Institut für Informatik

Lehrstuhl für Sprachen und Beschreibungsstrukturen

Programming Language Design, Analysis and Implementation for Automated and Effective Program Parallelization

Dipl-Phys. Univ. Alexander Herz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Hans Michael Gerndt
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Helmut Seidl
	2. Univ.-Prof. Dr. Sebastian Hack, Universität des Saarlandes Saarbrücken

Die Dissertation wurde am 07.04.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 02.07.2015 angenommen.

Abstract

In this thesis, we define the general purpose, automatically parallel programming language **funkyImp**. The design of the language semantics is tailored specifically towards the goal that the analysis required to prove independence of program fragments should be fast and precise while the performance of the sequentially executed program parts should be on a par with C++. This is achieved by restricting aliasing and side-effects in such a way that the analysis becomes intra-procedural and allows to identify all static task-parallelism precisely on the function level.

In addition, we introduce sub-arrays, e.g. the diagonal or upper triangle of a matrix, as language primitives using a novel type system based on array index sets defined via linear constraints. The composition of data-parallel computations on sub-arrays in conjunction with the restriction of side-effects ensures that optimizations based on the polytope model and automatic parallelization using threads and SIMD can be applied even in complex and deeply nested settings where current compilers fail.

Furthermore, we present a static scheduling algorithm that realistically models thread coordination overhead. It enables optimization of automatically extracted task-parallelism for specific target systems and guarantees a real world speedup.

Finally, we present the implementation of the **funkyImp** compiler and run-time system along with benchmarks to show the effectiveness of our approach.

The second part of this thesis is devoted to simplifying manual parallelization of legacy applications by using points-to analyses to predict parallelization hazards before manual parallelization is performed. Given a sequential application and a specification describing the intended parallelization, the analysis finds unprotected shared resources. We present a method that uses such specifications in order reduce the amount of code that must be analyzed by specializing existing whole-program analyses. We successfully applied the method on a large, industrial C++ code base.

Zusammenfassung

In dieser Arbeit wird die universelle, automatisch parallele Programmiersprache **funkyImp** entwickelt. Die Semantik der Sprache wurde speziell so konstruiert, dass eine schnelle und präzise Analyse zum Auffinden unabhängiger Programmteile benutzt werden kann, während die Ausführungsgeschwindigkeit der sequentiellen Programmteile vergleichbar mit C++ ist. Dies wird erreicht, indem Aliasing und Seiteneffekte auf eine solche Art und Weise eingeschränkt werden, dass die benötigte Analyse präzise und intraprozedural allen statischen Task-Parallelismus in einer Funktion ermitteln kann.

Zusätzlich werden Teilarrays eingeführt, wie z.B. die Diagonale oder das obere Dreieck einer Matrix, als Sprachprimitive zum Ausdruck von Daten-Parallelismus. Diese Teilarrays basieren auf einem Typsystem, welches Mengen und Teilmengen von Arrayindizes über lineare Begrenzungen definiert. Die Möglichkeit datenparallele Berechnungen über Teilarrays auszudrücken in Kombination mit den eingeschränkten Seiteneffekten, erlaubt es Optimierungen, die auf dem Polytopmodell und Threads so wie SIMD basieren, auch in komplexen und tief verschachtelten Szenarien anzuwenden, in denen aktuelle Compiler scheitern.

Darüber hinaus wird ein statischer Ausführungsplanungsalgorithmus entwickelt, welcher die durch Thread-Koordination entstehenden Ausführungszusatz-kosten realistisch modelliert. Dieser Algorithmus ermöglicht es automatisch extrahierten Task-Parallelismus für spezifische Zielsysteme zu optimieren.

Abschließend wird die Implementierung des **funkyImp** Compilers und Laufzeitsystems vorgestellt zusammen mit Experimenten, welche die Effektivität des Ansatzes illustrieren.

Der zweite Teil dieser Arbeit widmet sich der manuellen Parallelisierung von Altprogrammen und nutzt Zeigeranalysen um Parallelisierungsprobleme vorherzusagen. Mittels einer Spezifikation, welche eine geplante Parallelisierung beschreibt, können solche Analysen ungeschützte, geteilte Ressourcen finden. Solche Spezifikationen können verwendet werden, um existierende Analysen, welche das komplette Programm als Eingabe benötigen, so zu spezialisieren, dass die Menge des zur Analyse benötigten Codes stark eingeschränkt wird.

Acknowledgements

Foremost, I would like to thank my family for making this work possible. My parents, for being the nice people that they are and for providing me with an environment where I could successfully develop my interests and follow my ambitions. My wife Daniela and my son Samuel for supporting me with love and happiness through the highs and lows of creating a PhD thesis and life in general.

Furthermore, I would like to thank my adviser, Helmut Seidl, for giving me the opportunity to follow my research goals to the very end, even if the success of the methods was initially not clearly visible.

I'm also grateful to the many students who have contributed to my work via their bachelor's or master's thesis or as research assistants, especially Armin Gensler, Andreas Wagner, Alexander Pöppl and Dominic Giebert should be mentioned.

Of course I want to thank all friends and colleagues who have made my stay at this chair pleasant and interesting. I would like to thank Andreas, Raphaela, Stefan, Ralf and especially Julian for going the extra mile and supporting this thesis with corrections and discussions.

I would also like to thank Axel, Kalmar, Martin and Chris for being my paper buddies and sustaining the rush of my drafts and ideas to the point where submission was inevitable.

Finally, I would like to thank Martin, Sylvia, Holger and Aleks for every 'Punktenspiel' I have won against them and Bogdan as well as Michael and Vesal for their emotional involvement in every kicker match they have played using the hinterfuss, an around the world or the natural tox.

Contents

Chapter 1

Introduction

In this chapter we state the problem that this thesis is devoted to before discussing existing methods to attack the problem and we present the contribution of our work.

This chapter is structured as follows. First, we discuss the problems inherent to parallel programming in Sec. 1.1. Then, in Sec. 1.2, we outline the specific problem this thesis is devoted to, which is fully automatic transformation of sequential code into parallel code. Afterwards, in Sec. 1.3, we discuss existing methods to simplify program parallelization before we evaluate existing auto-parallelizing compilers and the benefits of restricted side-effects for automatic parallelization in Sec. 1.4. This includes Sec. 1.4.4 and Sec. 1.4.5, where data-parallelism and non-determinism in the context of programming languages with restricted side-effects are discussed. Finally, in Sec. 1.4.6, we discuss static optimization of automatically extracted parallelism before we introduce our automatically parallel language **funkyImp** in Sec. 1.5.

1.1 Problem Discussion

Software developers have enjoyed a 'free lunch' [133] for several decades. Hardware designers had managed to roughly double the computational power of the single processors becoming available every year during that time. As a consequence, more computations per time could be performed every year, even with old software simply by running it on a more powerful processor. Exponential growth processes in a finite physical system cannot continue forever. Around the year 2004 hardware designers hit a physical limit which enforced an upper limit on processor speeds which Peter Kogge [78] refers to as the 'perfect storm'. Since processor speeds do not increase much further, most hardware platforms released since 2004 are equipped with an increasing number of parallel processors while the speed of individual processors remains almost constant over the years.

The majority of software produced before 2004 uses only a single processor, except for large scale numerical simulations and similar niche products. In order to access the additional parallel processing power, techniques which explicitly distribute software components across several processors and explicitly manage communication between these components must be employed. In the following we will refer to these techniques as *parallel programming*.

Software architects and developers today face the challenge to cope with the additional complexity and their lack of experience in parallel programming.

There exists a consent that parallel programming is much harder than sequential programming [20, 101, 104, 133] although, to the best of our knowledge, little has been

written about the root causes of the additional complexity.

The increased complexity can be investigated from five different perspectives.

Resource Management In a sequential program that is optimized for computational throughput the programmer usually assumes that he has unrestricted and uncontested access to all computing resources (processor, memory, I/O bandwidth). Except for the program logic itself, only memory might need to be managed explicitly in the absence of a garbage collector. In a sense, resource management in sequential programming is a local problem because, except for the previously allocated memory, the computing resources are used exclusively by the currently executing instruction. So that in principle, neglecting cache effects etc., each section of code can be inspected and reasoned about independently of the rest of the program.

In parallel programs, different parts of a program contest to use the same set of shared computing resources simultaneously. Now resource management has become a global problem because the resource usage and explicit resource management of all program parts, that may execute in parallel, may need to be taken into account when modifying a single line of code.

Semantics The semantics of a well defined program fragment from a sequential program without I/O depends only on the program state which holds before the program fragment is executed. This allows the programmer to infer program state invariants that must hold before a program fragment is executed and then use these invariants to safely reason about the semantics of the following program fragment.

In contrast, most parallel programs are non-deterministic so that reasoning about a single global program state and the related invariants becomes practically impossible. In languages like C++, the programmer must explicitly protect the memory reads and writes in a given program fragment against modification by any program fragment that may execute concurrently. In the current C++11 standard, a *data-race* is defined as a program fragment that reads from a memory location which might be concurrently written. The semantics of code that contains a data-race is undefined [52].

Data-races are hard to find by testing because code containing data-races only rarely exhibits unexpected behavior. This is the case because the visible semantics of the code in question depends on whether the specific processor instructions producing the data-race actually execute concurrently or not. The relative timing of execution of two instructions from two concurrently executing program fragments is undefined in the absence of synchronization instructions. So any interleaving of the instructions from both fragments may be observed in a program run. Usually, only few pairs of instructions are in conflict so that only a few of the many possible interleavings actually expose the undefined semantics. Assuming a roughly equal probability for each interleaving to occur in a single run, actually observing a data-race becomes highly unlikely if the ratio of safe to unsafe instructions is high. In Chap. 6 we discuss protection against data-races in more detail.

Many commonly used languages, like C++, give the programmer little help in preventing data-races. It is alone the programmer's responsibility to find data-races and protect against them. To do so, the programmer must infer if any aliases to any memory his code may access exist, which may be accessed concurrently. This

requires non-local inter-procedural reasoning [63]. As data-races only occur rarely and fail silently they often remain undetected.

Debugging In the absence of non-deterministic operations, sequential programs are easy to debug because error conditions can be reliably reproduced and inspected. Step-by-step execution of the program has a clear semantics (what is executed within a single step) and helps to understand the code in focus of the debugger.

Even in the absence of data-races, debugging parallel applications is hindered by the inherent non-determinism of parallel execution. The occurrence of errors might critically depend on the relative timing of specific program events which cannot be consistently reproduced. In addition to executing the code line in focus, step-by-step debugging of a parallel program executes an unpredictable amount of code that is running concurrently. The programmer must distinguish which side-effects stem from the code line in focus and which side-effects must be attributed to other code without having any control and little information about which other code did actually execute concurrently.

Optimization Goals We assume that the goal of parallelizing a program is to improve the computational throughput in the presence of multiple processors. An obvious optimization goal is that the parallel program should have more computational throughput than a sequential version of the same program. In the sequential version all tasks are executed in a predefined order on a single processor and all synchronization is removed since it is not required anymore but would cause overhead. It turns out that the parallel version of a program may very well execute slower than its sequential version. The actual execution time of a parallel program depends on the hard- and software environment it is executed in, as well as the specific distribution of the program onto the available processors. From now on we will refer to the process of distributing a program onto multiple processors as *parallel decomposition*. Program fragments that may execute concurrently are referred to as *tasks*. A detailed discussion of parallel decomposition is presented in Chap. 2.

In general, finding the best mapping of a parallel decomposition of a program onto a hardware with N identical processors is NP-complete [46]. This shows that trying to find the best solution by starting with a best guess and refining the guess with a greedy strategy need not succeed. In other words, given a parallel program that performs bad on a specific target hardware, there is no simple heuristic that guarantees to improve the solution. Instead, a set of conflicting goals must be solved simultaneously:

On the one hand, we want to expose parallelism to benefit from parallel processors. On the other hand, increasing the parallelism of the program reduces the amount of computation performed per task. It turns out that the overall execution time of a parallel program critically depends on the amount of computation performed per task, as discussed in Chap. 3. If the amount of computations per task is low then the parallel program will exhibit a lot of parallelism but show little speedup or even slowdown compared to the sequential version, because the overhead generated by thread coordination outweighs the benefits of parallel execution. If the ratio of computation to overhead is high, better speedups can be expected but less parallelism can be exploited. For optimal results, a fine balance between the amount of exposed parallelism and the amount of computation per task must be struck. Even

worse, the optimal solution depends on the specific target hard- and software so that the complex and error prone process of finding and implementing a good parallel decomposition must be performed for each combination of target hard- and software that is not largely identical to those target systems for which a good solution was already found. In Chap. 3 we discuss this problem in detail.

Maintainability As the 'free lunch' is over, even standard software (like word processing, browsing, etc.) must employ parallelism to benefit from future hardware development. Usually, this kind of software has a long lifetime and the majority of development effort (somewhere between 40% and 80%, on average 60% [50]) is spend on maintaining software whereas the remaining time is spend developing the software in the first place. This introduces another conflicting goal. As the previous discussion indicates, the complexity of developing and maintaining software may grow dramatically with the amount of parallelism introduced. Therefore, introducing parallelism into an application may have a large impact on the overall development costs.

1.2 Thesis Scope

Obviously, our discussion of what makes parallel programming difficult is not holistic. Depending on the specific target system and specific problem that should be parallelized, some of the problems we have mentioned may become trivial and new problems may arise. Usually, these new problems will be related to additional constraints placed on the parallel decomposition by the hardware. Since we are interested in generic concepts that simplify the development of parallel software, we try to abstract the discussion as far as possible from specific hardware constraints unless these constraints are common among today's hardware. On the other hand, all common hardware constraints that have a reasonable impact on the performance of parallel programs must be taken into account to avoid a purely theoretical contribution.

Our discussion of parallel programming so far restricted itself on the problems that arise when extracting parallelism from an initially sequential implementation. Obviously, the amount of effectively exploitable parallelism depends critically on the algorithms and techniques used in the sequential implementation. Finding and employing algorithms and techniques which are amenable to parallelization is an additional burden placed on the programmer. There exists an increasing range of literature dealing with parallel algorithms [84] and their discussion is beyond the scope of this thesis.

The main motivation of this thesis is to find methods that simplify the process of turning a given non-parallel implementation into a semantically equivalent parallel program that reaches near optimal speedups (for the given, initial algorithm) on a specific target system. Specifically, we discuss how this process can be fully automated so that parallelization errors like data-races and dead-locks are excluded by construction. Furthermore, we show that our automated approach is viable because the quality (in terms of real-world performance) of our automated parallelization is close to manually parallelized C++ code.

The remainder of this chapter is devoted to discussing existing methods aiding in the prallelization of programs and how the contribution of this thesis is related to these methods.

1.3 Existing Methods

In this section we will discuss existing methods that try to aid the programmer in producing correct and efficient parallel programs, thus alleviating some of the problems associated with manual parallelization of programs.

The existing methods can be roughly grouped into three categories.

Syntax Sugar With syntax sugar we refer to language syntax, introduced into many mainstream languages like C++, that abbreviates the code necessary to write parallel programs and makes the code more portable by abstracting from the native libraries like POSIX threads [10]. In this sense Cilk [71] introduces keywords that allow to fork and join parallel computations, OpenMP [111] uses pragmas to parallize loops and tasks while MPI [108] allows portable message passing and synchronization across a network of computers. All instances of syntax sugar have in common that they merely shorten the amount of code that needs to be written. In particular, they do neither automate any of the key steps required to parallelize a program nor safeguard against parallel programming errors.

Analysis Tools A wide range of modelling techniques to over- or under-approximate the semantics of parallel programs have been developed , for example abstract interpretation [33], model checking [30], trace checking [148], etc. A thorough discussion is beyond the scope of this thesis. These techniques can be used to find errors in an already parallelized program, e.g. insufficient locking of shared memory [140]. As discussed in the previous section, correctness does not guarantee speedup. Thus, some of the problems of parallelizing programs are unaffected by these tools.

Auto-Parallelizing Compilers Auto-parallelizing compilers take as input a sequential or declarative program and automatically produce a semantically equivalent, parallel program and thus eliminate the complexity and correctness problems associated with parallel programming. Generally, this would be the preferred solution by the software industry. Programmers keep writing sequential software in the languages they already know and profit automatically and with little extra effort from parallel hardware. Many auto-parallelizing compilers for existing languages and domain specific languages have been created [12, 35, 48, 58, 75, 96, 106].

1.4 Auto-Parallelizing Compilers

1.4.1 Auto-Parallelizing Compilers for Side-Effecting Languages

Even ten years after the rise of multi processor hardware, the majority of parallel software is still created using manual, imperative parallel programming. With imperative programming we refer to languages where the program is defined via commands that mutate the state of variables using side-effects, e.g. C++, Java, etc. Many auto-parallelizing compiler techniques have been integrated into compilers for imperative languages, like g++ [106], which can parallelize simple loop nests. Pluto [12] applies even more advanced polyhedral techniques to extract *data*-parallelism from a given loop nest. Data-parallelism denotes the parallelism available when applying the same computations on different data.

Fortran, OSCAR and the work from Polychronopoulos on C [48, 75] have led to compilers that extract higher level, *task*-parallelism from programs. In contrast to data-

parallelism, task-parallelism denotes the parallelism available by performing different, independent computations on potentially the same data in parallel. These methods require that the compiler finds provably independent program fragments that can be parallelized without altering the program semantics and which are large enough to yield an actual speedup when executed in parallel.

