+1} \\ & \approx_\alpha & \lambda x_1 \cdots x_i \; x \; z_1 \cdots z_n . x = \lambda \overline{\Gamma} . x \end{array}$$

Case / Abs: $M = \lambda x.N$.

The premise is $\Gamma^n \vdash \lambda x.N \sim a$. The goal is $\lambda[\![a]\!] \approx \lambda \overline{\Gamma} \; \lambda x.N$. If $\Gamma^n \vdash \lambda x.N \sim a$, then $\Gamma^n, x : A \vdash N \sim a$. By the induction hypothesis, $\lambda[\![a]\!] \approx \lambda \overline{\Gamma, x : A}.N = \lambda \overline{\Gamma} \; x.N = \lambda \overline{\Gamma}.\lambda x.N$

Case / App: $M = N_1 \; N_2$.

The premise is $\Gamma^n \vdash N_1 \; N_2 \sim @^n \; a_1 \; a_2$. The goal is $\lambda[\![@^n \; a_1 \; a_2]\!] = \lambda \overline{\Gamma}.N_1 \; N_2$. By reducing $\lambda[\![@^n \; a_1 \; a_2]\!]$,

$$\begin{array}{l} \lambda[\![@^n \; a_1 \; a_2]\!] \\ = \lambda x_1 \cdots x_n . \lambda[\![@^0 \; (a_1 \; [x_1] \cdots [x_n]) \; (a_1 \; [x_1] \cdots [x_n])]\!] \\ = \lambda x_1 \cdots x_n . \lambda[\![a_1 \; [x_1] \cdots [x_n] \; (a_2 \; [x_1] \cdots [x_n])]\!] \\ = \lambda x_1 \cdots x_n . \lambda[\![a_1 \; [x_1] \cdots [x_n]]\!] \; \lambda[\![a_2 \; [x_1] \cdots [x_n]]\!] \\ = \lambda x_1 \cdots x_n . (\lambda[\![a_1]\!] \; x_1 \cdots x_n) \; (\lambda[\![a_2]\!] \; x_1 \cdots x_n) \end{array}$$

The assumption $\Gamma^n \vdash N_1 \; N_2 \sim @^n \; a_1 \; a_2$ implies $\Gamma^n \vdash N_1 \sim a_1$ and $\Gamma^n \vdash N_2 \sim a_2$. By the induction hypothesis,

$$\begin{array}{l} \lambda[\![@^n \; a_1 \; a_2]\!] \\ \approx \lambda x_1 \cdots x_n . ((\lambda \overline{\Gamma}.N_1) \; x_1 \cdots x_n) \; ((\lambda \overline{\Gamma}.N_2) \; x_1 \cdots x_n) \end{array}$$

Since none of the x_i occur free in N_i, by α-renaming $x_1 \cdots x_n$:

$$\begin{array}{l} \lambda[\![@^n \; a_1 \; a_2]\!] \\ \approx_\alpha \lambda \overline{\Gamma} . ((\lambda \overline{\Gamma}.N_1) \; \overline{\Gamma}) \; ((\lambda \overline{\Gamma}.N_2) \; \overline{\Gamma}) \end{array}$$

Finally, by η-reduction:

$$\begin{array}{l} \lambda[\![@^n \; a_1 \; a_2]\!] \\ \approx \lambda x_1 \cdots x_n . ((\lambda \overline{\Gamma}.N_1) \; x_1 \cdots x_n) \; ((\lambda \overline{\Gamma}.N_2) \; x_1 \cdots x_n) \\ \approx_\alpha \lambda \overline{\Gamma} . ((\lambda \overline{\Gamma}.N_1) \; \overline{\Gamma}) \; ((\lambda \overline{\Gamma}.N_2) \; \overline{\Gamma}) \\ \approx_\eta \lambda \overline{\Gamma}.N_1 \; N_2 \end{array}$$

***Case* / *Case*: $M = \texttt{case}\ M\ \texttt{of}\ \{p_1 \to N_1; \ldots; p_k \to N_k\}$.**

The premise is

$$\begin{aligned}\Gamma^n \vdash\ & \texttt{case}\ M\ \texttt{of}\ \{p_1 \to N_1; \cdots; p_k \to N_k\} \\ & \sim \langle \nabla_k \rangle^n_{k+1}\ a_1 \cdots a_k\ (\langle [\texttt{out}(p_1 \cdots p_k)] \rangle^n_1\ a_M).\end{aligned}$$

The goal is

$$\begin{aligned}& \lambda[\![\langle \nabla_k \rangle^n_{k+1}\ a_1 \cdots a_k\ (\langle [\texttt{out}(p_1 \cdots p_k)] \rangle^n_1\ a_M)]\!] \\ & \approx \lambda\overline{\Gamma}.\texttt{case}\ M\ \texttt{of}\ \{p_1 \to N_1; \ldots; p_k \to N_k\}.\end{aligned}$$