The performance results obtained with these auto-parallelizing compilers suffer from the problem that common existing programming languages were not designed with auto-parallelization in mind. As a consequence aliasing and mutation of shared state is a common usage scenario in imperative languages like Fortran and especially C/C++ [104]. In general, an expensive inter-procedural points-to analysis [63] is required to prove that program fragments are indeed independent. In Chap. 9 we present such a points-to analysis and discuss more details. The over-approximative nature of these analyses may and in practice will flag some program fragments as dependent although they are not. Thus, the amount of parallelism that can be extracted from an imperative program is reduced.

Summarizing, the existing auto-parallelizing compilers for imperative languages deliver impressive results [12, 75] for specific problem formulations. However, the imprecise extraction of parallel tasks limits the applicability of these compilers. A more precise and efficient extraction of parallelism is hindered by the unconstrained use of mutable shared state in the underlying programming language. Therefore, these tools have found limited application in real-world software development. Type systems that constrain the use of aliasing to simplify automatic parallelization are actively being researched [129].

1.4.2 Auto-Parallelizing Compilers for Side-Effect Free Languages

In contrast to imperative languages, side-effect free languages are amenable to automatic parallelization. Due to the lack of mutable state, independence of computations can be easily proven. The following code example illustrates this.

```
c=g(a); //may write to a
d=h(b); //may write to b
i(c,d);
```

(a) C/C++

```
let c=g(a) in (*a immutable*)
let d=h(b) in (*b immutable*)
i(c,d)
```

(b) Haskell

Figure 1.1: Comparing a side-effecting language like C++ in (a) to a side-effect free language like Haskell in (b). g, h, i are functions and a, b, c, d are mutable variables in (a), immutable values in (b) respectively.

In a side-effecting language (example code in Fig. 1.1 (a)), the arguments to functions h and g may contain pointers to the same memory and g (or a function transitively invoked by g) might write to this memory while h may read from the same memory. In this way an implicit data dependency from g to h may be constructed inside these functions. These kind of dependencies must be resolved to prove independence of program fragments like $g(a)$ and $h(b)$, so that they can be correctly parallelized. Without side-effects (example code in Fig. 1.1 (b)), all values passed into g and h are read-only. So even if the arguments contain pointers to the same memory, no implicit dependency can be constructed by writing to the aliased memory. In this setting, only values which are defined by one program fragment and which are explicitly referenced by name in another fragment create data dependencies. For example, the expression $i(c, d)$ explicitly depends

on *let* $c = g(a)$ and *let* $d = h(b)$. This kind of explicit data flow [43] can be analyzed intra-procedurally and precisely. In Chap. 2, we discuss how the *static* task-parallelism of a side-effect free functional language can be extracted precisely. With *static* parallelism we denote the set of computations in a program that is always independent, irrespective of the dynamic program values like user input. Note that automatic parallelization of a side-effect free language cannot produce data-races because all data is read-only after its initial construction so that concurrent writes to data cannot exist.

1.4.3 Side-Effects in Functional Languages

As discussed in the previous sections, many methods exist which enable either good parallel and sequential performance (e.g. parallel C++) or safe and simple parallelization (e.g. OSCAR, Pluto, Haskell). Sadly, these two features are not orthogonal and cannot simply be combined by composition. On the one hand, to obtain safety, we want to disallow all dangerous program constructs like side-effects. On the other hand, to obtain good performance, some dangerous program constructs may be required. To solve this dilemma, mutable state must be carefully reintroduced into a side-effect free language to retain both the parallel performance and the safety of the side-effect free language.

In existing functional languages there exist several mechanisms to reintroduce side-effects. We will discuss the most common mechanisms in the following. Depending on the specific mechanism some or all of the problems associated with auto-parallelization in side-effecting languages may be reintroduced into the language.

Ocaml [120] adds unrestricted references to mutable memory, effectively reintroducing all problems from side-effecting languages into all parts of the program.

Haskell [72] uses monads [142] to encapsulate side-effects. As the side-effects are strictly separated from the rest of the program, at least parts of Haskell programs are side-effect free and may benefit from auto-parallelization. The program fragments inside the I/O monad can be interpreted as a side-effecting sub-programs and thus have the same parallelization problems as side-effecting languages. Furthermore, Haskell uses demand-driven, lazy evaluation [83], so only expressions that are needed are actually computed. This allows to express, for example, infinite lists and is optimal for sequential programs because the minimum amount of computation required to produce the result is performed. As a consequence, only expressions that will be used can be safely evaluated, other expressions may diverge. For example, an infinite lists cannot be fully evaluated in finite time and space. It has been recognized that laziness inhibits certain forms of parallelism because some computations may be deferred unnecessarily [57, 70]. Laziness is contrasted with strict evaluation [5], where all expressions are evaluated as soon as possible, exposing the maximum amount of parallelism. Languages and execution environments that try to find a middle ground between lazy and strict evaluation have been investigated [136]. Furthermore, strictness analysis [56] can be used to find instances where lazy evaluation can be safely turned into strict evaluation. Just like points-to analysis, strictness analysis is in general inter-procedural and approximative.

Eventually, the trade-off between strictness and laziness determines the size or granularity of the tasks that can be extracted from the side-effect free program part. As we will show in Chap. 3, the best granularity critically depends on the target hard- and software so that fixing the granularity at the language level may lead to non-optimal results. Instead, Haskell allows the programmer to control the amount of exposed parallelism explicitly using *par*, *seq* and $force$. In order to choose suitable tasks, the intricacies of lazy evaluation must be well understood and finely controlled. To simplify this process,

parallelization strategies [98] provide composeable parallel algorithm skeletons.

A performance comparison of parallel functional languages was performed by Loidl et al. [95]. In case performance comparisons to imperative parallel languages (usually C) are presented, the imperative languages usually outperform the functional parallel languages by several hundred percent. Sometimes, even the non-parallel, imperative programs outperform the parallel functional programs [95]. This indicates that, in general, the programmer controlled, partially lazy evaluation in combination with parallel graph rewriting [115] used to execute these programs cannot compete with the performance of imperative program execution and manual program parallelization.

We will conclude our discussion of task-parallel functional languages with Clean [116] as its use of uniqueness types [141] is especially relevant for Chap. 5 where we discuss non-shared mutable state in the context of our own parallel programming language **funky-Imp**. Uniqueness types allow to embed mutation of non-aliased references into a language while keeping the language side-effect free [134]. Similar to Haskell's monads, uniqueness types can express I/O operations but do not reintroduce shared mutability problems. In addition, we will show how uniqueness types can be used to improve the performance of sequential, side-effect free programs.

1.4.4 Data-Parallelism in Functional Languages

Today's processors and graphics cards deploy vector processing capabilities which allow to perform data-parallel operations [54, 69, 88].

In side-effecting languages, nested loops represent an awkward way to specify data-parallel computations on sub arrays, simply by iterating over the index subset that is of interest. Since the semantics of these loops is inherently sequential, compiler optimizations, usually based on the polyhedral model [12], are applied to reorder loop iterations in order to execute several independent iterations in parallel using data- and task-parallelism. These optimizations are easily broken by non-linear computations or imprecise pointer aliasing and alignment information.

Furthermore, loops are not readily expressed in side-effect free languages because they require side-effects on the loop-counter. Instead, several domain specific languages to express data-parallelism in functional languages have been proposed [25, 76, 124]. The problem here is to specify potentially nested map and reduce operations on whole arrays and especially on sub arrays. Higher level map and reduce operations are more readily parallelized because they have no implicit sequential semantics. Nevertheless, existing map and reduce operations can be applied only on whole arrays or on hyper cubic sub arrays. Furthermore, sub arrays cannot be efficiently passed into functions, practically inhibiting composition of array operations. Therefore, not all data-parallel algorithms a C++ programmer may specify using nested loops and explicit parallelization can be expressed efficiently using existing map and reduce based functional data-parallel languages.

In Chap. 4 we discuss data-parallel language constructs which, unlike previous languages, allow compositional specifications of arrays by adding sub arrays as primitive types. This allows the programmer to retain some control over the parallel decomposition, enables the compiler to prove the safety of array operations and guarantees that polyhedral model based optimizations can be applied. Thus, safe, efficient and generic data-parallel operations in side-effect free languages matching the performance of parallel C++ can be expressed.

1.4.5 Non-determinism and Communication in Functional Languages

So far we have covered methods to safely express static and deterministic parallelism in functional languages. Communication between tasks, e.g. the asynchronous response to events or the asynchronous communication needed to implement non-deterministic parallel algorithms, requires mutable state that is exchanged between tasks. In order to communicate in a shared memory environment, at least one task must be able to modify some state which can be read by at least one other task. In general, this communication is non-deterministic if several producers or consumers of messages use the same communication channel because the order of messages received by a consumer cannot be predicted. In the context of side-effecting languages, aliases to shared memory can be distributed to several tasks and then concurrent read and write operations on that memory can be performed to communicate. In order to protect this process from data-races, the programmer must explicitly guard the concurrent memory accesses, e.g. using locks [41] or other suitable protection schemes.

In Chap. 6 we will discuss how an extension of actors [31] can be used to introduce side-effects on shared mutable state into our language without breaking the analyses required for automatic parallelization. Using actors for communication between tasks guarantees that the resulting program is dead-lock and data-race free. Furthermore, actors allow to model locked and lock-free parallel data structures and introduce non-determinism into our language. Although data-race and dead-lock freedom is guaranteed, reasoning about and debugging non-deterministic programs, irrespective of parallelism, is non-trivial but non-determinism may expose *dynamic* parallelism and, in specific cases, may improve the performance of parallel programs [4]. Here, dynamic parallelism refers to programs where the execution order of tasks is not statically known and depends on the ordering of messages communicated between different tasks, which is generally not deterministic. In addition, actors extend the expressible data structures of a side-effect free language from trees to graphs [110], by introducing a controlled form of side-effects. The clear separation of deterministic program parts from non-deterministic parts allows the programmer to consciously trade complexity for performance where necessary.

1.4.6 Static Scheduling of Automatically Extracted Parallelism

After extracting task-parallelism from a program, the resulting tasks must be optimized for execution on a specific target hardware to guarantee a decent speedup. Automatic analysis of independent tasks will extract all program fragments that can be proven independent, irrespective of the task size. Most programs are composed of many small, basic operations that expose a large amount of tiny parallelism. For example, all non-constant summands in a sum, that have no data flow in between them, may be computed in parallel. Therefore, a large amount of small tasks will be extracted for most programs. We will show in Chap. 3 that, due to the overhead of thread coordination, executing small tasks on today's hardware will yield little speedup or even slowdown compared to the non-parallel program. Current static scheduling algorithms fail to take this overhead into account. The theoretical execution times these algorithms try to minimize become uncorrelated to the real execution time of the program, in case the program contains many small tasks.

In Chap. 3, we will present a model that predicts a target system specific thread coordination overhead in combination with a novel static scheduling algorithm that takes this overhead into account in order to guarantee a minimum real-world speedup.

1.5 Contribution

The first part of this thesis presents the design and implementation of the **funkyImp** language. It is based on a side-effect free core language. In the previous sections, we have given some intuition into how static task-parallelism can be precisely extracted from this core language and how this language can be extended by data-parallelism, unshared and shared mutable state while retaining the property that task-parallelism can be extracted easily and precisely. The semantics of each of these additional features must be carefully crafted, so that no precision is lost when the data flow analysis, which extracts the static task-parallelism, is lifted from the core language to the language including the additional features. As a result, the **funkyImp** language allows to implement efficient programs using side-effects and non-determinism where necessary, without inhibiting precise, automatic parallel decomposition. In combination with our static scheduling algorithm, that takes into account the specific hardware and software characteristics of the target system, highly efficient parallel programs can be generated. In Chap. 7 we present the **funkyImp** compiler and run-time system implementation, which is used to evaluate the performance characteristics of the different language components in comparison to common existing languages and libraries.

1.6 Overview

Part I of this thesis describes how the dead-lock free and data-race free, auto-parallelizing language **funkyImp** was designed and implemented.

In Chap. 2, we define a simple first order core language and present the necessary data flow analyses to precisely extract static task-parallelism from it. Afterwards, in Chap. 3, we present our task graph preconditioning algorithm which is used to optimize task graphs for specific target systems. In Chap. 4, we describe how to safely incorporate data-parallelism into the core language. Chap. 5 adds unshared mutability via uniqueness types and object orientation to the core language before shared mutability is introduced in Chap. 6. For every new feature we add to the core language we discuss in each respective chapter why the semantics of the language remains invariant under automatic task-parallelization so that the new feature does not reduce the precision of the data flow analyses from Chap. 2. In Chap. 7, we present the **funkyImp** compiler and runtime system which implement the language design and analyses defined in the preceding chapters. Chap. 8 summarizes the first part of the thesis and discusses future work.

In Part II of this thesis, in Chap. 9, we present an analysis tool to predict parallelization hazards in sequential C++ programs based on models that describe the intended manual parallelization.

Generally, the method is useful if automatic parallelization methods fail to achieve good results and a reformulation of legacy applications in a form or language that is more amenable to automatic parallelization is too costly. Performing manual parallel decomposition, especially in large code bases, requires high effort and the results are hard to predict as detailed knowledge about the different components is usually spread across different programmers and teams. In addition, the programmers need to model the parallel execution at least informally in order to perform a parallel decomposition. So instead of iteratively modeling and performing the expensive implementation of parallel decompositions to find a suitable solution, all promising models can be evaluated for parallelization hazards and thus, the number of models that actually need to be implemented can be

reduced. Usually, evaluation of a model is cheaper to perform than the actual implementation of the model so that this approach may reduce the effort of parallelizing large legacy applications. In a case study, we construct and apply such a parallelization model to derive a points-to analysis which detects parallelization inhibiting resources in a large industrial C++ application while analyzing only specific class hierarchies instead of the whole program.

Part I

The funkyImp Language

Chapter 2

Core Language and Task-Parallelism

In this chapter, we introduce the **funkyImp** core language in combination with the necessary analyses and definitions to extract and exploit task-parallelism from this language.

The chapter is structured as follows. First, we discuss data flow in the context of side-effecting and side-effect free languages in Sec. 2.1, before we define the side-effect free **funkyImp** core language in Sec. 2.2. Afterwards, in Sec. 2.3, we formally define a data flow analysis for our language and use this analysis to construct *data flow graphs*, which encode the existing data flow dependencies between expressions inside a single function. Then, in Sec. 2.4, the data flow graph is enriched by control flow dependencies, in order to construct *task graphs* which encode both, data flow and control flow. The semantics for task graphs, defined in Sec. 2.5, can exploit that all dependencies are encoded explicitly inside task graph, so that any nodes in a task graph, which are mutually unreachable, may be evaluated in parallel. In Sec. 2.6 we show that the semantics of a task graph is identical to the semantics of the function it was generated from. Finally, in Sec. 2.7, we apply the analysis to a simple Fibonacci based benchmark to study the effects of recursion on the performance of task-parallelism.

2.1 Data Flow

In the context of programming languages, data flow describes the dependencies between expressions through data. An expression e_2 is said to be data dependent on another expression e_1 if e_2 references (consumes) data produced by e_1. In side-effecting languages data can be produced by writing to memory in e_1 and afterwards reading the same memory in e_2. Here, allocation and construction of data are two separate actions. Therefore, the same variable or memory address may link different producers and consumers at different program points. In principle, every consumer of a memory address may become a producer by writing to the address. So, in order to construct the data flow into a specific expression, all preceding expressions must be checked for writes to the memory addresses consumed by the specific expression of interest. In general, finding all possible writes to the memory addresses consumed by the expression of interest requires a complex, over-approximating and inter-procedural points-to analysis [63].

In contrast, side-effect free languages require that data must be constructed at allocation time and cannot be modified afterwards. So for these languages, the producer of some data is uniquely defined by the expression constructing the data. Data flow happens only from the producer of a value to those expressions (consumers) that explicitly reference the value by name. Hence, in the context of side-effect free languages, reasoning

about data flow into an expression e reduces to syntactically extracting all identifiers that are not defined in the expression. The analysis is intra-procedural, it is not necessary to transitively compute the values flowing into function parameters at the call site of a function to prove independence expressions referencing the function parameters by name inside the function.

2.2 Core Language Grammar

In order to allow fast and precise extraction of data flow dependencies, we base **funkyImp** on a side-effect free core language which is shown in Fig. 2.1.

$$
\begin{aligned}
\text{program } p &::= f^* \\
\text{fun } f &::= x^+ = e \\
\text{exp } e &::= \textbf{let } x = e \textbf{ in } e \mid e \; op \; e \mid \textbf{if}(e) \; e \textbf{ else } e \mid (e) \mid x \; e^* \mid v \\
\text{values } v &::= ... \\
\text{id } x &::= ...
\end{aligned}
$$

*Figure 2.1: Core language grammar of **funkyImp**.*

To simplify the discussion, we present a minimalistic core language. The core language grammar of **funkyImp** is based on a strict, first order core language which is deterministic and side-effect free. Our language is strict in order to not limit parallelism through laziness [57, 70], as was discussed in Sec. 1.4.3. We assume the standard semantics for the given language constructs. More details about the full **funkyImp** language implementation can be found in Chap. 7 and tutorials for the full language can be found in the online documentation [59]. Furthermore, we assume that all value and function names are unique to avoid clustering the notation with scoping issues.

2.3 Data Flow Analysis and Graphs

Next, we define the data flow analysis $Dep[\![e]\!] : \text{exp} \to \mathcal{P}(\text{id})$ which computes the data flow into an expression e by syntactically extracting all undefined value names referenced inside e.