By the definition of $\lambda[\![\cdot]\!]$, the LHS of the goal is simplified to:

$$\begin{aligned}& \lambda[\![\langle \nabla_k \rangle^n_{k+1}\ a_1 \cdots a_k\ (\langle [\texttt{out}(p_1 \cdots p_k)] \rangle^n_1\ a_M)]\!] \\ & \quad = (\lambda x_1 \cdots x_n.\lambda[\![\langle \nabla_k \rangle^0_{k+1}\ (a_1\ [x_1] \cdots [x_n]) \cdots (a_k\ [x_1] \cdots [x_n]) \\ & \qquad\qquad (\langle [\texttt{out}(p_1 \cdots p_k)] \rangle^0_1\ (a_M\ [x_1] \cdots [x_n]))]\!] \\ & \quad \approx_\beta (\lambda x_1 \cdots x_n.\ (\lambda[\![a_1]\!]\ x_1 \cdots x_n \nabla \cdots \nabla \lambda[\![a_k]\!]\ x_1 \cdots x_n) \\ & \qquad\qquad (\texttt{out}(p_1 \cdots p_k)\ (\lambda[\![a_M]\!]\ x_1 \cdots x_n))\end{aligned}$$

By unfolding `out` and β-reducing the resulting λ,

$$\begin{aligned}\approx_\beta (\lambda x_1 \cdots x_n.\ & (\lambda[\![a_1]\!]\ x_1 \cdots x_n \nabla \cdots \nabla \lambda[\![a_k]\!]\ x_1 \cdots x_n) \\ & \left(\begin{array}{l}\texttt{case}\ (\lambda[\![a_M]\!]\ x_1 \cdots x_n)\ \texttt{of}\{ \\ \qquad p_1 \to \texttt{inj}_1\ \texttt{TP}[\![p_1]\!]; \\ \qquad \cdots; \\ \qquad p_k \to \texttt{inj}_1\ \texttt{TP}[\![p_1]\!]\}\end{array}\right)\end{aligned}$$

By unfolding ∇, and doing a `case-case` merge:

$$\begin{aligned}\approx_{\texttt{case-case}} (\lambda x_1 \cdots x_n.\ & \texttt{case}\ (\lambda[\![a_M]\!]\ x_1 \cdots x_n)\ \texttt{of}\{ \\ & p_1 \to \lambda[\![a_1]\!]\ x_1 \cdots x_n\ \texttt{TP}[\![p_1]\!]; \\ & \cdots; \\ & p_k \to \lambda[\![a_k]\!]\ x_1 \cdots x_n\ \texttt{TP}[\![p_k]\!]\}\end{aligned}$$

By applying the *Case* inference rule to the premise, the following derivations are obtained:

$$\begin{aligned}& \Gamma^n \vdash M \sim a_M \\ & \Gamma^n, x : T' \vdash N_i[p_i \downarrow x] \sim a_i\end{aligned}$$

By applying the induction hypothesis,

$$\begin{aligned}\approx \lambda x_1 \cdots x_n.\ \texttt{case}\ ((\lambda\overline{\Gamma}.M)\ x_1 \cdots x_n)\ \texttt{of}\{\\ & p_1 \to (\lambda\overline{\Gamma}\ x.N_1[p_1 \downarrow x])\ x_1 \cdots x_n\ \texttt{TP}[\![p_1]\!];\\ & \cdots;\\ & p_k \to (\lambda\overline{\Gamma}\ x.N_k[p_k \downarrow x])\ x_1 \cdots x_n\ \texttt{TP}[\![p_k]\!]\}\end{aligned}$$

By first α-renaming the x_i to $\overline{\Gamma}$, and then η-reducing,

$$\begin{aligned}\approx_{\alpha\eta} (\lambda\overline{\Gamma}.\ \texttt{case}\ M\ \texttt{of}\{\\ & p_1 \to (\lambda x.N_1[p_1 \downarrow x])\ \texttt{TP}[\![p_1]\!];\\ & \cdots;\\ & p_k \to (\lambda x.N_k[p_k \downarrow x])\ \texttt{TP}[\![p_k]\!]\}\end{aligned}$$

Finally, by Lemma 5.3.1:

$$\approx_\beta \lambda\overline{\Gamma}.\ \texttt{case}\ M\ \texttt{of}\{\ p_1 \to N_1;\ \cdots;\ p_k \to N_k\}$$

Case / Let: $\boldsymbol{M = \texttt{let}\ x = M\ \texttt{in}\ N}$.

The premise is

$$\Delta; \Gamma^n \vdash \texttt{let}\ x = M\ \texttt{in}\ N : @^n\ a_2\ (\texttt{D}^n_x\ \Gamma\ a_1) \sim B$$

The goal is

$$\lambda[\![@^n\ a_2\ (\texttt{D}^n_x\ \Gamma\ a_1)]\!] \approx \lambda\overline{\Gamma}.\texttt{let}\ x = M\ \texttt{in}\ N.$$

The goal can be rewritten to:

$$\lambda[\![@^n\ a_2\ (\texttt{D}^n_x\ \Gamma\ a_1)]\!] \approx \lambda x_1 \cdots x_n.\lambda[\![a_2]\!]\ x_1 \cdots x_n\ (\lambda[\![\texttt{D}^n_x\ \Gamma\ a_1]\!]\ x_1 \cdots x_n)$$

The *Let* rule implies that:

$$\Delta, \Gamma, x : A; [] \vdash M : a_1 \sim A \qquad \Delta; \Gamma^n, x : A \vdash N : a_2 \sim B.$$

By the induction hypothesis,

$$\lambda[\![\texttt{D}^n_x\ \Gamma\ a_1]\!] \approx \lambda\overline{\Gamma}.\texttt{Y}\ (\lambda x.\lambda[\![a_1]\!]) \approx \lambda\overline{\Gamma}.\texttt{Y}\ (\lambda x.M) \qquad \lambda[\![a_2]\!] \approx \lambda\overline{\Gamma}\ x.N.$$

This can be used to further rewrite the goal. Followed by α renaming and η-reduction,

$$\lambda[\![@^n\ a_2\ (\texttt{D}^n_x\ \Gamma\ a_1)]\!] \approx \ldots \approx_{\alpha\eta} \lambda\overline{\Gamma}.(\lambda x.N)(\texttt{Y}\ (\lambda x.M))$$

Since $\texttt{let}\ x = M\ \texttt{in}\ N \approx N[M/x] \approx (\lambda x.N)\ M$,

$$\lambda[\![@^n\ a_2\ (\mathtt{D}^n_x\ \Gamma\ a_1)]\!] \approx \ldots \approx \lambda\overline{\Gamma}.\texttt{let}\ x = \mathtt{Y}\ (\lambda x.M)\ \texttt{in}\ N$$

Both x must be the same, since they both come from the premise. The semantics of a recursive let-in definition $\texttt{let}\ x = M$ is defined as $\mathtt{Y}\ (\lambda x.M)$. Also, if $x \notin \mathsf{fv}(M)$, then $\mathtt{Y}\ (\lambda x.M) \approx M$. This finishes the proof:

$$\lambda[\![@^n\ a_2\ (\mathtt{D}^n_x\ \Gamma\ a_1)]\!] \approx \ldots \approx \lambda\overline{\Gamma}.\texttt{let}\ x = M\ \texttt{in}\ N$$

□

Bibliography

[AAG03] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew D. Gordon, editor, *Foundations of Software Science and Computation Structures*, pages 23–38, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[ABB+16] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. Behavioral Types in Programming Languages. *Foundations and Trends® in Programming Languages*, 3(2-3):95–230, 2016.

[AD07] Marco Aldinucci and Marco Danelutto. Skeleton-based Parallel Programming: Functional and Parallel Semantics in a Single Shot. *Computer Languages, Systems & Structures*, 33(3):179–192, 2007.

[ADK+11] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating Code on Multi-cores with Fastflow. In *European Conference on Parallel Processing*, pages 170–181. Springer, 2011.

[ADKT13] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: High-Level and Efficient Streaming on Multi-Core. *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2013.

[ADT03] M. Aldinucci, M. Danelutto, and P. Teti. An Advanced Environment Supporting Structured Parallel Programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.

[AGLP01] Marco Aldinucci, Sergei Gorlatch, Christian Lengauer, and Susanna Pelagatti. Towards Parallel Programming by Transformation: The FAN Skeleton Framework. *PARALLEL ALGORITHMS AND APPLICATION*, 16(2):87–121, 2001.

[AMT11] Khari Armih, Greg Michaelson, and Phil Trinder. Cache Size in a Cost Model for Heterogeneous Skeletons. In *Proceedings of the fifth international workshop on High-level parallel programming and applications*, pages 3–10. ACM, 2011.

[AV90] Joe L Armstrong and SR Virding. Erlang - an Experimental Telephony Programming Language. In *Switching Symposium, 1990. XIII International*, volume 3, pages 43–48. IEEE, 1990.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a Nonprocedural Language with Iteration. *Comm. ACM*, 20(7):519–526, July 1977.

[Awo10] Steve Awodey. *Category theory*. Oxford University Press, 2010.

[Bar84] HP Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1984.

[Bar17] Adam Barwell. *Program Shaping for Parallelism*. PhD thesis, University of St Andrews, 2017.

[BBCH16] Adam D Barwell, Christopher Brown, David Castro, and Kevin Hammond. Towards Semi-Automatic Data-Type Translation for Parallelism in Erlang. In *Proceedings of the 15th International Workshop on Erlang*, pages 60–61, Nara,Japan, 2016. ACM.

[BC13] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: the Calculus of Inductive Constructions*. Springer Science & Business Media, 2013.

[BCD+97] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, and M. Vanneschi. An environment for structured parallel programming. In Lucio Grandinetti, Janusz Kowalik, and Marian Vajtersic, editors, *Advances in High Performance Computing*, pages 219–234. Springer Netherlands, Dordrecht, 1997.

[BCGH05] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible Skeletal Programming with eSkel. In *European Conference on Parallel Processing*, pages 761–770. Springer, 2005.

[BCP05] Manuel Barbosa, Alcino Cunha, and Jorge Sousa Pinto. Recursion Patterns and Time-analysis. *ACM SIGPLAN Notices*, 40(5):45–54, 2005.

[BDH+13] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick, and Archibald Elliott. Cost-directed refactoring for parallel Erlang programs. *International Journal of Parallel Programming*, 42(4):1–19, 2013.

[BDO+95] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. P3L: A Structured High-Level Parallel Language, and its Structured Support. *Concurrency: practice and experience*, 7(3):225–255, 1995.

[BFH+14] István Bozó, Viktoria Fordós, Zoltán Horvath, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Köszegi, Adam Barwell, Christopher Brown, and Kevin Hammond. Discovering Parallel Pattern Candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 13–23. ACM, 2014.