To shorten the notation, we define $e[e_0, ..]$ which denotes an expression from the grammar from Fig. 2.1 where the sub-expressions e_i are pattern matched from left to right to the occurrences of e in the grammar rule that matches $e[e_0, ..]$. For example, given $e[e_0, e_1] = (e_l) \; op \; (e_r)$, the grammar rule $e \; op \; e$ matches $e[e_0, e_1]$ so that $e_0 = (e_l)$ and $e_1 = (e_r)$. $Dep[\![e]\!]$ computes the set of identifiers x which flow into e, where we apply the first equation from top to bottom that matches e.

$$\begin{aligned} Dep[\![let\ x = e_1\ in\ e_2]\!] &\triangleq Dep[\![e_1]\!] \cup (Dep[\![e_2]\!] \setminus \{x\}) \\ Dep[\![x]\!] &\triangleq \{x\} \\ Dep[\![e[e_0, .., e_n]]\!] &\triangleq \bigcup_{e_i \in \{e_0,..,e_n\}} Dep[\![e_i]\!] \end{aligned} \tag{2.1}$$

Now, Eqs. 2.1 can be used to compute data flow graphs. A data flow graph is a directed graph where each node is an expression and each edge from e_1 to e_2 defines a data flow dependency of e_2 on e_1. To shorten the notation, we define $e[e_i] \triangleq e[e_0, .., e_n]$ where $e_i \in \{e_0, .., e_n\}$.

In the following, we formally define an analysis that, given the body of a function $x^+ = e$, computes a data flow graph which respects the data flow according to Eq. 2.1. Formally, a data flow graph is a pair (V, E) where $V' \subseteq \{(e, e') \mid e, e' \in \exp\}$ and $E' \subseteq \{(u, v) \mid u, v \in V\}$. The union of graphs is defined as a pairwise union

$$(V_1, E_1) \sqcup (V_2, E_2) \triangleq (V_1 \cup V_2, E_1 \cup E_2). \tag{2.2}$$

The analysis uses the function $Res[\![e]\!] : \exp \to \exp$ to compute the result of let-expressions (e.g. $Res[\![let\ x = 5\ in\ let\ y = 7\ in\ x * y]\!] = x * y$), where $e[e_i := e']$ denotes that expression e_i is replaced by e' in e.

$$\begin{aligned} Res[\![let\ x = e_1\ in\ e_2]\!] &\triangleq Res[\![e_2]\!] \\ Res[\![e[e_i]]\!] &\triangleq e[e_i := Res[\![e_i]\!]]. \end{aligned}$$

The function **insert** adds node x to the graph (V, E) and connects the node to all nodes x depends on via data flow

$$\textbf{insert}\ (V, E)\ x\ e \triangleq (V \cup \{(x, e)\}, E \cup \{(u, x) \mid u \in Dep[\![e]\!] \wedge u \neq x\}). \tag{2.3}$$

Given a function $x^+ = e$, the analysis $DF[\![e]\!]_{\mathsf{true}}$ generates a data flow graph for the function body e in the following way: For every let-expression $let\ x = e_1\ in\ e_2$ in the body, the analysis creates a new sub-graph that contains e_2 and a new node which computes $x = e_1$ together with data flow edges that link the new node to the required nodes in the sub-graph. The boolean flag f is used to track whether an inner expression ($f = \mathsf{false}$) or an outermost expression ($f = \mathsf{true}$) is currently evaluated.

$$\begin{aligned} DF[\![let\ x = e_1\ in\ e_2]\!]_f &\triangleq \textbf{insert}(DF[\![e_1]\!]_{\mathsf{false}} \sqcup DF[\![e_2]\!]_f, x, Res[\![e_1]\!]) \\ DF[\![e[e_i]]\!]_f &\triangleq \begin{cases} \textbf{insert}(\bigsqcup_{e_i \in e} DF[\![e_i]\!]_{\mathsf{false}}, e, e[e_i := (Res[\![e_i]\!])]) & : f \\ \bigsqcup_{e_i \in e} DF[\![e_i]\!]_{\mathsf{false}} & : \neg f \end{cases} \end{aligned}$$

The following example illustrates the analysis result:

Given the function `a b=e` from Fig. 2.2, and connecting all nodes in the data flow graph $DF[\![e]\!]_{\mathsf{true}} = (V, E)$ that have no predecessor to a node labeled 'fun a b' yields the data flow graph shown in Fig. 2.3. Here, a node $(e, e') \in V$ is labeled '$e = e'$' except for the result node which is simply labeled 'e''. In addition to the analysis result, edge labels indicating the source of data dependencies are shown.

```
a b =
  let c = let e = 2*b in 3*e in
  let d = let f = 5*b in 7*f in
  let res = c+d in res
```

Figure 2.2: Example program used to construct the data flow graph in Fig. 2.3.

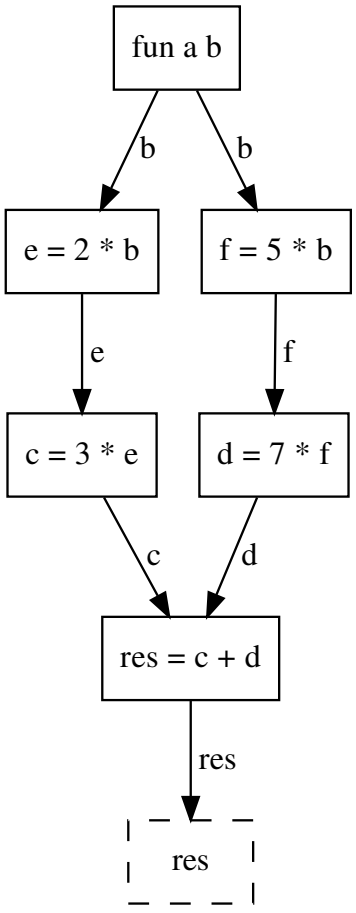

Figure 2.3: Data flow graph computed from function a defined in Fig. 2.2.

2.4 Control Flow

In order to capture all possible dependencies between expressions in our core language, control flow must be taken into account in addition to data flow.

If the evaluation of expressions depends on a control flow decision (e.g. the expression is in a branch of an if-expression), then they must not be evaluated, before the control flow decision was evaluated although no data dependency between the control flow decision and the expressions may exist.

Note that there are speculative methods that may allow branches to execute concurrently before the control flow decision was made [81]. In future work, these methods could be added as a further optimization to our compiler.

In order to incorporate such control flow dependencies into a data flow graph, we define the analysis $TG[\![e]\!]^{\perp}_{\mathsf{true}}$ which extends $DF[\![e]\!]$ by a special case for branches. This requires the function **insertCF** which adds edges from every node to its respective control flow node in addition to the data flow edges added by **insert**.

$$\textbf{insertCF}\ (V,E)\ x\ e\ c \triangleq \begin{cases}(V', E' \cup \{(c,x)\}) : & c \neq \bot \\ (V', E') : & \text{else}\end{cases} \tag{2.4}$$

where $(V', E') = \textbf{insert}\ (V,E)\ x\ e$.

The analysis $TG[\![e]\!]^{\bot}_{\mathsf{true}}$ adds a node labeled $if\ e_1$ for every if expression $if(e_1)\ e_2\ else\ e_3$ and edges from this node to the control flow nodes $e_1 = true$ and $e_1 = false$ which represent the true and false branch respectively.

$$\begin{aligned}
TG[\![let\ x = e_1\ in\ e_2]\!]^c_f &\triangleq \textbf{insertCF}(TG[\![e_1]\!]^c_{\mathsf{false}} \sqcup TG[\![e_2]\!]^c_f, x, Res[\![e_1]\!], c) \\
TG[\![if(e_1)\ e_2\ else\ e_3]\!]^c_f &\triangleq \\
(V \cup \{(e_1 = true, ()), &(e_1 = false, ())\}, E \cup \{(if\ e_1, e_1 = true), (if\ e_1, e_1 = false)\}) \\
&\text{where } (V,E) = \textbf{insertCF}(TG[\![e_2]\!]^{e_1=true}_f \sqcup TG[\![e_3]\!]^{e_1=false}_f, if\ e_1, Res[\![e_1]\!], c) \\
TG[\![e[e_i]]\!]^c_f &\triangleq \begin{cases}\textbf{insertCF}(\bigsqcup_{e_i \in e} TG[\![e_i]\!]_{\mathsf{false}}, e, e[e_i := (Res[\![e_i]\!])]) & : f \\ \bigsqcup_{e_i \in e} TG[\![e_i]\!]^c_{\mathsf{false}} & : \neg f\end{cases}
\end{aligned}$$

The following example shows control flow dependencies inside a task graph. Given function `a b=e` as defined in Fig. 2.4, and connecting all nodes in $TG[\![e]\!]^{\bot}_{\mathsf{true}} = (V,E)$ that have no predecessor to a node labeled 'fun a b' yields the task graph in Fig. 2.5.

```
a b =
 let c = let e = 2*b in 3*e in
 let d = let f = 5*b in 7*f in
 let g = if(c<2)
    let w=4*b in let x=3*d in x+w
  else
    let w=6*b in let x=9*d in x*w
 in let res=c+d+g in res
```

Figure 2.4: Example program illustrating control flow in the corresponding task graph in Fig. 2.5.

Note that the analysis $TG[\![e]\!]$ extracts parallelism only for let-expressions inside a single function $x^+ = e$. The analysis could be easily adapted to extract *all* static task-parallelism (e.g. the operands of binary operators could be evaluated in parallel before the operator is applied) without affecting any of the methods we present in this thesis. We have chosen parallelization at the level of let-expressions instead, to keep the visualization of task graphs simple and small.

2.5 Task Graph Semantics

In this section we present semantics for task graphs defining how to execute the parallel program denoted by a task graph.

A task graph node may be evaluated as soon as all predecessor nodes, if any, have finished evaluation. After evaluation of a node has finished, the result is communicated along the outgoing edges to the successor nodes, potentially triggering the evaluation of

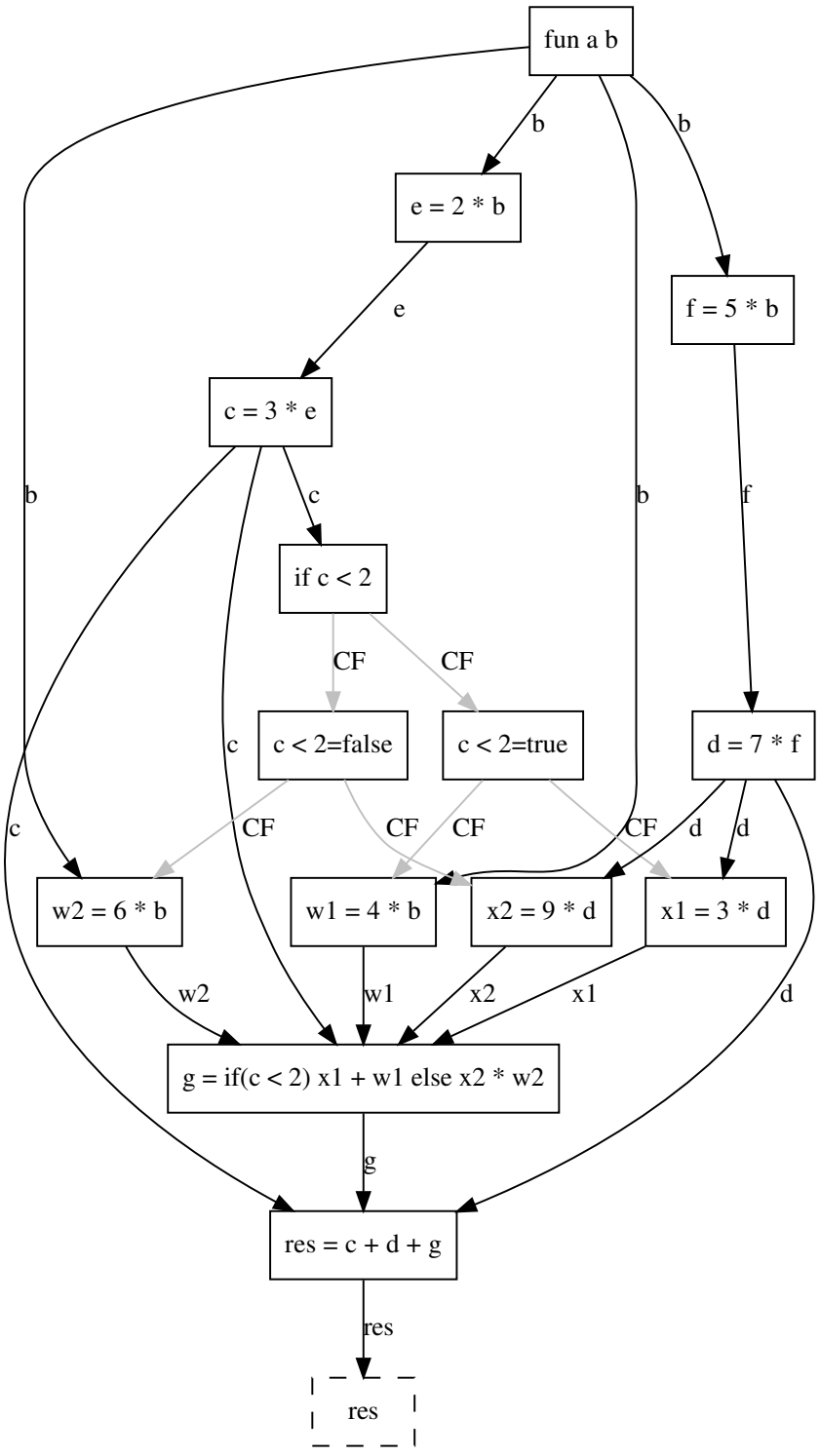

Figure 2.5: Task graph with data flow and control flow edges (labeled CF) computed from function a defined in Fig. 2.4.

more nodes. We discuss some of the possible communication mechanisms in Chap. 3 and Chap. 7. Evaluation of a task graph finishes after any sink node was evaluated.

This definition shows that a task graph is a high level representation of a parallel program where each node represents a program fragment that may execute in parallel with all nodes which are *mutually* unreachable in the graph. For example, the nodes defining e and c in Fig. 2.3 may execute in parallel with the nodes defining f and d.

2.6 Semantic Invariant Parallelization

The parallelization of a function $x^+ = e$ is given by the semantics of the corresponding task graph $TG[\![e]\!]$. It is clear that the parallelization of a function must have the same semantics as the original function. Therefore, we introduce Lemma 1 which guarantees this property.

Lemma 1 describes the most important property of our language design. It allows to reason, write and debug sequential code that can be automatically transformed into task-parallel code using $TG[\![e]\!]$. Note that Lemma 1 implies that our intra-procedural analysis $TG[\![e]\!]$ is sound. Our analysis is also complete, because $TG[\![e]\!]$ extracts all dependencies syntactically and without over-approximation. We refrain from a formal proof for soundness and completeness, comparing the formal semantics of the core language and of task graphs, because our core language is extremely simple. It is obvious that the evaluation order of independent expressions does not matter and that precise flow analysis in our core language is possible due to the absence of side-effects. The task graph we have extracted is analogous to the program dependence graph [44] used to automatically parallelize imperative programs like C++ citestreit2012sambamba. Note that languages which allow unrestricted side-effects, like C++, require an inter-procedural points-to analysis to compute the flow for task graphs with reasonable precision. The strength of the language design presented in this and the following chapters of this thesis lies in the observation that, in contrast to existing languages like C++, we can keep the task graph analysis intra-procedural and highly efficient as well as precise although we add mutable state, arrays and other features that allow to achieve outstanding performance in C++ and our own language. In the following we present a proof sketch of Lemma 1.

Lemma 1. *The semantics of any function $x^+ = e_f$ from our language is identical to the semantics of the corresponding task graph $TG[\![e_f]\!]^{\perp}_{\mathsf{true}}$.*

Proof. Any expressions e, e' from our side-effect free language, which mutually have no flow into each other, cannot affect each other. Thus, the result of the evaluation of e and e' is independent of the order in which e and e' are evaluated. The partial order defined by $TG[\![e_f]\!]^{\perp}_{\mathsf{true}}$ respects all data and control flow dependencies of the function $x^+ = e_f$ by construction. The evaluation of task graphs, as defined in Sec. 2.5, guarantees that all possible evaluation orders of the task graph's nodes are compatible with the partial order defined by the task graph. Hence, the semantics of any function $x^+ = e_f$ is identical to the semantics of the corresponding task graph $TG[\![e_f]\!]^{\perp}_{\mathsf{true}}$. □

Although our task graph analysis is precise, strictness and modularity of functions have a price in terms of exploitable parallelism. In order to parallelize a whole program using our intra-procedural analysis, a task graph must be computed for every function and function applications are evaluated by evaluating the corresponding task graph. Strictness requires that we evaluate all function arguments before evaluating the function application, even if evaluation of some of the arguments could be postponed. So in the context of the whole program, some parallelism is lost for function arguments that are not required immediately inside the function application. In case the loss of parallelism is relevant for the overall program performance, the compiler can simply inline the affected function application and thus regain all parallelism. In Chap. 3 we present methods to predict the execution times of task graphs which can be used to statically determine if inlining of a function would yield a viable speedup.

In spite of the effective extraction of task-parallelism exposed by our core language, our language is not yet sufficient for high performance programming. The lack of arrays, side-effects and non-determinism makes it impossible to implement many problems efficiently. Throughout this thesis, we will extend our simple core language by data-parallelism, linear side-effects and actors to accommodate the different forms of parallel programming paradigms and side-effects required to build a general purpose programming language which allows to write programs with performance on par with parallel C++. With every new feature we add to our core language, great care is taken that Lemma 1 is not violated, which is especially interesting when side-effects are added to the language. When necessary, the chapters introducing these new language features will extend $TG[\![e]\!]$ by additional edges and nodes, thereby placing additional constraints on the execution order of nodes in order to ensure that Lemma 1 is not violated.