[BGR08] Gilles Barthe, Benjamin Grégoire, and Colin Riba. Type-Based Termination with Sized Products. In *International Workshop on Computer Science Logic*, pages 493–507. Springer, 2008.

[BH05] Edwin Brady and Kevin Hammond. A dependently typed framework for static analysis of program execution costs. In *Symposium on Implementation and Application of Functional Languages*, pages 74–90. Springer, 2005.

[BH15] Patrick Bahr and Graham Hutton. Calculating Correct Compilers. *Journal of Functional Programming*, 25, 2015.

[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.

[Bir88] Richard Bird. A Calculus of Functions for Program Derivation. Technical report, Programming Research Group, Oxford University, 11, Keble Road, Oxford, OX1 3QD, U.K., 1988.

[Bir89] Richard S Bird. Lectures on Constructive Functional Programming. In *Constructive Methods in Computing Science*, pages 151–217. Springer, 1989.

[BJK+96] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: an Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[BK95] George Botorog and Herbert Kuchen. Algorithmic Skeletons for Adaptive Multigrid Methods. *Parallel Algorithms for Irregularly Structured Problems*, pages 27–41, 1995.

[BL14] Thomas W. Bartenstein and Yu David Liu. Rate Types for Stream Programs. In *Proc. OOPSLA 2014: Intl. Conf. on Obj. Oriented Prog. Systs. Langs. & Appls.*, pages 213–232, 2014.

[Ble95] Guy E Blelloch. Nesl: A nested data-parallel language.(version 3.1). Technical report, DTIC Document, 1995.

[BLM90] Jan Biemond, Reginald L Lagendijk, and Russell M Mersereau. Iterative Methods for Image Deblurring. *Proceedings of the IEEE*, 78(5):856–883, 1990.

[BLOMPM96] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. Eden—the Paradise of Functional Concurrent Programming. In *Euro-Par'96 Parallel Processing*, pages 710–713. Springer, 1996.

[BN98] F Baader and T Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bra94] Tore Andreas Bratvold. *Skeleton-based Parallelisation of Functional Programs*. PhD thesis, Heriot-Watt University, 1994.

[Bra13] Edwin Brady. Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[Bra17] Edwin Brady. Type-driven Development of Concurrent Communicating Systems. *Computer Science*, 18(3), 2017.

[Bro13] Nick Brown. Applying Type Oriented Programming to the PGAS Memory Model. In *7th International Conference on PGAS Programming Models*, page 93, 2013.

[Bro14] Nick Brown. A Type-Oriented Graph500 Benchmark. In *International Supercomputing Conference*, pages 460–469. Springer, 2014.

[Bro17] Nick Brown. Type oriented parallel programming for Exascale. *Advances in Engineering Software*, 2017.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*, volume 1. Prentice Hall New York, 1988.

[CDE+06] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous Kahn Networks. In *Proc. POPL '06: ACM Symposium on Principles of Programming Languages*, pages 180–193, 2006.

[CFC58] Haskell Brooks Curry, Robert Feys, William Craig, and William , Craig. *Combinatory Logic, vol. 1.* North-Holland Publ., 1958.

[CH14] David Castro and Kevin Hammond. Skeletor: A DSL for Describing Type-based Specifications of Parallel Skeletons. In *Proc. Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU 2014)*, 2014.

[CHS16] David Castro, Kevin Hammond, and Susmit Sarkar. Farms, Pipes, Streams and Reforestation: Reasoning About Structured Parallel Processes using Types and Hylomorphisms. In *Proceedings of ICFP 2016: ACM International Conference on Functional Programming*, pages 4–17, Nara, Japan, September 2016.

[CHSA17] David Castro, Kevin Hammond, Susmit Sarkar, and Yasir Alguwaifli. Automatically deriving Cost Models for Structured Parallel Processes using Hylomorphisms. *Future Generation Computer Systems*, 2017.

[CLPJ+07] Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[CM11] Yun-Yan Chi and Shin-Cheng Mu. Constructing List Homomorphisms from Proofs. In *Proc. APLAS '11: Asian Symposium on Programming Languages & Systems*, pages 74–88. Springer, 2011.

[Col89] Murray Irvin Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, London, 1989.

[Col93] Murray Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. In *Proceedings of Parco 93. Elsevier Series in Advances in Parallel Computing*, 1993.

[Col04] Murray Cole. Bringing Skeletons out of the Closet: a Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389 – 406, 2004.

[Coo12] Shane Cook. *CUDA Programming: a Developer's Guide to Parallel Computing with GPUs*. Newnes, 2012.

[CP95] Paul Caspi and Marc Pouzet. A Functional Extension to Lustre. In *Proc. ISLIP '95: Intl. Symp. on Langs. for Intensional Programming*, Sydney, Australia, May 1995.

[CP96] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *Proc. ICFP '96: ACM Conf. on Functional Programming*, pages 226–238, Philadelphia, PA, May 1996.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Real-time Programming. In *Proc. POPL '87: ACM Symposium on Principles of Programming Languages*, pages 178–188, 1987.

[CPP05] Alcino Cunha, Jorge Sousa Pinto, and José Proença. A Framework for Point-free Program Transformation. In *Symp. on Implementation and Application of Functional Languages*, pages 1–18. Springer, 2005.

[CPP06] Alcino Cunha, Jorge Sousa Pinto, and José Proença. A framework for point-free program transformation. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages: 17th International Workshop, IFL 2005, Dublin, Ireland,*

September 19-21, 2005, Revised Selected Papers, pages 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[CT09] Francesco Cesarini and Simon Thompson. *Erlang Programming: a Concurrent Approach to Software Development.* " O'Reilly Media, Inc.", 2009.

[Cun05] Alcino Cunha. *Point-free Program Calculation.* PhD thesis, Departamento de Informática, Universidade do Minho, 2005.

[Cur30] H. B. Curry. Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52(3):509–536, 1930.

[DD06] Marco Danelutto and Patrizio Dazzi. Joint Structured/Unstructured Parallelism Exploitation in Muskel. In *International Conference on Computational Science*, pages 937–944. Springer, 2006.

[DFH+93] John Darlington, Anthony Field, Peter Harrison, Paul Kelly, David Sharp, Qian Wu, and R While. Parallel Programming using Skeleton Functions. In *PARLE'93 Parallel Architectures and Languages Europe*, pages 146–160. Springer, 1993.

[DGTY95] John Darlington, Yi-ke Guo, Hing To, and Jin Yang. Functional Skeletons for Parallel Coordination. *EURO-PAR'95 Parallel Processing*, pages 55–66, 1995.

[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.

[DMCN12] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A Survey of Parallel Programming Models and Tools in the

Multi and Many-core Era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

[DT13] M. Danelutto and M. Torquati. A RISC Building Block Set for Structured Parallel Programming. In *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '13)*, pages 46–50, 2013.

[EBDH12] A Elliott, C Brown, M Danelutto, and K Hammond. Skel: A Streaming Process-based Skeleton Library for Erlang. In *24th Symposium on Implementation and Application of Functional Languages, IFL*, 2012.

[FG02] Jörg Fischer and Sergei Gorlatch. Turing Universality of Recursive Patterns for Parallel Programming. *Parallel Processing Letters*, 12(02):229–246, 2002.

[FKNS14] Vojtěch Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs. In *International Symposium on Formal Methods*, pages 263–278. Springer, 2014.

[Fly72] Michael J Flynn. Some Computer Organizations and their Effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[FM91] Maarten M Fokkinga and Erik Meijer. Program Calculation Properties of Continuous Algebras. Technical Report, CWI, 1991.

[FS08] Joel Falcou and Jocelyn Sérot. Formal Semantics Applied to the Implementation of a Skeleton-Based Parallel Programming Library. *Parallel Computing: Architectures, Algorithms and Applications (Proc. of PARCO 2007, Julich, Germany)*, 38:243–252, 2008.

[FW78] Steven Fortune and James Wyllie. Parallelism in Random Access Machines. In *Proc. STOC '78: 10th Symp. on Theory of Computing*, pages 114–118, 1978.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–, February 1992.

[GDTF17] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005, 2017.

[GG99] Alfons Geser and Sergei Gorlatch. Parallelizing Functional Programs by Generalization. *Journal of Functional Programming (JFP)*, 9(06):649–673, 1999.

[GGL12] Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Toward static timing analysis of parallel software. In *OASIcs-OpenAccess Series in Informatics*, volume 23. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[GGP+15] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Static analysis of energy consumption for llvm ir programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, pages 12–21. ACM, 2015.

[GH09] Andy Gill and Graham Hutton. The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(2):227–251, 2009.

[Gha95] Neil Ghani. $\beta\eta$-Equality for Coproducts. In *Proc. International Conference on Typed Lambda Calculi and Applications*, pages 171–185. Springer-Verlag, 1995.

[Gib96a] Jeremy Gibbons. Computing Downwards Accumulations on Trees Quickly. *Theoretical Computer Science*, 169(1):67–80, 1996.

[Gib96b] Jeremy Gibbons. The Third Homomorphism Theorem. *Journal of Functional Programming (JFP)*, 6(4):657–665, 1996.

[Gib02a] Jeremy Gibbons. Calculating Functional Programs. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction: International Summer School and Workshop Oxford, UK, April 10–14, 2000 Revised Lectures*, pages 151–203. Springer, Berlin, Heidelberg, 2002.

[Gib02b] Jeremy Gibbons. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 151–203. Springer, 2002.

[GKCE17] Kyriakos Georgiou, Steve Kerrison, Zbigniew Chamski, and Kerstin Eder. Energy Transparency for Deeply Embedded Programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(1):8, 2017.

[GL95] Sergei Gorlatch and Christian Lengauer. Parallelization of Divide-and-Conquer in the Bird-Meertens Formalism. *Formal Aspects of Comp.*, 7(6):663–682, 1995.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, volume 1. MIT press, 1999.

[Gor99] Sergei Gorlatch. Extracting and Implementing List Homomorphisms in Parallel Program Development. *Sci. of Computer Programming*, 33(1):1 – 27, 1999.

[Gor04] Sergei Gorlatch. Send-receive Considered Harmful: Myths and Realities of Message Passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[GzVL10] Horacio González Vélez and Mario Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Par-

allel Programming Enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[HAB+11] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In *International Symposium on Formal Methods for Components and Objects*, pages 218–236. Springer, 2011.