2.7 Recursive Task Generation

Nevertheless, even our simple core language can be used to generate decent speedups when traversing immutable, recursive data structures, as discussed in this section.

Consider the simple Fibonacci example in Fig. 2.6. Note that an iterative version of Fibonacci outperforms any parallel implementation by far, because the parallel implementation unnecessarily recomputes most data. Nevertheless, the parallel implementation is analogous to a parallel binary tree walk which is interesting in its own right.

```
fun fib n =
   if(n>1)
      let l=fib (n-2) in
      let r=fib (n-1) in
      l+r
   else
      n
```

Figure 2.6: Recursive Fibonacci-sequence implementation.

By inspecting the corresponding task graph in Fig. 2.7, we see that at every level of the recursion at most two tasks are spawned so that less than 2^l tasks have been spawned at recursion level l.

The amount of computation per task decreases with every recursion level so that more and more inefficient tasks are created at deeper recursion levels. Obviously, the actual task size and recursion depth depend on the parameter n and are not statically known. Nevertheless, we can statically restrict the number of spawned tasks to a sensible amount. Using measurements and theoretical work [89], a good upper bound on the number of tasks for a specific system can be found. In order to place a limit on the number of tasks generated at run-time, our compiler generates a fully sequential version next to the parallel version of fib and automatically switches from the parallel to the sequential version as soon as the recursion depth indicates that the number of spawned tasks exceeds the upper bound of tasks for the system.

This fully automatic switch may not always be the best decision, so the programmer can use pragmas to specify alternative dynamic criteria to switch between parallel and sequential execution inside recursive functions. More advanced, dynamic profiling

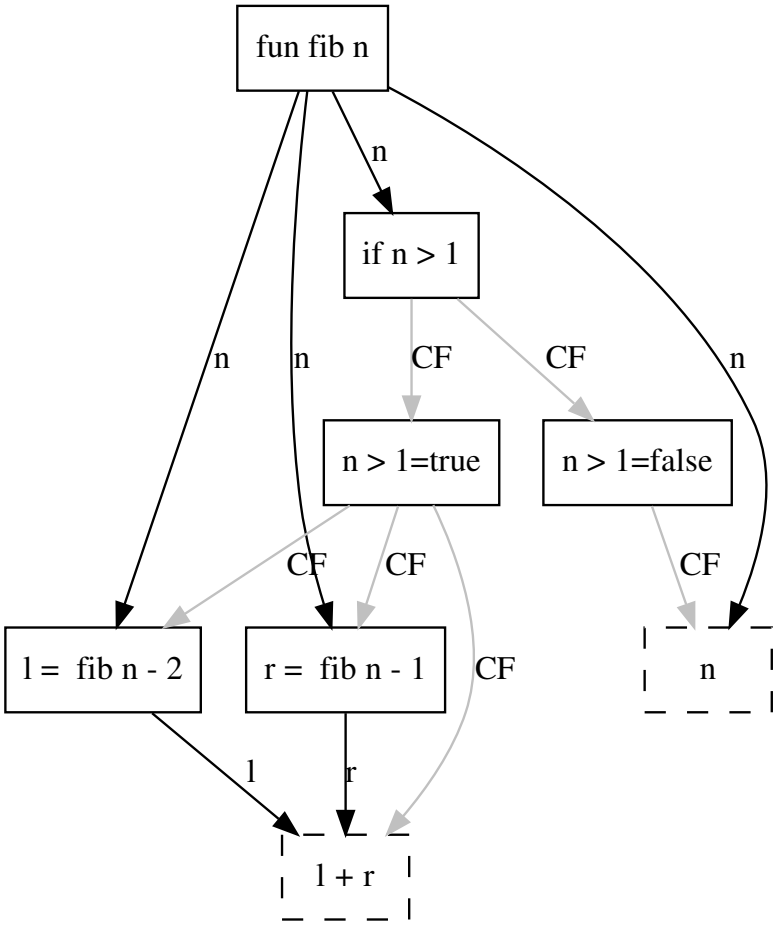

Figure 2.7: Task graph for the fibonacci implementation from Fig. 2.6.

methods [130] may further improve the performance of task graphs where the full structure cannot be statically inferred. Fig. 2.8 shows the performance of different Fibonacci sequence implementations.

Code	parallel	avrg. exec. time [s]	σ [s]
rec. fib	no	0.88	0,0007
rec. fib (switch)	yes	0.19	0,0007
rec. fib (no switch)	yes	27.9	0,02

*Figure 2.8: Different versions of computing the Fibonacci sequence, generated by the **funkyImp** compiler from the code shown in Fig. 2.6, are evaluated. All times were measured on an Intel Core i7 CPU 860 @ 2.80GHz with ca. 12 GB/sec memory throughput and hyper threading and speedstep disabled.*

The version that includes the automatic switch from parallel to sequential (marked with (switch)) outperforms the sequential implementation by a factor of 4.63 on a machine with 4 cores. This shows the effectiveness of the automatic parallelization. The super linear speedup may be caused by better cache efficiency of the parallel version as indicated by profiling. The parallel version without the switch to the sequential version, marked with (no switch), is orders of magnitude slower than all other versions. Here the run-time is flooded with minuscule tasks which yield slowdown.

2.8 Conclusion

The methods shown in this chapter can be used to precisely extract all statically available task-parallelism from a function given in the core language. Note that *dynamic* parallelism is discussed in Chap. 6. Obviously, the available static parallelism depends on the algorithm chosen by the programmer. Nevertheless, for a given algorithm implementation we have extracted all statically available task-parallelism so that executing the corresponding task graphs is maximally static parallel for the given function.

In contrast, the limited precision of points-to analyses available for languages with unconstrained aliasing and side-effects may severely reduce the amount of statically exploitable task-parallelism.

In the next chapter, we present our task graph preconditioning algorithm which uses a target hard- and software specific model to predict realistic execution times of task graphs and merges parallel nodes until a minimum speedup on the specific target system is guaranteed. By merging tasks from the maximally parallel task graph, it is guaranteed that the optimal (statically parallel) task graph is addressable. In fact, all semantics preserving, static parallel decompositions that a programmer could specify manually are addressable by our approach.

Chapter 3

Optimization of Task-Parallelism

In this chapter we discuss the static scheduling of task graphs that may contain many small tasks. Previous *static* scheduling algorithms assume negligible run-time overhead of spawning and joining tasks. We show that this overhead is significant for small- to medium-sized tasks which can often be found in automatically generated task graphs.

By comparing real-world execution times of a schedule to the predicted static schedule lengths we show that the static schedule lengths are uncorrelated to the measured execution times and underestimate the execution times of task graphs by factors up to millions if the task graph contains small tasks. The static schedules are realistic only in the limiting case when all tasks are vastly larger than the scheduling overhead. Thus, for non-large tasks the real-world speedup achieved with these algorithms may be arbitrarily bad, maybe using many cores to realize a speedup even smaller than one, irrespective of any theoretical guarantees given for these algorithms.

We derive a model to predict parallel task execution times on symmetric schedulers, i.e. where the run-time scheduling overhead is homogeneous. The soundness of the model is verified by comparing static and real-world overhead of different run-time schedulers. Finally, we present a clustering algorithm which guarantees a real-world speedup.

Given a task graph as defined in Chap. 2, two target system specific constants and the average execution time of each node in the task graph, the algorithm optimizes the task graph by merging parallel nodes that do not guarantee a user defined, minimum speedup per core.

In order to obtain the average node execution times, required for our scheduling algorithm, the compiler applies analyses which predict the average execution time of program fragments [24,123] given by the individual task graph nodes. In case these execution times are not precise enough, the **funkyImp** run-time supports profiling to improve the result.

The performance model for task graph evaluation and the static scheduling algorithm we present in this chapter were published in [61].

This chapter is structured as follows. First, we discuss existing static scheduling algorithms in Sec. 3.1. Then, in Sec. 3.2 we show that the measured execution times of a simple task graph have a non-linear relationship to the static schedule length from traditional scheduling algorithms that ignore the scheduling overhead. Afterwards, in Sec. 3.3, we derive a statistical model to predict scheduling overhead for symmetric schedulers where the run-time scheduling overhead is homogeneous. This is followed by Sec. 3.4, where we show that our overhead model accurately predicts the scheduling overhead on several platforms and scheduler implementations. The model is used in Sec. 3.5, to construct a clustering algorithm which guarantees a user defined minimum speedup per core in $O(n^3)$, where n is the number of nodes in the task graph. Sec. 3.6 presents benchmarks showing

how our algorithm improves the average speedup of a large set of randomly generated task graphs with small and large task sizes. Finally, we discuss related work in Sec. 3.7, as well as possible future work in Sec. 3.8.

3.1 Existing Static Scheduling Algorithms

Due to the NP-completeness of many instances of the static scheduling problem, a set of heuristics trying to approximate the best solution have been proposed. Kwok [80] and McCreary [77] have compared the quality of a range of well-known heuristics in terms of the static schedule length predicted by the different scheduling heuristics for a versatile set of input task graphs. For some heuristics (e.g. linear clustering [47]) it has been proven that the produced schedule length (the makespan as predicted by the heuristic) is no longer than the fully sequential schedule. For large-grain task graphs it has been shown that the schedule lengths from greedy algorithms are within a factor of two of the optimal schedule [51].

To the best of our knowledge, no comparison of static schedule lengths to the real-world execution time of the schedule has been undertaken. If the real execution time of a schedule on a specific target platform does not at least roughly correlate to the static schedule length, then any guarantees or schedule length advantages of one heuristic compared to another are purely theoretical as the static schedule does not model reality.

In contrast to a Gantt-chart's implication that tasks are started at a predefined time, most parallel systems (e.g. [67,108]) implement a dynamic signaling mechanism to spawn and join tasks as soon as all preconditions are satisfied. This removes the burden of providing hard real-time guarantees for the soft- and hardware which may produce unnecessary long schedules as worst case estimates must be used everywhere.

Our measurements show that on some platforms the overhead is in the order of $2 \cdot 10^4$ [clocks] so that it can be ignored only in the limiting case when all tasks are in the order of 10^6 [clocks] and larger. The maximum possible task size for a game or numerical simulation running at 60 frames per second on 2 GHz CPUs is in the order of 10^7 [clocks] and typically much smaller for non-trivially parallelizable problems. This shows that the overhead is relevant for typical parallel applications. For task graphs that contain tasks with sizes in the order of the overhead, traditional scheduling algorithms may produce schedules with arbitrarily bad speedup (e.g. a speedup smaller than one on more than one core compared to the fully sequential schedule) as they ignore the run-time overhead (cost of spawning and joining tasks). This includes algorithms that in theory guarantee a schedule length shorter than the fully sequential schedule. The insufficient speedup produced by these algorithms is especially harmful on battery driven devices that could shut down cores that are used inefficiently.

Furthermore, the neglected overhead may shift data ready times used to perform scheduling in most algorithms asymmetrically, so that tasks which appear to run in parallel for the algorithm will not run in parallel in reality.

Our main contributions to solve these problems are:

- We derive a generic model to predict run-time scheduler overhead for symmetric schedulers, i.e. the scheduling overhead is homogeneous.
- We show that our model accurately predicts scheduling overhead for stealing [17] and non-stealing schedulers on different hardware.

- We define a task granularity for communication-free task graphs related to the parallel task execution times on a specific platform.
- We present the first clustering algorithm that guarantees a minimum real-world speedup per core for communication-free task graphs.

3.2 Example

We will examine the task graph in Fig. 3.1. Each node contains an additional label $w : work$, where $work$ states the average execution time of the node in clock ticks on a specific target system.

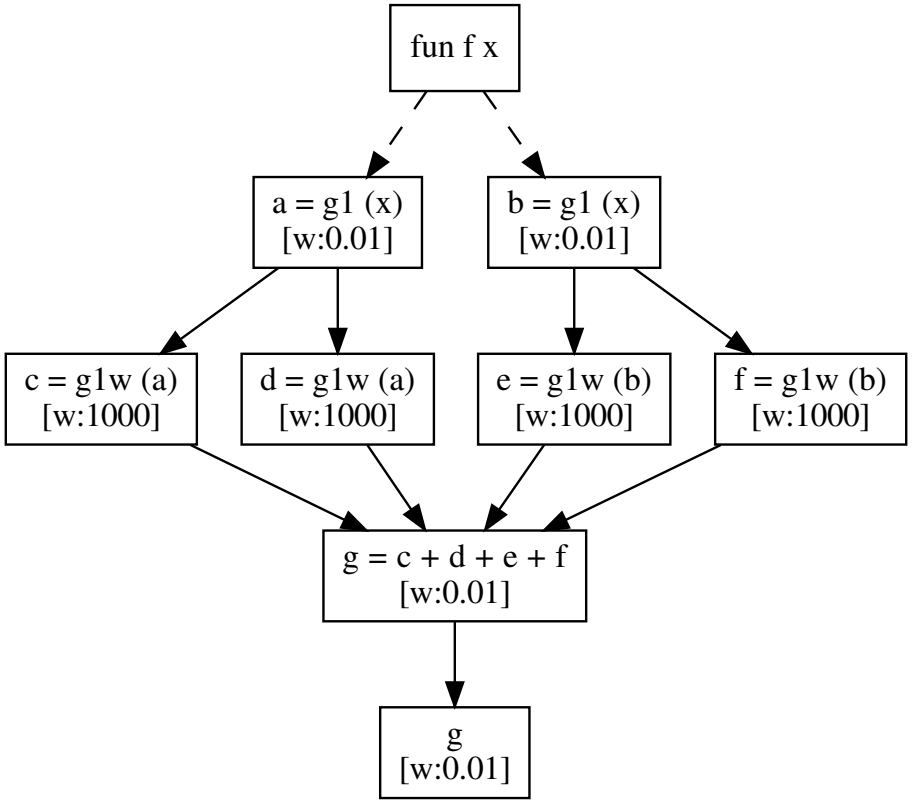

Figure 3.1: Example task graph containing small tasks.

Scheduling the program depicted in Fig. 3.1 on multi-core hardware appears trivial. Given a machine with more than four cores, perfect scheduling [36] can be applied to produce the optimal schedule, according to the scheduler's model of program execution, in $O(n + m)$. This may produce the Gantt-chart shown in Fig. 3.2.

Although the Gantt-chart implies that tasks may start execution at a specified time this is rarely implemented. In order to start tasks based on the times from the chart, hard real-time constraints need to be placed on the executing hard- and software and sound worst case execution times for all tasks must be calculated. In addition, the executed schedule might be less than optimal as some tasks may be able to execute before the worst case finish time of their predecessors because the predecessors finished earlier than the conservative estimate.

Most scheduler implementations (e.g. [67,108]) signal waiting tasks as soon as all their preconditions have been fulfilled so that they can start executing as soon as possible.

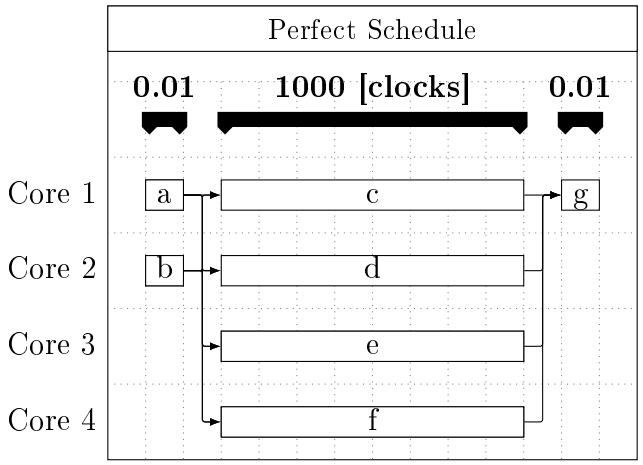

Figure 3.2: Gantt-chart for task graph in Fig. 3.1 produced using perfect scheduling [36]. Arrows show dependencies but do not require any time according to the scheduling algorithm. The predicted sequential execution time of 4000 clocks compared to the predicted parallel execution time of 1000 clocks suggests a speedup of about 4. Actually executing the task graph yields a measured execution time of ca. 4000 clocks on a 4 Core Nehalem for the fully sequential program and a measured ececution time of 6000 clocks for the fully parallel program. In contrast to the prediction shown in the Gantt-chart, sequential execution is about 1.5 times faster than parallel execution, not 4 times slower.

Usually, spawning tasks without preconditions also requires to signal to another thread.

The Gantt-chart in Fig. 3.2 suggests a parallel execution time of about $0.01 + 1000 + 0.01 = 1000.01$ clocks, so a speedup of about 4 compared to the sequential execution of ca. $2 \cdot 0.01 + 4 \cdot 1000 + 0.01 = 4000.03$ clocks is predicted. Actually running the program in its fully parallel version using TBB's [67] stealing scheduler takes about 6000 clocks per task graph execution on a 4 Core Nehalem. In contrast, executing the sequential version of the program on the same hardware yields an execution time of ca. 4000 clocks which is about 1.5 times faster than the parallel version.

Apparently, the existing scheduling algorithms do not model the real-world scheduling process, but an artificial scheduler that has no run-time overhead and always yields linear speedup. This leads to unrealistically small execution time predictions from these scheduling algorithms. The existing literature does not define a grain size for task graphs without communication costs (and an unrealistic one for small tasks with small communication costs). In Sec. 3.3 we define such a grain size in direct connection to the parallel execution time of a task on a specific platform.

In Sec. 3.4 and Sec. 3.6, it will be shown that the scheduling overhead is not negligible on real-world systems even for bigger task sizes up to 10^5 clocks and more (depending on the specific hardware and scheduler implementation). This emphasizes that the effect is relevant for normal task graphs that do not contain minuscule tasks.