[Ham99] Mohammad M Hamdan. A Survey of Cost Models for Algorithmic Skeletons. Technical report, Technical report, Heriot-Watt University, Dept. of Computers and Electrical Engineering, 1999.

[Ham07] G. W. Hamilton. Distillation: Extracting the Essence of Programs. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 61–70, New York, NY, USA, 2007. ACM.

[Har89] Thérèse Hardin. Confluence Results for the Pure Strong Categorical Logic CCL. λ-calculi as Subsystems of CCL. *Theoretical Computer Science*, 65(3):291–342, 1989.

[HBL03] Kevin Hammond, Jost Berthold, and Rita Loogen. Automatic Skeletons in Template Haskell. *Parallel processing letters*, 13(03):413–424, 2003.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

[HBS16] Kevin Hammond, Christopher Brown, and Susmit Sarkar. Timing Properties and Correctness for Structured Parallel Programs on x86-64 Multicores. In *Proc. FOPARA 2015:*

Foundational and Practical Aspects of Resource Analysis, Revised Selected Papers, Springer LNCS 9964, pages 101–125, 2016.

[HC02] Yasushi Hayashi and Murray Cole. Static Performance Prediction of Skeletal Parallel Programs. *Parallel Algorithms and Applications*, 17(1):59–84, 2002.

[Her00] Christoph Armin Herrmann. *The Skeleton Based Parallelization of Divide and Conquer Recursions*. Logos-Verlag, 2000.

[HH06] Catherine Hope and Graham Hutton. Compact Fusion. In *Workshop on Mathematically Structured Functional Programming*, Kuressaare, Estonia, July 2006.

[HH15] Jennifer Hackett and Graham Hutton. Programs for cheap! In *Thirtieth Annual ACM/IEEE Symposium on Logic in Computer Science*, 2015.

[Hin10] Ralf Hinze. Adjoint Folds and Unfolds. In Claude Bolduc, Jules Desharnais, and Béchir Ktari, editors, *Mathematics of Program Construction: 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings*, pages 195–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[HIT96] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving Structural Hylomorphisms from Recursive Definitions. In *Proc. ICFP '96: ACM Int. Conf. on Functional Programming*, ICFP '96, pages 73–82, New York, NY, USA, 1996. ACM.

[HIT97] Zhenjiang Hu, Hideya Iwasaki, and Masato Takechi. Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):444–461, 1997.

[HL00] Christoph A Herrmann and Christian Lengauer. A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(02n03):239–250, 2000.

[HM00] Kevin Hammond and Greg Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, UK, 2000.

[Hoa78] Charles Antony Richard Hoare. Communicating Sequential Processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. POPL '96: 23rd ACM Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[HTC98] Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. Parallelization in Calculational Forms. In *Proc. POPL '98: 25th ACM Symposium on Principles of Programming Languages*, pages 316–328, 1998.

[Hug82] R. J. M. Hughes. Super-combinators a new implementation method for applicative languages. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 1–10, New York, NY, USA, 1982. ACM.

[Hug00] John Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37(1):67 – 111, 2000.

[Hut16] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2016.

[HVK98] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *Proceedings of ESOP'98: European Symposium on Programming*, volume 1381, pages 273–284. Springer, 1998.

[HWG15] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Conjugate Hylomorphisms – Or: The Mother of All Structured Recursion Schemes. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 527–538, New York, NY, USA, 2015. ACM.

[JBM+16] Vladimir Janjic, Christopher Brown, K Mackenzie, Kevin Hammond, Marco Danelutto, Marco Aldinucci, and J Daniel Garcia. RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications. In *Proc. PDP 2016: Euromicro International Conf. on Parallel, Distributed, and Network-Based Processing*, pages 288–295. IEEE, 2016.

[JH10] Vladimir Janjic and Kevin Hammond. Granularity-Aware Work-Stealing for Computationally-Uniform Grids. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 123–134. IEEE, 2010.

[JL11a] Noman Javed and Frédéric Loulergue. A Formal Programming Model of Orléans Skeleton Library. In *International Conference on Parallel Computing Technologies*, pages 40–52. Springer, 2011.

[JL11b] Noman Javed and Frédéric Loulergue. Parallel Programming and Performance Predictability with Orléans Skeleton Library. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 257–263. IEEE, 2011.

[JL12] Noman Javed and Frédéric Loulergue. Verification of a heat diffusion simulation written with orléans skeleton library. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics: 9th International Conference, PPAM*

2011, Torun, Poland, September 11-14, 2011. Revised Selected Papers, Part II, pages 91–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Joh88] Eric E. Johnson. Completing an MIMD Multiprocessor Taxonomy. *SIGARCH Comput. Archit. News*, 16(3):44–47, June 1988.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[KC98] Gabriele Keller and ManuelM.T. Chakravarty. Flattening Trees. In *Proc. Euro-Par '98: European Conference on Parallelism*, pages 709–719. Springer, 1998.

[KE15] Steve Kerrison and Kerstin Eder. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(3):56, 2015.

[KH17] Venkatesh Kannan and G. W. Hamilton. "functional program transformation for parallelisation using skeletons". *International Journal of Parallel Programming*, May 2017.

[KI08] Yuki Karasawa and Hideya Iwasaki. Parallel Skeletons for Sparse Matrices in SkeTo Skeleton Library. *Information and Media Technologies*, 3(2):301–315, 2008.