3.3 Thread Coordination Cost Model

In this section, a model for the prediction of the execution time of a fork/join task graph which accounts for the scheduling overhead is derived, improving the execution time prediction accuracy by up to 1000% (compared to previous scheduling algorithms which neglect the scheduling overhead for smaller tasks). The model assumes that the fork/join

task graph is executed in isolation, so that all computational resources can be utilized.

Since the actual execution times on real hardware fluctuate heavily depending on the overall system state, a stochastic model is developed. Adve and Vernon [1] have found that random fluctuations have little impact on the parallel execution time, so that the expectation values we derive should be a good model of the real execution time. First, we derive the model for a stealing scheduler as implemented in Thread Building Blocks 4.1 (TBB [67]). Then we generalize the model to schedulers with symmetric scheduling overhead, i.e. the overhead is homogeneous.

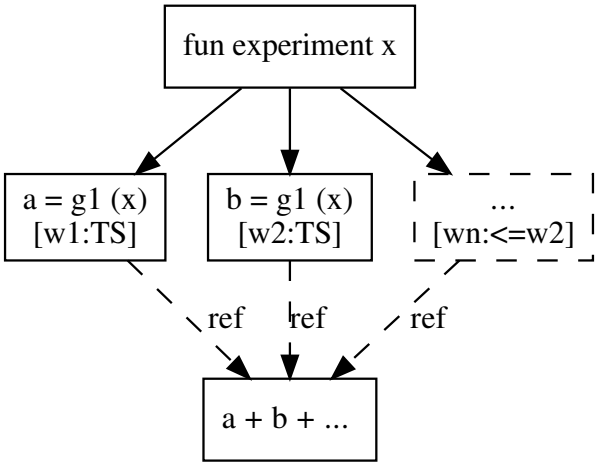

Figure 3.3: Generic fork/join task graph with n nodes. Here g1 performs a trivial loop that takes TS clocks and does not interfere with the other tasks. The tasks are spawned by the root node and the last task is notified as soon as all predecessor tasks have finished.

3.3.1 Two Node Fork/Join Graphs

In the following, the execution of fork/join task graphs with two tasks are modeled. After deriving a model for two tasks, we will extend the model for more tasks. Our clustering algorithm will apply these models on generic task graphs by decomposing them into fork/join task graphs.

The work weight associated with each task represents the average execution time of the task on the target platform including all costs (i.e. resource contention, cache effects, etc.) except for the dynamic scheduling costs. Without loss of generality, we assume that the task calculating a (from Fig. 3.3 with $n = 2$) is executed locally after the second task calculating b has been spawned at time t_0. For stealing schedulers, spawning means that the task is enqueued in a list on the thread where the task is spawned. If no other idle core steals the task from this list for parallel execution, the task will eventually be executed locally on the thread that originally spawned the task. The spawned task can be stolen (and hence be executed in parallel) from the time it was spawned up until the first task finished execution at $t_0 + TS$ (where $TS = \max\{w_1, w_2, .., w_n\}$ is the maximum size of all tasks because the largest task determines the overall execution time) and starts to execute the spawned task itself. For convenience, we set $t_0 = 0$.

First, the time to spawn a task on another core is calculated.

Since the stealing process is independent from the spawn (other idle cores check regularly if there is something to steal), the normalized probability that a steal attempt is

made at time t is

$$p_{\text{attempt}} = \frac{\sigma}{TS}, \tag{3.1}$$

where $0 < \sigma < 1$ determines the steal attempt frequency which depends on the actual scheduler and hardware. The frequency is smaller than one because less than one attempt per clock can be made on real systems. A steal attempt need not succeed, e.g. if there is nothing to steal. The probability that the available task was not stolen until time t is given by the inverse of the probability that it was stolen:

$$p_{\text{not-stolen}}(t) = 1 - \int_0^t p_{\text{attempt}}\, dt = 1 - \frac{\sigma}{TS} \cdot t. \tag{3.2}$$

So the probability that a steal attempt is successful at time t is given by the probability that it was not yet stolen times the steal attempt probability (when within the time frame where the task can be stolen at all):

$$p_{\text{stolen}}(t) = \begin{cases} p_{\text{not-stolen}}(t) \cdot p_{\text{attempt}} & 0 \leq t \leq TS \\ 0 & \text{else.} \end{cases} \tag{3.3}$$

Finally, the expected time T_{steal} for the second task being stolen is

$$T_{\text{steal}} = \int_0^{TS} p_{\text{stolen}}(t) \cdot t\, dt = \frac{(3\sigma - 2\sigma^2)}{6} \cdot TS =: \beta \cdot TS. \tag{3.4}$$

The overall parallel execution time (PET) of the potentially stolen task is given by

$$pet(TS) = \beta \cdot TS + \alpha' + TS. \tag{3.5}$$

Here, α' is the expectation value of the fixed overhead required to execute the steal (which is initiated at time T_{steal}) and the join to wait for both tasks. After all overhead is accounted for, the time TS needed to execute the task must be added. Like σ, α' depends on the hard- and software and must be obtained by running an experiment on the specific target platform.

The speedup achieved when running all tasks in parallel is obtained via

$$su(w_1, ..., w_n) = \frac{\sum_i^n w_i}{pet(\max\{w_i\})}, \tag{3.6}$$

where $\sum_i^n w_i$ is the sequential execution time (SET).

For non-stealing schedulers (e.g. MPI [108] based code), the derivation is essentially identical. The tasks are signaled rather than stolen. The signaling is implemented via MPI_send and MPI_receive or MPI_barrier so that only negligible data sizes are communicated, transmission of larger amounts of data is not modeled as the article's scope is limited to cost-free communication. Hence, p_{attempt} becomes the probability that the signal starting the second task arrives at time t. In addition, the second task can be signaled long after the first finished executing when a non-stealing scheduler is used. Assuming there exists a maximum time $T_{\max} > TS$, after which the second task is guaranteed to have been signaled, we substitute $T_{\max}$ for all TS in Eqs. 3.1 to 3.4 to obtain the expectation time that the signal arrives $T_{\text{signal}} = \beta \cdot T_{\max}$. Rewriting this with $T_{\max} = TS + \delta T$ we get

$$T_{\text{signal}} = \beta \cdot TS + \beta \cdot \delta T. \tag{3.7}$$

As $\beta \cdot \delta T$ is a hardware dependent constant we can subsume it into $\alpha = \alpha' + \beta \cdot \delta T$ and add it to the final parallel execution time prediction by substituting α' by α in Eq. 3.5:

$$pet_{\text{general}}(TS) = \beta \cdot TS + \alpha + TS. \tag{3.8}$$

The form of the final expression to evaluate the parallel execution time pet_{general} of the tasks is independent of the underlying scheduler.

The break even point (BEP) on a specific target platform is defined as the task size BEP that gives a speedup of 1:

$$su(BEP, BEP) = 1. \tag{3.9}$$

Finally, granularity for communication-free task graphs is defined as follows. A task is said to be small-grain if its associated work is smaller than the BEP. Conversely, it is considered large-grain if its work exceeds the BEP. A task graph is said to be small-grain if it contains any small-grain tasks.

3.3.2 Many Node Fork/Join Graphs

Fork/Join graphs with more than two spawned nodes as shown in Fig. 3.3 are handled as follows. The base overhead γ of parallel execution (as determined by the measured execution time of two empty tasks) is subtracted from the predicted two node execution time pet_{general} and divided by the sequential execution time to get the speedup of both tasks compared to the sequential execution. The square root of the combined speedup gives the speedup per task.

$$su_{\text{task}}(TS) = \sqrt{\frac{pet_{\text{general}}(TS) - \gamma}{TS}} \tag{3.10}$$

The final execution time for a fork/join graph with n tasks is given in Eq.3.11 by applying the speedup per task for every task and adding the base overhead:

$$pet^n(TS) = su_{\text{task}}(TS)^n \cdot TS + \gamma \tag{3.11}$$

The algorithm presented in Sec. 3.5 will decompose more complex task graphs into simple fork/join task graphs compatible to Eq.3.11 to predict their speedup. In order to apply this model, the target (scheduler and hardware) specific constants α and β as well as the base overhead γ must be measured.

In the next section, the quality of the model for two task predictions for several different target platforms is evaluated. In section 3.6, the model for many tasks is applied on a large range of randomly generated task graphs showing that Eq.3.11 can accurately predict parallel execution times.

3.4 Verification of the Two Task Cost Model

In order to verify that the model derived in Eq. 3.5 is sound, the execution time of fork/join task graphs as shown in Fig. 3.3 executed on several different hardware platforms with using the different scheduler implementations available for the **funkyImp** run-time system (see Chap. 7) are measured and compared to the model predictions.

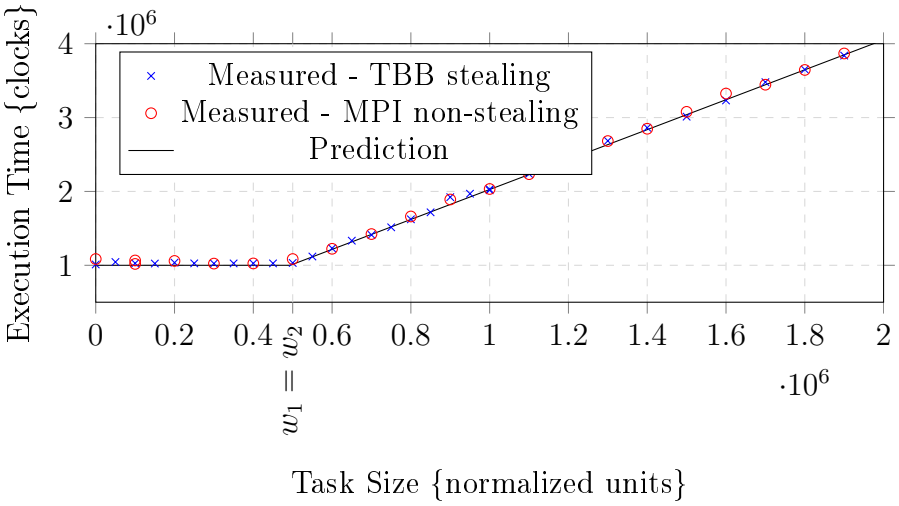

Figure 3.4: Comparison of the execution times of two parallel tasks of different size to the prediction that the overall run-time is dominated by the longer task. The first task's size is increased from 0 *to* $2 \cdot 10^6$ *while the size of the second task is fixed to* $0.5 \cdot 10^6$ *(and vice versa). Error bars have been omitted for readability, all measured data points lie within one standard deviation from the model prediction.*

Fig. 3.4 shows that the execution times of tasks of different lengths depend only on the longest task (assuming there are enough hardware resources to execute all tasks in parallel).

Fig. 3.5 shows that the model predictions are in very good agreement with the real behavior of the analyzed hardware and schedulers.

As to be expected, TBB's stealing-based scheduler performs better than the MPI based scheduler for smaller tasks. The BEP is about half the size for the stealing scheduler (2242 vs. 4636 clocks) on the Nehalem[1] platform. The difference is more pronounced on the mobile Sandy[2] platform (6812 vs. 22918 clocks). For the experiments, all MPI processes were placed on the same node, so that no actual network communication was executed.

The final values for α and β for each target platform (hardware and scheduler combination) are obtained by fitting the model from Eq. 3.6 to the measurements.

TBB's stealing scheduler and MPI's non-stealing task execution differ completely from a conceptual and implementation point of view. As shown in this section, they are both well represented by the model.

In addition, the benchmarks from Sec. 3.6 will show that Eq. 3.11 gives a realistic execution time prediction for fork/join graphs with more than two tasks.

In the next section, a preconditioning algorithm is presented that uses the model to merge tasks in a task graph that cannot be efficiently executed in parallel.

[1]Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz SMP x86_64 GNU/Linux 3.5.0-37-generic
[2]Intel(R) Core(TM) i7-3667U CPU @ 2.00GHz SMP x86_64 GNU/Linux 3.5.0-17-generic

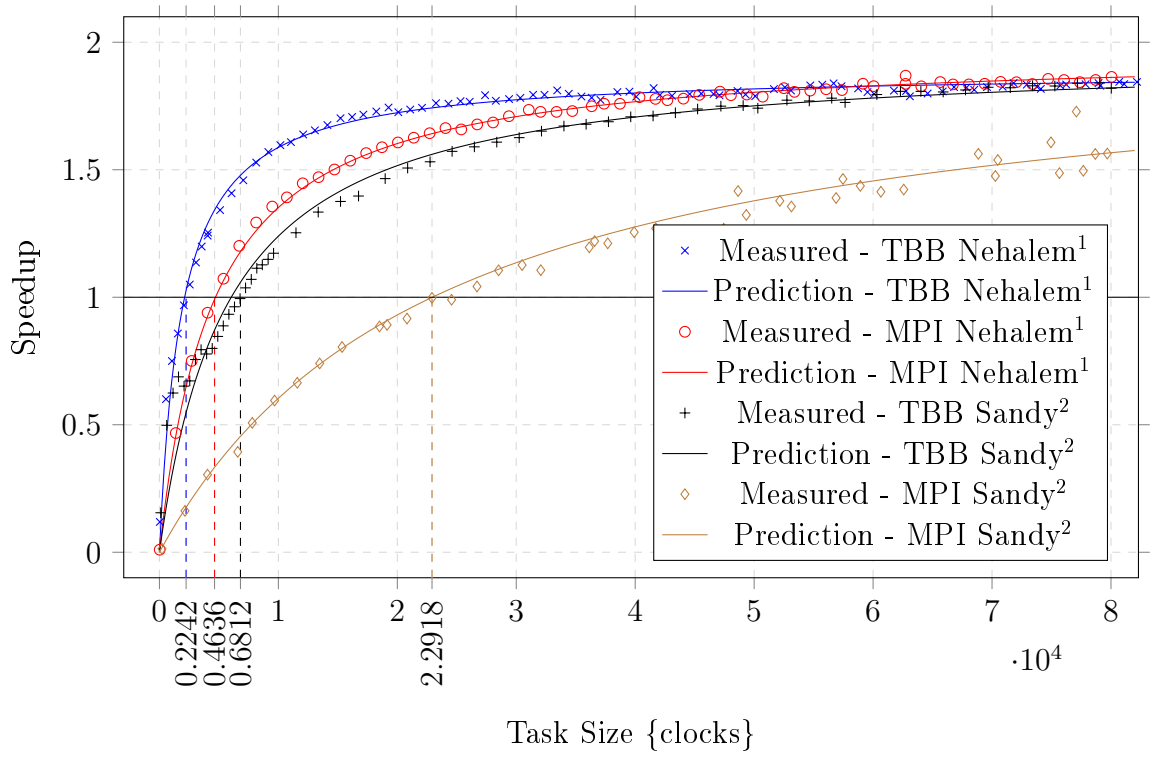

Figure 3.5: Comparison of the speedup of two parallel tasks with the predicted speedup from Eq. 3.6. Here, α and β are determined by fitting the model to the measured data. β is related to the signal probability per time and α quantifies the delay after the second thread has received the signal until it can start processing the second task. The experiment measures the average time it takes the scheduler to spawn, execute and join two completely independent tasks with equal task size. Error bars have been omitted for readability, all measured data points lie within one standard deviation from the model prediction and are in the order of 10% of the measured value. TBB denotes Thread Building Blocks 4.1 and MPI denotes open MPI 1.4.5. Speedstep was disabled on all systems during measurement to avoid large fluctuations due to the processors power management. The minimum task size required to realize 98% of the possible speedup for the these platforms reaches from 10^5 to 10^6 [clocks]. For larger tasks, the overhead may be neglected.

3.5 Scheduling Algorithm

Some previous scheduling algorithms attempt to give guarantees that the produced schedule length is no longer than the fully sequentialized version of the task graph [47] or that the schedule length is within a factor of two from the optimal schedule for task graphs where the computation to communication ratio is high [51]. As will be shown in Sec. 3.6, scheduling small-grain (w.r.t. task size) task graphs with scheduling algorithms that ignore scheduling overhead (even with perfect scheduling) can lead to surprisingly bad results where the real execution time of the task graph is orders of magnitude worse than the sequential execution.

This shows that the guarantee of the traditional scheduling algorithms is of theoretical nature. The actual execution times and speedups predicted by the traditional algorithms hold only in the limiting case when all tasks sizes are much larger than the BEP. Still, our algorithm improves the speedup even for such large tasks on average by 16% as shown in Sec. 3.6. Fig. 3.5 shows that close to linear speedup can be expected only for task sizes in the order of tens to hundreds of thousands of clocks or more. This means that the data ready times used to schedule tasks in many traditional algorithms [47] will be underestimated for many tasks and may shift tasks that appear to be parallel by different time offsets so that they will not run in parallel in practice.

In this section we present our clustering and execution time prediction algorithm which preconditions the task graph by collapsing parallel tasks that do not yield a minimum real-world speedup of ρ per core. The algorithm guarantees that the real-world execution time of the preconditioned task graph (on sufficiently many cores and if it actually contains any parallelism) is strictly smaller than the sequential execution time.

In principle, the algorithm decomposes input task graphs into instances of our modeled fork/join task graph from Fig. 3.3, applies our model via Eq.3.11 and composes the results. This is achieved by the following steps. Due to the Hasse property enforced on the graph, all predecessors of join nodes are parallel nodes. For parallel nodes, the algorithm finds the lowest common ancestor and the highest common descendant, which form a fork/join graph. The model is applied to these fork/join graphs to predict the speedup and merges parallel nodes if the speedup is insufficient. After merging the nodes, the Hasse property is restored. This last step is intuitively done in $O(n^4)$, but we present a more elaborate approach to get an upper bound of $O(n^3)$.

In the following, the algorithm is presented in several parts. The first part, as seen in Alg. 1, is the preprocessing part. Here we build several data structures to make sure that the overall upper bound of $O(n^3)$ is met. In the second part, see Alg. 2, the real work is done by calculating the estimated processing times of all tasks and merging appropriate tasks together.

The preprocessing shown in Alg. 1 computes for all predecessor pairs of join nodes the lowest common ancestor (LCA) with the minimal speedup, as well as the shortest paths from the LCA to the respective pair nodes in the unmerged graph. Later on, the corresponding paths inside the merged graph will be constructed from the paths from the LCA. Transitive edges are removed from the input graph by applying the Hasse reduction, as the algorithm assumes that tasks preceding a join node may execute in parallel which is not true for transitive edges. In addition, all linear task chains are removed from the graph. Both operations reduce the signaling overhead of the graph, as every edge in the graph can be interpreted as a signaling operation for the run-time scheduler.

Next, the outer loop of the actual algorithm is shown in Alg. 2. The nodes are visited in a topological order and data ready times are computed from the predecessors unless

Algorithm 1 preprocessing

Require: graph G
  $H \leftarrow \text{HasseReduction}(G)$
  $H' \leftarrow H$
  calculate APSP
  preprocessing for common ancestors
  create hashmap um from unmerged to merged nodes $um : \{\text{nodes}\} \to \mathcal{P}(\{\text{nodes}\})$
  initialize um: node $\mapsto$ {node}
  create hashmap pp^A from pairs of nodes to (lca, path1, path2) $pp^A : (\text{node}, \text{node}) \to (LCA, \text{path}, \text{path})$
  initialize pp^A:
  for all join nodes j' **do**
    for all predecessor pairs (i', k') of j' **do**
      get the LCA with minimal speedup: $lca' \leftarrow \text{getLCA}(i', k')$
      $p_{i'} \leftarrow \text{shortestPath}(lca', i')$
      $p_{k'} \leftarrow \text{shortestPath}(lca', k')$
      $pp^A \leftarrow pp^A + \left\{(i', k') \mapsto (lca', p_{i'}, p_{k'})\right\}$
    end for
  end for

the current node is a join node which needs the special treatment shown in the **merge** procedure in Alg. 5.

Algorithm 2 outer loop

  while traverse nodes j in topological order **do**
    mergeLinear (predecessor of j, j)
    if node j is a join node **then**
      merge j
      mergeLinear (predecessor of j, j)
    end if
    store estimated starting time est and estimated finishing time drt for current node
  end while

In the following, the merge operation $ik \leftarrow i \cup k$ means that a new node ik is created in the merged graph $\hat{H}$ that inherits all edges from i and k before these nodes are deleted from the graph.

mergeLinear is used in order to remove the overhead generated by the communication between consecutive tasks.

mergeParallel is used to merge the parallel predecessors of join nodes. If the indegree of the merged node is greater than one then the **merge** procedure is recursively applied to the merged node as it may be a newly created join node. Note that tasks sets which contain nodes with different control flow dependencies cannot be merged. in this case **mergeParallel** is not executed and the two sets remain unmerged.

The general **merge** procedure applied to the join nodes is shown in Alg. 5.

Here, variables with hat, like $\hat{H}$, denote information from the merged graph in its current state, whereas variables with prime, like i', denote unmerged information. Variables without hat or prime refer to information from the unmerged graph available from preprocessing. The information from the unmerged graph is translated into the domain

Algorithm 3 mergeLinear

Require: task sets i, k to be merged and k has exactly 1 predecessor
  $ik \leftarrow i \cup k$
  for all tasks t in ik **do**
    $um \leftarrow um + \{t \mapsto ik\}$
  end for
  remove Hasse violating edges
  update *nsp* distances and paths and *drt*

Algorithm 4 mergeParallel

Require: task sets i and k to be merged
  $ik \leftarrow i \cup k$
  for all tasks t in ik **do**
    $um \leftarrow um + \{t \mapsto ik\}$
  end for
  remove Hasse violating edges
  update *nsp* distances and paths and *drt* transitively
  if indegree $ik > 1$ **then**
    merge ik
  end if

of the merged graph using the *um* mapping.

In order to obtain realistic task execution times, the available parallel work (fork/join tasks preceding the join node) must be calculated. The algorithm performs this by considering the paths from the lowest common ancestor [14] for each pair of nodes preceding a join node. These paths may be considerably larger than the pair nodes alone. So, for each predecessor of a join node, the longest path from common ancestor to the predecessor node along with the path's start time is stored in the *tasklist*. The *tasklist* is passed to the multifit algorithm from Coffman, Garey and Johnson [32] to find a schedule for the parallel tasks preceding the current join node. The multifit algorithm uses a k-step binary search to find a schedule for n independent tasks in $k \cdot O(n \cdot log(n))$ with w.c. error bound $1.22 \cdot opt + \frac{1}{2^k}$. Experimental results for multifit indicate that the average error is in the order of 1.01 for $k = 7$, so slightly above optimal execution times are expected. If better heuristics than multifit are found to solve this specific scheduling problem then these can be plugged in instead. The schedule length returned by multifit must be corrected using the model from Eq. 3.11 if more than one core is used to obtain realistic data ready times.

Multifit is modified to not merge pairs of nodes where it has been already established that merging them is not effective. This information is stored in a hashmap.

The final while loop searches for the biggest number of cores which yields a speedup of at least ρ per core rather than minimizing the (parallel) execution time of the tasks in question. This avoids that a large number of cores is utilized to achieve small speedups (e.g. 100 cores for speedup 1.01). Obviously, the algorithm could be modified here to minimize the execution times. This might be desirable if the given task graph describes the complete program. Often, hierarchical task graphs are used [49] to represent complex programs so that one individual subgraph describes only part of a larger program that may run in parallel to the subgraph under consideration. In this situation or when energy efficiency is considered, optimizing for speedup per core yields better results as cores are used only if a minimum speedup can be achieved.

Algorithm 5 merge

Require: join node j
create list $tasklist$
$drt_{\min} = Infinity$
for all predecessors i of j in $\hat{H}$ **do**
for all predecessors $k \neq i$ of j in $\hat{H}$ **do**
unmerged nodes inside merged nodes
for all $i' \in i, k' \in k$ **and** $um(i') \neq um(k')$ **and** $pp^A(i', k')$ exists **do**
get the LCA and the corresponding paths from the unmerged graph H' : $(lca', p_{i'}, p_{k'}) \leftarrow pp^A(i', k')$
get the corresponding merged nodes and paths : $(est_i^p, est_k^p, \hat{p}_i, \hat{p}_k) \leftarrow um(lca', p_{i'}, p_{k'})$
$\hat{lca} \leftarrow$ update$(lca', est_i^p, est_k^p, \hat{p}_i, \hat{p}_k)$: $\hat{lca}$ is last common node in $\hat{p}_i, \hat{p}_k$ and all arguments of update are modified accordingly
if $work(\hat{p}_i) > maxwork_i$ **then**
get path's run-time (includes delays from nodes the path depends on) : $maxwork_i \leftarrow work(\hat{p}_i)$
$est_i \leftarrow est_i^p$
end if
calculate $drt_{\min}$ for common ancestors :
$drt_{\min} \leftarrow \min\{drt_{\min}, drt_{\hat{lca}}\}$
end for
end for
$tasklist \leftarrow tasklist + (i, maxwork_i, est_i)$
end for
start with maximal parallelism : cores = size of $tasklist$
$C \leftarrow$ **multifit**$(tasklist,$ cores$)$
while cores > 1 **and**
$(\sum_i maxwork_i)/($cores > 1 ? $pet^{cores}(C) : C) <$
$\max(1, \rho \cdot$ cores$)$ **do**
decrement cores
$C \leftarrow$ **multifit**$(tasklist,$ cores$)$
end while
put the unmerged tasks into bins according to **multifit**
update hashmap with all pairs of nodes that are in different bins
for all bins b **do**
mergeParallel (tasks in b)
end for
recalc $drt_{\min}$ for merged predecessors
handle overhead : $est \leftarrow drt_{\min} +$ cores > 1 ? $pet^{cores}(C) : C$
finish time of join node : $drt = est + work(j)$

Eventually, all tasks scheduled to the same core by multifit are merged using **mergeParallel** while maintaining that the graph is a Hasse diagram.

Both merge operations update the mapping um from unmerged to merged nodes and the DRTs of the merged node (and all nodes reachable from it) by adding the work from all nodes that were merged to the previous DRT. Moreover, both operations update the distances of the nearly shortest paths (NSP), as well as the paths themselves, which represent approximations of the shortest paths inside the merged graph after merging the nodes. They do so by iterating over all ancestors of the merged node, calculating their distances to it, and checking whether there is now a shorter path to a descendant of the merged node over a path that traverses the merged node. In order to retain the Hasse property, transitive edges are removed inside **mergeParallel**. Therefore, the NSPs are correct paths, but might not be the actual shortest paths in general.

As a side-effect, the algorithm calculates an execution time prediction for the complete task graph in $O(n^3)$. If only this prediction is desired the last while loop of **merge** in Alg. 5 and everything beyond that can be omitted.

Our algorithm correctly predicts that the task graph from Fig. 3.1 will be executed about 1.5 times faster on the specific target platform if all nodes are collapsed into a sequential program compared to the fully parallel program. Of course, the quality of the prediction is highly dependent on the quality of the (target specific) task size estimates. So far, the preconditioning algorithm assumes infinitely many cores. If the maximum parallelism in the result graph does not exceed the available cores of the target hardware then the result graph can be executed without further modifications. This may often be the case, as the preconditioning algorithm removes all inefficient parallelism and the number of available cores in modern hardware increases. If the graph contains too much parallelism after preconditioning, any traditional algorithm may be used to produce a schedule. If this algorithm uses data ready times then it must be modified to incorporate the realistic execution time prediction from Eq. 3.5 in order to produce realistic schedules.

Running a scheduling overhead corrected version of a traditional scheduling algorithm without the preconditioning algorithm is not sufficient to avoid bad schedules as these algorithms are not aware of the overhead and would treat it like useful computation.

As an alternative, when dealing with finitely many cores, the preconditioning algorithm may be turned into a complete scheduling algorithm. The algorithm is executed k times in order to find the ρ-speedup value which produces a task graph with as many or less tasks as the hardware supports. ρ is obtained by performing a k-step binary search with $\rho \in [0 \leq \rho_{\min} .. 1]$. This increases the execution time of the algorithm to $k \cdot O(n^3)$ and finds the optimal ρ value with an error of $\frac{1-\rho_{\min}}{2^k}$.

3.6 Experimental Results

We have generated 1000 random task graphs of varying complexity and a wide range of task sizes using the TGFF library from Dick, Rhodes and Wolf [40] to evaluate the preconditioning algorithm. All task graphs are preconditioned and their execution times averaged 250000 times on the Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz SMP x86_64 GNU/Linux 3.5.0-37-generic (hyper-threading and speedstep disabled) system. Eight task graphs were removed from the data set because they contained more than 4 parallel tasks after preconditioning and a precise measurement of their execution times was not possible on the 4 core system. The results are presented in Fig. 3.6.

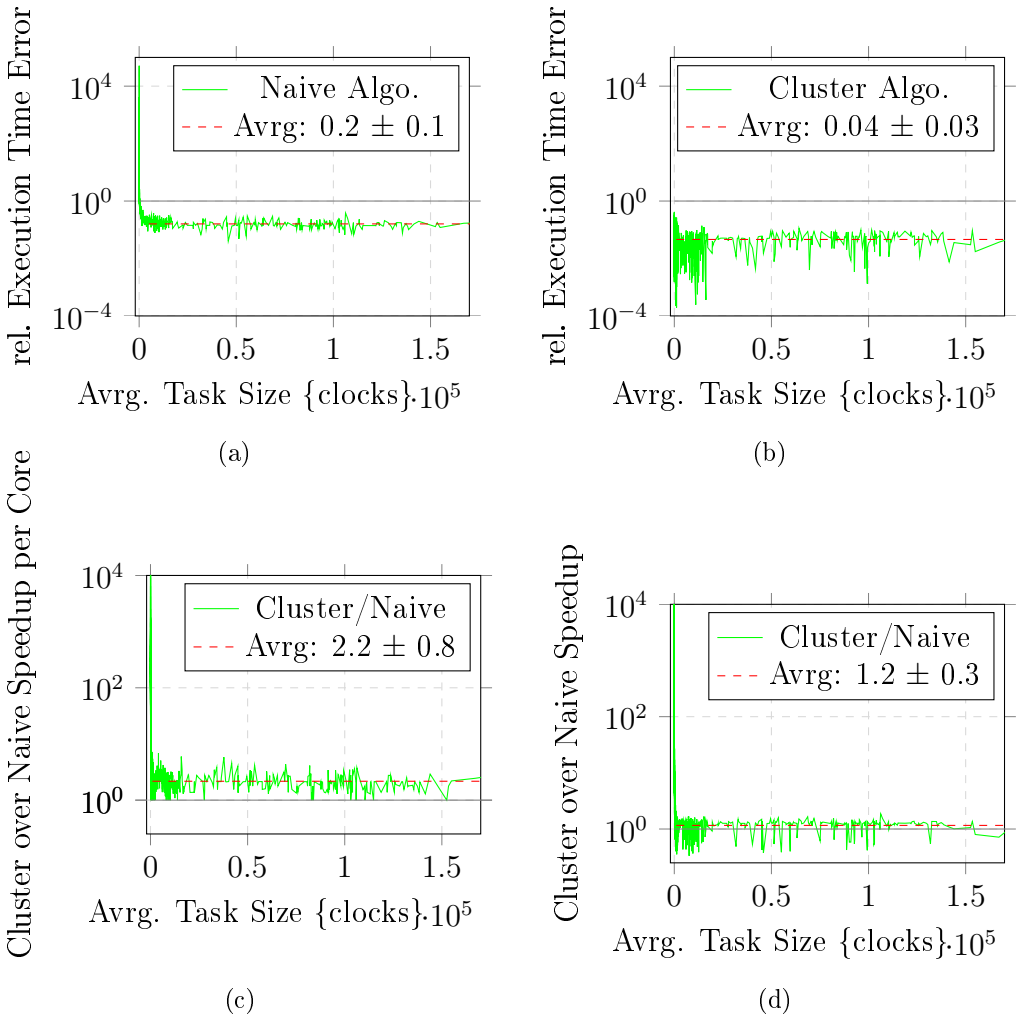

Figure 3.6: Experimental results comparing our clustering algorithm to naive scheduling algorithms that neglect run-time scheduling overhead (perfect scheduling). Averages were calculated for average task sizes bigger 1000 clocks so that they are not biased from the values for on average small task sizes where our algorithm outperforms the classical algorithm by several orders of magnitude.

The relative uncorrected error depicted in Fig. 3.6(a) shows how much the measured run-time of a given task graph deviated from the prediction generated using perfect scheduling (which neglects scheduling overhead). This error is relatively large with an average of 16% ± 14% for average task sizes bigger 5000 clocks (logarithmic scale). For smaller tasks the execution times are mispredicted by up to 10^7% showing that it is essential to take the overhead into account.

The relative corrected error depicted in Fig. 3.6(b) shows how much the measured run-time of a given task graph deviated from the prediction generated using our algorithm (which incorporates scheduling overhead). The average deviation is 4.6% with a standard deviation of 3.1% (for average task sizes bigger 1000 clocks). For extremely small tasks with an average task size near zero the error increases up to 40% (but is still many orders of magnitude smaller compared to perfect scheduling) for some graphs because the timer resolution on the test system is not good enough to create such small tasks more precisely.

Fig. 3.6(c) shows that the speedup per core achieved by our algorithm relative to perfect scheduling is $\geq$ one, so that our algorithm never creates a schedule with a speedup per core that is worse than the original schedule. For bigger tasks, the speedup per core is improved on average by 117%. For smaller tasks the improvement is much stronger because the dynamic scheduling overhead dominates the execution time.

Our algorithms optimizes for speedup per core rather than overall speedup. Nevertheless, Fig. 3.6(d) shows that on average the overall speedup of the task graphs is improved by 16% by our algorithm compared to perfect scheduling. Again, for smaller tasks the effect is much more pronounced. Therefore, it can be seen that speedup per core is a measure that does not generally lead to decreased overall speedup. In some specific instances, optimizing for speedup may yield slightly shorter execution times at the expense of utilizing many more cores (and highly reducing energy efficiency).

Our algorithm merges all parallel tasks of local fork/join sub-graphs until the desired speedup per core (and a local speedup > 1) is achieved. Globally, a task graph's critical path consists of sequences of fork/join sub-graphs. Inductively it follows that the task graph's sink node finishes before or at the same time as in the fully sequentialized version of the task graph so that the global speedup ≥ 1. Also, the speedup per core $\geq \rho$ as all parallelism that would violate this invariant is removed. This holds if α, β and γ from Eq. 3.11 are chosen so that all task execution times prediction $\leq$ real execution times. Otherwise, since there is an average error of $4.6\% \pm 3.1\%$ associated with the predicted task execution times, slight violations of these constraints are possible.

3.7 Related Work

Adve and Vernon [2] predict task graph execution times for a given scheduler model and complete program input data in $O(n + m)$. They present a system model where the scheduler and most other parts are modeled using queuing theory. For large task sizes their predictions are fairly good, results for small task sizes are not shown but would suffer from the lack of detailed scheduling overhead modeling. Their results are not applied to scheduling.

McCreary, Khan and Thompson [100] and Kwok [80] compare the makespan and Liu [93] compares worst case bounds of various scheduling heuristics but neglect the real-world execution times and overheads.