[KRBJ12] Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic Parallelism via Liquid Effects. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 45–54. ACM, 2012.

[LCKR13] Ben Lippmeier, Manuel M.T. Chakravarty, Gabriele Keller, and Amos Robinson. Data Flow Fusion with Series Expressions in Haskell. In *Proc. 2013 Haskell Symposium*, pages 93–104. ACM, 2013.

[LGG13] Pedro Lopez-Garcia and Neville Grech. Energy Consumption Analysis of Programs Based on XMOS ISA-level Models. *Logic-Based Program Synthesis and Transformation*, page 72, 2013.

[LGK+15] Umer Liqat, Kyriakos Georgiou, Steve Kerrison, Pedro Lopez-Garcia, John P Gallagher, Manuel V Hermenegildo, and Kerstin Eder. Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR. In *International Workshop on Foundational and Practical Aspects of Resource Analysis*, pages 81–100. Springer, 2015.

[LH96] Hans-Wolfgang Loidl and Kevin Hammond. A Sized Time System for a Parallel Functional Language. In *Proc. Glasgow Workshop on Functional Programming, Ullapool, Scotland*, 1996.

[LHM11] Yu Liu, Zhenjiang Hu, and Kiminori Matsuzaki. Towards Systematic Parallel Programming over Mapreduce. In *Proc. Euro-Par 2011: European Conference on Parallelism*, pages 39–50. Springer, 2011.

[LL10] Oleg Lobachev and Rita Loogen. Estimating Parallel Performance, a Skeleton-Based Approach. In *Proc. HLPA '10: Intl workshop on High-level Parallel Prog. and Appls.*, pages 25–34, 2010.

[LMM+15] Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Protocol-based Verification of Message-passing Parallel Programs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 280–298, New York, NY, USA, 2015. ACM.

[LSB09] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the*

24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[LZC+02] Glenn R Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. Deadlock Detection in MPI Programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.

[Mat17] Kiminori Matsuzaki. Efficient Implementation of Tree Skeletons on Distributed-Memory Parallel Computers. *Scalable Computing: Practice and Experience*, 18(1):17–34, 2017.

[MBCP14] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access. In *Proc. of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 325–337, New York, NY, USA, 2014. ACM.

[Mee86] Lambert Meertens. Algorithmics - Towards Programming as a Mathematical Activity. In *Proc. of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[Mee96] Lambert Meertens. Calculate polytypically! In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, pages 1–16, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proc. FPCA '91: ACM Conf. on Functional Programming and Computer Architecture*, pages 124–144, 1991.

[MHT06a] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallel Skeletons for manipulating General Trees. *Parallel Computing*, 32(7):590–603, 2006.

[MHT06b] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Towards Automatic Parallelization of Tree Reductions in Dynamic Programming. In *Proc. SPAA 2006: Symp. on Parallelism in Algorithms and Architecture*, pages 39–48, 2006.

[MIEH06] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *Proceedings of the 1st international conference on Scalable information systems*, page 13. ACM, 2006.

[MIK97] Greg Michaelson, Andrew Ireland, and Peter King. Towards a Skeleton Based Parallelising Compiler for SML. In *Proceedings of 9th International Workshop on Implementation of Functional Languages*, pages 539–546, 1997.

[Mil97] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi Calculus*. Cambridge university press, 1999.

[Mis94] Jayadev Misra. Powerlist: A Structure for Parallel Recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1737–1767, 1994.

[Mit96] John C Mitchell. *Foundations for Programming Languages*, volume 1. MIT press Cambridge, 1996.

[MM10] Akimasa Morihata and Kiminori Matsuzaki. Automatic Parallelization of Recursive Functions using Quantifier Elimination. In *Proc. FLOPS '10: Functional and Logic Programming*, pages 321–336. Springer, 2010.

[MML+10] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq No More: Better Strategies for Parallel Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 91–102, New York, NY, USA, 2010. ACM.

[MMM+07] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic Inversion Generates Divide-and-Conquer Parallel Programs. In *Proc. PLDI '07: ACM Conf. on Prog. Lang. Design and Impl.*, pages 146–155, 2007.

[MMT15] Patrick Maier, John Magnus Morton, and Phil Trinder. Towards an Adaptive Skeleton Framework for Performance Portability. unpublished, 2015.

[MMT16] Patrick Maier, John Magnus Morton, and Phil Trinder. Towards an Adaptive Skeleton Framework for Performance Portability. unpublished, 2016.

[MNPJ11] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM.

[Mor13] Akimasa Morihata. A Short Cut to Parallelization Theorems. In *Proc. ICFP 2013: 18th ACM Conf. on Functional Programming*, pages 245–256, 2013.

[MPJS09] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[NAA+95] Rishiyur S Nikhil, J Hicks Arvind, Shail Aditya, Lennart Augustsson, J Maessen, and Yuli Zhou. pH Language Reference Manual, 1995.

[NCT99] Michael K Ng, Raymond H Chan, and Wun-Cheung Tang. A Fast Algorithm for deblurring Models with Neumann Boundary Conditions. *SIAM Journal on Scientific Computing*, 21(3):851–866, 1999.

[NPP04] James G Nagy, Katrina Palmer, and Lisa Perrone. Iterative Methods for Image Deblurring: a Matlab Object-Oriented Approach. *Numerical Algorithms*, 36(1):73–93, 2004.

[NU09] Michael J Neely and Rahul Urgaonkar. Optimal Backpressure Routing for Wireless Networks with Multi-Receiver Diversity. *Ad Hoc Networks*, 7(5):862–881, 2009.

[NY16] Nicholas Ng and Nobuko Yoshida. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 174–184. ACM, 2016.

[Pel98] Susanna Pelagatti. *Structured Development of Parallel Programs*, volume 102. Taylor & Francis, Abington, 1998.

[Pie02] Benjamin C Pierce. *Types and programming languages*. The MIT Press, 2002.

[PnS01] Ricardo Peña and Clara Segura. Sized Types for Typing Eden Skeletons. In *Proceedings of IFL '01: Intl. Symp. on Implementation of Functional Languages*, pages 1–17, 2001.

[PnS05] Ricardo Peña and Clara Marıa Segura. Reasoning about Skeletons in Eden. In *Parallel Computing: Current & Future Issues of High-End Computing*, 2005.

[Rei93] John H Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[RG03] Fethi A Rabhi and Sergei Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[RHJ09] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[San95] David Sands. A Naïve Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.

[SB17] Franck Slama and Edwin Brady. Automatically proving equivalence by type-safe reflection. In *International Conference on Intelligent Computer Mathematics*, pages 40–55. Springer, 2017.

[SBHG08] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 253–264, New York, NY, USA, 2008. ACM.

[SC95] David B Skillicorn and Wentong Cai. A Cost Calculus for Parallel Functional Programming. *J. Parallel Distrib. Comput.*, 28(1):65–83, 1995.

[Sch24] M. Schönfinkel. Über die Bausteine der Mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

[SFLD15] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating Performance Portable Code Using Rewrite Rules. In *Proc ICFP 2015: 20th ACM Conf. on Functional Prog. Lang. and Comp. Arch.*, pages 205–217, 2015.

[SG02] Jocelyn Sérot and Dominique Ginhac. Skeletons for Parallel Image Processing: an Overview of the Skipper Project. *Parallel computing*, 28(12):1685–1708, 2002.

[SGD99] Jocelyn Sérot, Dominique Ginhac, and Jean-Pierre Dérutin. Skipper: a Skeleton-Based Parallel Programming Environment for Real-time Image Processing Applications. In *International Conference on Parallel Computing Technologies*, pages 296–305. Springer, 1999.

[SHMB05] Norman Scaife, Susumi Horiguchi, Greg Michaelson, and Paul Bristow. A parallel SML compiler based on algorithmic skeletons. *Journal of Functional Programming*, 15(4):615–650, 2005.

[Ski91] David B Skillicorn. Models for Practical Parallel Computation. *International Journal of Parallel Programming*, 20(2):133–158, 1991.

[Ski93a] David B Skillicorn. Deriving Parallel Programs from Specifications using Cost Information. *Science of Computer Programming*, 20(3):205–221, 1993.

[Ski93b] David B Skillicorn. *The Bird-Meertens Formalism as a Parallel Model*. Springer, 1993.

[SL08] Michael Süß and Claudia Leopold. Common mistakes in OpenMP and how to avoid them. In *OpenMP Shared Memory Parallel Programming*, pages 312–323. Springer, 2008.

[Sor94] Yves Sorel. Massively parallel computing systems with real time constraints: the" algorithm architecture adequation" methodology. In *Massively Parallel Computing Systems, 1994., Proceedings of the First International Conference on*, pages 44–53. IEEE, 1994.

[SRD17] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 74–85. IEEE Press, 2017.

[Sto77] Joseph E Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. MIT press, 1977.

[Tea] The Go Team. Effective Go. `https://golang.org/doc/effective_go.html`. Accessed: 2017-06-16.

[THLJ98] Philip W. Trinder, Kevin Hammond, H-W Loidl, and SL Peyton Jones. Algorithm+Strategy=Parallelism. *Journal of Functional Programming*, 8(01):23–60, 1998.

[TI09] Haruto Tanno and Hideya Iwasaki. Parallel Skeletons for Variable-Length Lists in SkeTo Skeleton Library. In *European Conference on Parallel Processing*, pages 666–677. Springer, 2009.

[Tra88] Kenneth R Traub. Sequential implementation of lenient programming languages. Technical report, MIT Lab for Computer Science, 1988.

[Tur79] D. A. Turner. Another Algorithm for Bracket Abstraction. *Journal of Symbolic Logic*, 44(2):267–270, 1979.

[Val89] Leslie G Valiant. *Bulk-Synchronous Parallel Computers*. Harvard University, Center for Research in Computing Technology, Aiken Computation Laboratory, 1989.

[Val90] Leslie G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Vas08] Pedro B Vasconcelos. *Space Cost Analysis using Sized Types*. PhD thesis, University of St Andrews, 2008.

[XKCH03] N Xu, Siau-Cheng Khoo, Wei-Ngan Chin, and Zhenjiang Hu. A Type-Based Approach to Parallelization. In *National University of Singapore*, 2003.

[Yok89] Hirofumi Yokouchi. Church-Rosser Theorem for a Rewriting System on Categorical Combinators. *Theoretical Computer Science*, 65(3):271–290, 1989.