Most of the known scheduling heuristics have a complexity of $O(n^2)$ to $O(n^3)$ while operating on local information inside the task graph like edge weights and data ready times. Our algorithm is within the usual complexity of $O(n^3)$ while preprocessing allows us to examine a wider view of the parallelism inside the graph by considering the paths leading from lowest common ancestors via parallel nodes to the next join node (local fork/join sub-graphs).

Liou [91] suggests that clustering before scheduling is beneficial for the final result and McCreary and Gill [99] present a grain packing algorithm. This algorithm is limited to linear and pure fork/join parallelism, more complex graphs are not accounted for in detail and scheduling overhead is neglected. Many other clustering algorithms that do not take scheduling overhead into account have been proposed [90, 147].

Power efficient scheduling has been investigated by [127] and others, taking into account special hardware features to run specific tasks slower and with lower power consumption or better thermal footprint with minimal impact on the makespan. Our algorithm guarantees a minimum core utilization efficiency, so that additional cores are used only

if a user defined speedup per core can be achieved. This allows otherwise inefficiently used cores to be turned off completely and can be combined with other power saving techniques.

To the best of our knowledge, none of the previous scheduling algorithms consider scheduling overhead, so in contrast to our algorithm no guarantees on real-world speedup and core utilization efficiency can be given.

3.8 Conclusion and Outlook

We have shown that task graphs which contain tasks with sizes in the order of 10^5 clocks and higher are not realistically scheduled by traditional scheduling algorithms as the scheduling overhead is neglected. We derived a realistic model for the scheduling overhead of symmetric schedulers and presented a task graph clustering algorithm which unlike previous scheduling algorithms guarantees a real-world speedup and core utilization efficiency. Generally, our algorithm provides a vastly more accurate execution time model compared to existing algorithms and improves the speedup per core in most cases while never making it worse. The effect is viable for large task sizes with improved speedup by 16% and improved speedup per core by 117%. For smaller tasks we improve existing methods by several orders of magnitude. Furthermore, we have shown that the scheduling overhead predictions should be incorporated into the existing scheduling algorithms to obtain realistic data ready times.

Using common APIs like MPI [108] for networked distributed computing, it is trivial to generate code that runs on such networks from a **funkyImp** program using the task graph representation. The only difference to shared memory parallel computing is that the communication costs to transfer data between different node must be taken into account when scheduling node from the task graph onto different nodes in the network. As we have shown in this chapter, the existing algorithms (which include communication) neglect the dynamic scheduling overhead and produce unrealistic makespans. So in order to achieve good static schedules that include communication overhead, the algorithm presented in this chapter will need to be extended to include realistic network communication costs in the future. On a shared memory system, a scheduling algorithm that incorporates communication costs can also be used to optimize cache locality of tasks by assigning a cost to accessing data that was computed on one CPU and is then used on another CPU.

Chapter 4

Data-Parallelism

In this chapter we discuss the integration of data-parallelism into the **funkyImp** language. Writing high-performance numeric code that is correct on variably-sized input data is challenging. Moreover, the need to exploit the computational power of current multi-core and vector architectures adds an additional angle of complexity that is only insufficiently addressed by existing loop-parallelizing compilers. This chapter presents the data-parallel sub-language of **funkyImp**, a pure functional language that allows for a concise specification of array computations. Specifically, new arrays are created by a set of computations, each defining one part of the new array. The language's key novelty is that the defined indices of an array are specified by a set of polytopes that are themselves parameterized by program variables. This generalizes the view of an array being a hypercube of values to programmer-centric partitionings of the index set such as rows or upper/lower triangles. We introduce language features called filters and lenses that allow for a declarative handling of these index sets, so that complex data-parallel algorithms can be expressed that formerly required an imperative formulation. The easy-of-use of our language is complemented by a novel type system that employs the power of off-the-shelf polyhedral solvers to statically ensure the absence of out-of-bound accesses and that every array element in a new array is defined by exactly one computation. As a result, the computations that construct a new array are independent so that a type correct program can be translated to highly efficient, parallel C++ code. Indeed, our experimental work suggests that **funkyImp** is on par with hand-written parallel C++ code on a variety of key benchmark algorithms.

The remainder of the chapter is structured as follows. First, we give some intuition into declarative array composition in Sect. 4.1. Afterwards, in Sect. 4.2 we define our language and introduce array index sets formally. In Sect. 4.3 we illustrate our type system by discussing the type checking process on `swapK''`, the formal type system is presented in Sec. 4.5. In Sect. 4.4 we discuss the effect of array subsets as first class citizens on writing generic and efficient code. In Sect. 4.7 we give details for the implementation of our compiler and run-time system. An experimental comparison with SAC, Repa, Pluto, C++ and MKL is given in Sect. 4.8. Related work is discussed in Sect. 4.9 before Sect. 4.10 concludes our presentation.

4.1 Declarative Array Composition

Pure, functional languages lend themselves to automatic parallelization of programs due to the absence of side-effects that are hard to reason about. A particular challenge is

expressing data-parallelism where the same operation is applied to many values simultaneously. Data-parallel operations are most naturally expressed using arrays. While functional language allow for an elegant construction and deconstruction of algebraic data structures, no good mechanisms exists for constructing and de-constructing arrays in a declarative, safe and yet efficient manner. For instance, a matrix is often stored as an array of row-vectors but it is unclear how a column-vector or trace can be extracted from this representation without manual computation and copying of the corresponding elements. In contrast to arrays, lists allow for elegant de- and reconstruction. The following Haskell program illustrates this:

```
swapK k xs = drop k xs ++ take k xs
```

The `swapK` function extracts the first k elements and moves them to the end of the vector, returning a vector of the same length as the input `xs`. While the algorithm is elegant and concise, a data-parallel translation is hindered by the underlying linked list data structure which requires that elements are accessed in sequence. A corresponding Haskell array function can be defined as follows:

```
swapK' k xs = let (l,u) = bounds xs; n=u-l in
  ixmap (0,n) (\i ->if i<=n-k then i+k else i-(n-k+1)) xs
```

This map operation can be executed in parallel. However, the explicit index manipulation is non trivial to construct, neither declarative, nor safe for $k < 0$, nor is it efficient due to evaluating an **if**-expression for each index.

In this work, we present a functional language that addresses these shortcomings. The key idea is to provide language constructs, namely *filters* and *lenses*, so that most data-parallel problems can be decomposed into independent computations without copying data and resorting to error prone, manual index computations. Our language allows to declaratively specify `swapK` on arrays by stating that the last $n - k$ elements of the input should be placed into the first $n - k$ indices of the output (the part computed by `drop`) and that the first k elements of the input should be placed into the last k elements of the output (the part computed by `take`).

For the sake of illustrating filters and lenses, arrays can be thought of as a set of tuples $\langle i, p\rangle$ where i is an index and p is an l-value (pointer to the array cell). Applying a filter will remove all tuples whose index violates the conditions of the filter from the set. Lenses apply a transformation on the index of each tuple. Note that the l-value of each tuple is not affected by either filter nor lens operation.

The two parts of the declarative `swapK` computation are illustrated in Fig. 4.1. We consider the diagram from the input and output arrays at the top and bottom border, towards the center. At the top of the figure, the input array is partitioned at k using filters `take` and `drop`, giving two arrays called *sources*, that is, two sets of index/l-value tuples. At the bottom, the output array is partitioned at $n - k$ using the same filters, yielding two *targets*. We now consider the computation moving from the upper left source to the lower right target. Here, the indices of the source and target are translated to a common index set using a lens each, yielding the *transl. src* and *transl. tgt* sets in the figure. The computation of an output element is performed as follows. For each tuple $\langle i, p\rangle$ in the translated target, the value $f(i)$ is written to the address p where f is a user-supplied function. In the example, $f(i)$ is defined by the value at p' where $\langle i, p'\rangle$ is a tuple in the source. The net effect is that the first k elements of the input array are written to the last k indices of the output array. Analogously, the computation on the right transfers the $n - k$ last elements from the input to the beginning of the output array.

Although the problem was decomposed into two independent computations, the output is a single array as indicated in the figure. By using a newly allocated array as the set of target tuples, multiple independent computations can be composed without resorting to mutable arrays or copying of arrays. This is particularly relevant to our pure language since all elements of an array must be defined at construction time.

We now show how the example is implemented using our language which follows the schema in Fig. 4.1. Note that our language separates the indices from the l-values of arrays so that filter and lens operations are defined on index sets only. The following code therefore defines a set `d1` of one-dimensional indices (i) and two filters `take` and `drop` that define subsets of `d1`. The lenses define maps between pairs of the given index sets. Further details and the `swapK` function are detailed below.

```
1  dom d1{n}={ (i) : 0<=i<n }
2
3  dom take{n}(k)={(i) in d1{n} : i<k}
4  dom drop{n}(k)={(i) in d1{n} : i>=k}
5
6  lens (i) in take{n}(k) -> (i) in d1{min(k,n)}
7  lens (i) in drop{n}(k) -> (i-max(k,0)) in d1{min(n-k,n)}
8
9  swapK'' : int -> int[d1{n}] -> int[d1{n':n'=n}]
10 swapK'' k xs =
11 array d1{n} (d1{n}.take(n-k)->d1,\(i)->xs.drop(k)->d1[i])
12             (d1{n}.drop(n-k)->d1,\(i)->xs.take(k)->d1[i])
```

Line 1 defines the index set `d1{n}` which is parameterized by the size n and contains one-dimensional indices $[0, n-1]$. A filter is constructed by adding constraints to a *parent* index set. For instance, lines 3 and 4 define filters that restrict the elements of parent set

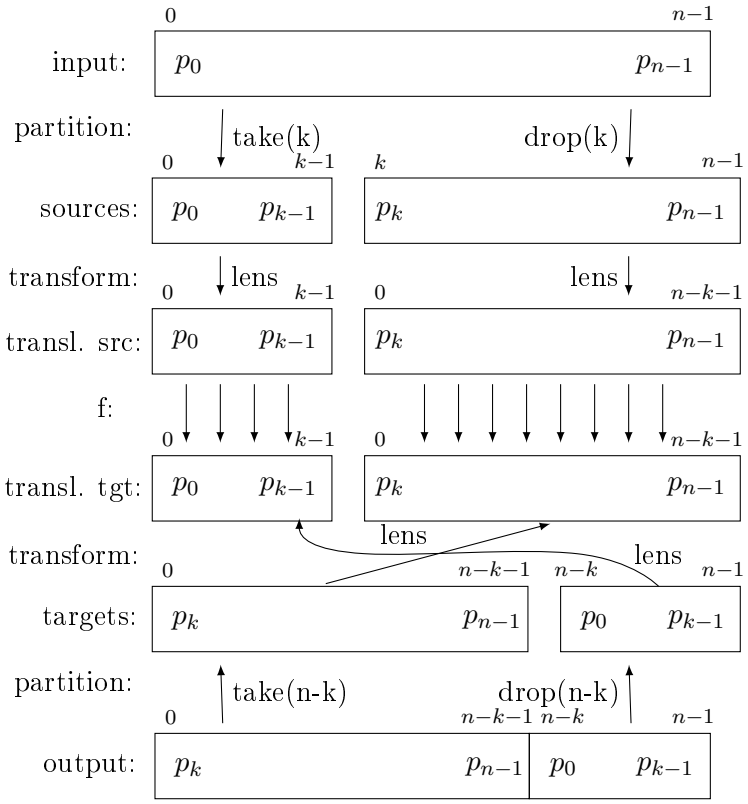

Figure 4.1: Decomposing, transforming and re-composing of arrays.

`d1{n}`. Specifically, besides the size n, they take the additional parameter k, denoting the cut-off point and remove indices above resp. below this cut-off point. A filter f is applied to an index set s by writing $s.f$. The filter parameters in curly parenthesis are inferred by matching the parent set of the filter f with the index set s; the parameters in round brackets must be given explicitly. For instance, `d1{n}.take{n}(k)` can be written as `d1{n}.take(k)`. The index set can also be taken from the type of a variable. For instance, `xs.drop(k)` denotes the index set $[k, n-1]$ if xs is an array over the index set $d1\{n\}$ (written `xs : int[d1{n}]`).

A lens declaration defines a bijection between two index sets. For instance, line 7 shows a lens definition that relates an index (i) in `drop{n}(k)` with an index (j) in `d1{min(n-k,n)}`, so that $i - max(k, 0) = j$. A common use of lenses is to shift the indices of array elements to a common index set.

A lens l is applied to an index set s by writing $s \to l$. For instance, $d1\{n\}.drop(k) \to d1$ applies the transformation from *drop* to $d1$ defined in line 7.

Line 9-12 define `swapK''` analogous to Fig. 4.1. The type annotation is necessary in order to communicate the various parameters used in the filter and lens operations. Here, the type `int[d1{n}]` is an integer array that is defined at indices $(i) \in d1\{n\}$. The result of the function is computed using a built-in function `array` that creates a new immutable array by performing the set of computations that are given as arguments. Each computation is a tuple defining the target index set and a function that defines the corresponding elements. The type of `array` can intuitively be written as

$$D_r \to (D_1 \times (int^{\mathsf{dim}(D_1)} \to \alpha)) \to \ldots \to (D_n \times (int^{\mathsf{dim}(D_n)} \to \alpha)) \to \alpha[D_r],$$

where $int^n = int \times .. \times int$ and $\mathsf{dim}(D_i)$ denotes the dimension of the indices represented by D_i. So `array` outputs an array with element type α that is defined at all result indices D_r where the elements of D_r are defined by the computations given in the tuples. For the example, the target indices D_i corresponds to the "translated target" in Fig. 4.1. Analogously, the functions inside the computations are λ terms that access the ith element of in the translated source. This completes the discussion of the example.

The lack of a good deconstruction mechanism for arrays in current functional and imperative languages has the following consequence. There exists no generic way to reference subsets of array elements (i.e. to extract a row, column or upper triangle from a matrix) so that they can be efficiently passed into a function without copying the elements or manually constructing the subset inside the function. In our language, in order to pass a subset of array elements into a function, we pass only the pointer to the underlying array and the filter or domain type associated with the data restricts the referenced elements to the desired subset (i.e. the i-th row). This allows efficient and compositional usage of array element subsets which is not possible with existing language constructs. It can be argued that the `ixmap` construct in `swapK'` uses linear constraints like our filters in the guards of the if-expression to apply different operations on different subsets of elements but this method is neither compositional as the subsets used to build the result array cannot be constructed independently of the `ixmap` expression nor can it be used to extract a subset and pass it to another function.

In addition to enabling compositional usage of sub arrays, the array construction in our language can be translated into efficient, parallel code without computing the actual index sets explicitly. The key insight is that each composition of filters and lenses translates to a single linear index computation that can be computed statically. Moreover, the **`array`** -expression itself translates to a sequence of simple loops over the target indices that can also be computed statically.

Another fundamental aspect of our language is to statically prove the safety of the declarative **array**-expressions, even in the presence of dynamic-sized multi-dimensional arrays and dynamic parameters (e.g. the size parameter `k` in `swapK''`). We present a type checker that relies on our language constructs to verify that:

- array accesses are in bounds
- target index sets used to specify an array neither over- nor under-specify the result array
- user defined lenses are linear bijections

Our language features a universe of types where the values of integer variables are characterized by disjunction of polytopes. The type checker is sound and complete for programs where all integer variables that are used in types are computed using linear expressions. We show how type checking reduces to a subset test between the index sets defined by two types. This test can be computed symbolically using off-the-shelf solvers [139]. Counter examples can be generated for all type errors.

Our declarative implementation `swapK''` is safe because all array constructions and accesses are statically proven to be in bounds. In addition, our implementation is efficient because the data-parallel operations contain no **if** expressions and dynamic out-of-bounds checks are not required, yielding 30% to 100% performance improvement over a version with bounds checks and **if** expressions.

To summarize, we propose a novel language, that

- introduces sub arrays as first class citizens
- allows de- and recomposition of data-parallel operations using filters and lenses that so far required an imperative formulation
- validates index transformations from the programmer
- offers strong static safety guarantees for **array**-expressions using parametric sets of polytopes
- generates multi-threaded, data-parallel code that is competitive with hand written parallel C/C++ code.

4.2 Data-Parallel Language

In this section our **funkyImp** language is defined. Our language is based on a strict, first order, pure functional language with explicit type checking. The grammar for the language is shown in Fig. 4.2. We define the necessary nomenclature and detail those grammar rules that are relevant to array indexing.

Array Index Domains. Generally, a multi-dimensional array can be understood as a partial function mapping a vector of integers (the index vector) to values. The domain of this function is a set that contains all index vectors for which this partial function is defined. Henceforth, we call the set of index vectors for which the array is defined the *index domain*.

Most languages, such as C, Pascal, Java, and Haskell, only allow index domains that are defined by a set of index vectors $\vec{I}$ that lie inside a hypercube $\{\vec{I} \in \mathbb{Z}^m | L_0 \leq I_0 <$

$$
\begin{aligned}
\textbf{program} &::= (\textbf{dom-def}|\textbf{dom-lens}|\textbf{fun-def})^*\textbf{exp}\\
\textbf{dom-def} &::= \mathit{dom}\ \textbf{decl} = \{<\textbf{id-list}>(\mathit{in}\ \textbf{decl})? : \textbf{cons}\}\\
\textbf{decl} &::= \mathit{tid}\{\textbf{id-list}\}(\textbf{id-list})\\
\textbf{id-list} &::= (\mathit{id}(,\mathit{id})^*)?\\
\textbf{cons} &::= \textbf{lin-eq}((\mathit{and}|\mathit{or})\ \textbf{lin-eq})^*|(\textbf{cons})\\
\textbf{lin-eq} &::= (\exists\ \mathit{id}:)^*\mathit{not}?\textbf{lin-exp}((< \mid <= \mid =)\textbf{lin-exp})?\\
\textbf{lin-exp} &::= \textbf{lin}(+\textbf{lin}|-\textbf{lin})^*|(\textbf{lin-exp})\\
\textbf{lin} &::= (\mathit{const}*)?\mathit{id}|\mathit{const}\\
\textbf{dom-lens} &::= \mathit{lens}\ (\textbf{id-list})\ \mathit{in}\ \textbf{dom-inst}\ \text{->}(\textbf{exp-list})\\
&\quad\ \mathit{in}\ \textbf{dom-exp}\\
\textbf{exp-list} &::= (\textbf{exp}(,\textbf{exp})^*)?\\
\textbf{id-cons} &::= (\mathit{id}(:\textbf{cons})?(,\mathit{id}(:\textbf{cons})?)^*)?\\
\textbf{dom-inst} &::= \mathit{tid}\{\textbf{id-cons}\}((\textbf{id-cons}))?\\
\textbf{dom-exp} &::= \mathit{tid}\{(\textbf{exp}(,\textbf{exp})^*)?\}\\
\textbf{fun-type} &::= \mathit{id} : \textbf{type}(\rightarrow \textbf{type})+\\
\textbf{fun-def} &::= \mathit{id}\ \textbf{id-list} = (\textbf{def}\ \mathit{in})^*\textbf{exp}\\
\textbf{array} &::= \mathit{array}\ \textbf{exp}\ ((\textbf{exp},\textbf{lambda}))+\\
\textbf{reduce} &::= \mathit{reduce}\ \textbf{exp}\ \textbf{exp}\ \textbf{lambda})\\
\textbf{lambda} &::= \backslash(\textbf{id-list}) \rightarrow \textbf{exp}\\
\textbf{def} &::= (\mathit{let}\ \mathit{id}\textbf{:type} = \textbf{exp})\\
\textbf{type} &::= \textbf{dom-inst}([\mathit{extends}?\ \textbf{dom-inst}])?\\
\textbf{exp} &::= \mathit{id}\ \textbf{exp}^*|(\mathit{id}|\textbf{dom-exp})\ \textbf{dom-sel}?|(\textbf{exp})|\textbf{array}\\
&\quad\ |\mathit{if}(\textbf{exp})\ \textbf{exp}\ \mathit{else}\ \textbf{exp}|\textbf{exp}\ \mathit{op}\ \textbf{exp}|\textbf{reduce}\\
\textbf{dom-sel} &::= ((\textbf{.}|\text{->})\mathit{tid}(\textbf{id-list}))^*([\textbf{exp-list}])?
\end{aligned}
$$

Figure 4.2: Strict first order language with polytope bounded array indices that support array and reduce operations.

$U_0 \wedge .. \wedge L_m \leq I_m < U_m\}$ where the vectors $\vec{L}$ and $\vec{U}$ define the boundaries of the hypercube. Our language generalizes the admissible shapes of index domains from hypercubes to parametric polytopes.

We now formally define these novel index domains together with some elementary operations that allow us to manipulate them. Afterwards, we discuss the grammar rules and how they give rise to the corresponding index domains.

Parametric Polytope Domains A parametric polytope domain, henceforth domain, is a λ-expression that maps a vector of parameters $\vec{X}$ onto a set of indices $\vec{I}$ that are constrained by a boolean formula f over linear (in)equalities as defined by the grammar rule **cons**. Thus, every domain is represented by the following syntactic structure:

$$
\begin{aligned}
&\mathit{dom} = \mathbb{Z}^n \rightarrow \mathcal{P}(\mathbb{Z}^m) \\
&\mathit{dom} \ni \lambda(\vec{X})\,.\,\{\vec{I} \in \mathbb{Z}^m | f\}
\end{aligned}
\tag{4.1}
$$

Although a domain is written as a function, it is never evaluated but the syntactic structure is used uninterpreted. Note that the names of parameters $\vec{X}$ and of indices $\vec{I}$ are relevant for the manipulation of domains.

We define the following operations on vectors. The dimension k in a vector $\vec{T} \triangleq \langle T_1, .., T_k \rangle$ is denoted as $|\vec{T}|$. The binary operator $\vec{X} \oplus \vec{Y} \triangleq \left\langle X_1, .., X_{|\vec{X}|}, Y_1, .., Y_{|\vec{Y}|} \right\rangle$ concatenates the the components of the two given vectors.

We define the empty domain as $D^0 \triangleq \lambda().\{\langle\rangle \,|true\}$, mapping the empty parameter vector to a set containing the empty vector.

For the following definitions, assume that D is given by $D = \lambda(\vec{X}).\{\vec{I} \in \mathbb{Z}^m | f\}$.

Let $D\{\}$ denote the parameter vector $\vec{X}$ of D and let $D\,\langle\rangle$ denote the index vector $\vec{I}$. Let $D\{X\}$ denote a copy of domain D where a new parameter named X is appended to the parameter vector $D\{\}$ unless a parameter with the same name exists already i.e. $D\{X\}\{\} = D\{\} \oplus \langle X \rangle$ if $X \notin D\{\}$ and $D\{X\}\{\} = D\{\}$ otherwise. The operation $D\{X\}$ naturally lifts to vectors $\vec{X}$, written $D\{\vec{X}\}$.

Let $D\,\langle I \rangle$ denote a copy of domain D where a new index named I is appended to the index vector $D\,\langle\rangle$. This operation lifts to vectors $\vec{I}$ by applying the append operation consecutively on the vector elements from left to right.

Our type system (and the solver we are using) can handle only linear relations between variables, so that all non-linear (in)equalities must be removed from a formula before it is placed inside a domain as f. To this end, let $D[\![]\!]$ denote the formula f in domain D and let $D[\![f']\!]$ denote a copy of D where the formula $f = D[\![]\!]$ is replaced by f *and* f'' where f'' is syntactically derived from f' as follows: all constraints involving non-linear expressions are replaced by *true* and all variables in f' that are neither existentially quantified in f' nor exist in $D\{\}$ are added as domain parameter. Note that operations `max(a,b)`, `min(a,b)` and `a mod b` can be expressed using linear constraints and existentially quantified variables and are therefor not removed.

Let $D(V \to V')$ denote a copy of D where variable V is renamed in to V'. We lift this operation to vectors of variables $\vec{V}$ and $\vec{V'}$ whenever $n = |\vec{V}| = |\vec{V'}|$ by defining $D(\vec{V} \to \vec{V'}) \triangleq D(V_1 \to N_1)..(V_n \to N_n)(N_1 \to V_1')..(N_n \to V_n')$ where N_i are fresh variables. The renaming via $\vec{N}$ ensures that the operation is well-defined even if $\vec{V}$ and $\vec{V'}$ have variables in common.

Finally, let $D(\vec{X})$ denote function application that results in the syntactic index set $\{\vec{I} \in \mathbb{Z}^m | f\}$. We now use these syntactic transformation to explain how program constructs translate to (parametric) index domains.

4.2.1 Grammar Rules

We now present the most relevant language constructs. We assume that all program variables are unique.

Domain Declarations A domain declaration (rule **dom-def** in Fig. 4.2) of the form **dom** `domain{x_1,...,x_l}={(i_1,...,i_m): f }` gives rise to the index domain $D^0\{x_1, ..., x_l\}\,\langle i_1, ..., i_m \rangle\,[\![f]\!]$ which evaluates to $\lambda(\vec{x})\,.\,\{\vec{i} \in \mathbb{Z}^m | f\}$ since the grammar ensures that f is linear.

Domain Instantiation A domain constructed from a declaration, as defined above, constitutes a *template* for a domain because the parameters $x_1, \ldots, x_l$ are undefined. A domain template is instantiated by renaming the parameters to program values and adding constraints which restrict the parameters in relation to other program variables and constants, thereby creating a domain instance.

By instantiating domains with program variables, the domain constraints can create relations between these program variables. For instance, the type system may check that the index set represented by `d1{N}` is a subset of the index set represented by `d1{M}`, which is meaningful only if a relation such as $N = M - 1$ exists.

Filter Declarations A filter is a domain that adds additional constraints to an existing domain (called *parent* domain) so that it represents a subset of the parent domain. A filter allows to introduce new parameters (called *filter* parameters). The values for these parameters are supplied by the programmer when applying the filter to the parent domain, so that the subset defined by a filter declaration can be parametric in the filter parameters.

For example, given the domain declaration **dom** `d2{x,y}()= { (j,k): 0 <= j < x` **and** `0 <= k < y }`, the domain `row` defined by **dom** `row{x,y}(j)={(a,b)`**in** `d2{x,y}:a=j}` requires that the first index from the *parent* domain `d2` is equal to `j`. It defines a filter that is parametric over a filter parameter `j` which allows to address the 2-d indices that correspond to a specific row of the parent domain `d2` which itself contains the index set of a 2-d matrix.

A filter is applied to the parent domain using the `.` operator (rule **dom-sel**) followed by the filter name and a list of filter parameters.

For instance, given a 2-d array `m : int[d2{5,5}]`, the computation **let** `r:int[row{5,5}]=m.row(2)`**in** `e` evaluates `r` to the third row of the five by five matrix `m`.

Given a filter declaration **dom** `filter{x_1,...,x_l} (l_1,...l_n)={(i_1,...,i_m)`**in** `parent{p_1,...,p_k}: f }` we construct the a corresponding index domain $parent\{l_1, ..., l_n\}[\![f]\!](\vec{x} \to parent\{\})(\vec{i} \to parent\,\langle\rangle)$. For brevity, we only consider the case where $\vec{x} = \vec{p}$. Here, the constraints f from the filter domain are added to the parent constraints. Afterwards, the parameter and indices are translated, so that f uses the names of the parent domain.

In the following, we write $\vec{X} \oplus \vec{F} = D\{\}$ to extract the domain parameters $\vec{X}$ and the filter parameters $\vec{F}$.

The parent domain of a filter D is denoted as $\mathsf{parent}(D)$. Since filters are themselves domains, a hierarchy of domains can be constructed by defining filters that have another filter as parent.

While filters can create a hierarchy of index domains it is useful to move within or across hierarchies. For example, the geometric shape of the index domain **dom** `inner{x,y}()= {(a,b)`**in** `d2{x,y}:a>0` **and** `b>0` **and** `a<x-1` **and** `b<y-1}` is identical to the shape of a `d2{x-2,y-2}` index domain. Yet, it is not possible to apply filters defined for the `d2` domain on the `inner` domain. We now present *lenses* that allow to view the rectangular index domain with lower-left corner $(1, 1)$ of `inner{x,y}` as a `d2{x-2,y-2}` domain, thereby making all filters available for `inner` which are defined on `d2`.

Lenses A lens applies a coordinate transformation in order to provide a different view of an index domain. Note that a lens specifies a bijection between two existing index domains and, thus, does not define a new domain.

A lens is defined using the rule **dom-lens**. A declaration `lens (i_1,..,i_n)`**in** `dom1 -> (e_1,..,e_n)`**in** `dom2` specifies that index $\vec{i}$ of the first index domain `dom1` maps to the vector given by evaluating the expressions e_i that must be functions of $\vec{i}$ and any parameter of `dom1`. Type checking will ensure that the indices computed by e_i are indeed in the index domain `dom2`.

For example, the target domain `d1{N}.drop(N-k)->d1` of the array application `array d1{N} (d1{N}.drop(N-k)->d1,\(l)-> xs.take(k)->d1[l])` from the introduction uses the lens from `drop` to `d1` to transform the index set $[k, .., N - 1]$ to

$[0, .., N-k-1]$.

Array/Reduce Our language allows to declaratively define arrays by combining disjunct domains using the built-in array function, that, intuitively, can be given the following type:

$$array : D_r \to (D_1 \times (int^{\mathsf{dim}(D_1)} \to \alpha_1)) \to \ldots \to (D_n \times (int^{\mathsf{dim}(D_n)} \to \alpha_n)) \to (\bigsqcup_i \alpha_i)[D_r].$$

The **`array`**-expression computes a new array. Its arguments are the result domain D_r and a set of tuples where each tuple (d, f) contains a domain d and a function f mapping indices from d to values. The values of the new array are computed by applying the function f on all indices represented by the domain d for every tuple (d, f), thereby combining the results of all computations defined by the tuples into a single new array.

The type system verifies that the pairwise meet of the domains defining the target indices of each computation is empty and that the join of the domains defining the target indices is equal to the result domain D_r, where the join/meet of two domains D_1 and D_2 is defined as

$$D_1{}^{\sqcup}/_{\sqcap} D_2 \triangleq \lambda(\vec{X} \oplus \vec{X'}) \to \{\vec{v} | \vec{v} \in D_1(\vec{X})^{\vee}/_{\wedge} \vec{v} \in D_2(\vec{X'})\}.$$

In addition, our language features a built-in reduce operation which has the following type:

$$reduce : \alpha[D_I] \to \alpha[D_R] \to (int^n \to \alpha[D_R] \to \alpha[D_R]) \to \alpha[D_R]$$

Here, a reduction over the domain D_I with the second argument as initial value is performed using the user supplied function which takes an index vector of dimension $n = \mathsf{dim}(D_I)$ and the current accumulator value with type $\alpha[D_R]$ as input. The order of the reduction is unspecified.

If If expressions (rule **exp**) may create additional constraints on domain parameters from the guard that the type system assumes to hold within the respective branch. Note that the type system only collects the linear parts of the guard.

Integer Domains In our language, integer values are defined over the predefined domain **`dom`** `int{l,u}={(i): l<=i<=u }` so that the type `int{a,b}` describes an int value that is within interval $[a, b]$. The type `int` is short for `int{-infty,infty}`.

Integers can be either explicitly defined values or can be introduced implicitly by matching them to domain parameters in **`let`**-expressions and function applications. For example, the parameter `X` in line 16 of Fig. 4.3 is matched to the parameter of the **`array`** -expression in line 17 so that `X=x`.

Domain Hierarchies Domain and filter declarations allow to define hierarchies of index sets. Given an array over a domain, the programmer can navigate one step downwards in the domain hierarchy by applying filters that decompose the array. **`array`**-expressions allow upwards navigation in the hierarchy by composing domains. Lenses are introduced to allow navigation across domain hierarchy levels as well as between different hierarchies. In this way, lenses allow to map domains with identical cardinality from different hierarchies into a shared domain before applying a function. The net effect is that the programmer can express computations in a simple, shared domain rather than manually translating between two complex domains.

In addition, lenses allow to write generic code in the sense that the programmer can define functions that accept all array types which can be lens transformed into the type the

function operates on. For example, the function *dot* in Sec. 4.6 can efficiently compute the dot product for any input array types which can be transformed into $d_1\{n\}$ by applying the corresponding lens. Previously, writing such generic code required inefficiently copying the vectors from their underlying source array into dedicated one-dimensional arrays. To avoid copying the data, a tedious definition of a specific version of *dot* for any possible combination of input types was required.

4.3 Type Checking

```
1  dom d1{n}={ (i) : 0<=i<n }
2
3  dom take{n}(k)={(i) in d1{n} : i<k}
4  dom drop{n}(k)={(i) in d1{n} : i>=k}
5
6  lens (i) in take{n}(k) -> (i) in d1{min(k,n)}
7  lens (i) in drop{n}(k) -> (i-max(k,0)) in d1{min(n-k,n)}
8
9  swapK'':int->int{A,B}[d1{N}]->int{U:U=A,V:V=B}[d1{M:M=N}]
10 swapK'' K xs =
11 array d1{N} (d1{N}.take(N-K)->d1,\(i)->xs.drop(K)->d1[i])
12             (d1{N}.drop(N-K)->d1,\(i)->xs.take(K)->d1[i])
13
14 main : int->int{A:A=0,B:B=4} =
15 main x =
16   let m:int{0,9}[d1{X}] =
17     array d1{x} (d1{x},\(i,j)-> max(0,min(i+j,9))) in
18     if(X>9) ((swapK'' 4 m)[m[X/2]])/3 else 4 in
```

Figure 4.3: Type Checking swapK"

For the sake of brevity, this section presents the type checking process on the running example in Fig. 4.3. We postpone the formal presentation of the typing rules and their soundness proof to Sec. 4.5.

First, domain and filter declarations are type checked. As they determine the number of elements that the **array**-expression allocates, finiteness of a domain/filter is checked. For instance, the definition of `drop` in line 4 has the internal representation $\lambda(\langle n, k\rangle).\{\langle i\rangle \mid 0 \leq i \text{ and } i < n \text{ and } i \geq k\}$, following the construction in Sec. 4.2.1. We interpret this term as a polytope with parameters n, k, and use the method of [11] to compute the number of its integral points, giving the following cardinality:

$$|\lambda(\langle n,k\rangle).\{\langle i\rangle \mid \ldots\}| = \begin{cases} n-k: & 0<k<n \\ n: & n \geq 1 \wedge k \leq 0 \\ 0: & \text{else} \end{cases}$$

In case there exists a parameter combination for which the cardinality is infinite, the domain is rejected.

The second step is to check lens declarations. A lens declaration $\vec{i} \in D_s \to t(\vec{i}) \in D_t$ defines a transformation t from source domain D_s to a target domain D_t that must be reversible. Let $t(D)$ denote the domain D after applying t to its index set. Intuitively, we check that $t(D_s) = D_t$ and that t^{-1} exists so that $t^{-1}(t(\vec{i})) = \vec{i}$. In the example, we

rename the index to i' and add $i' = t(i) = (i - max(k, 0))$ to D_s =`drop{n}(k)` which yields $\lambda(\langle n, k\rangle).\{\langle i'\rangle \mid \exists i : i' = i - max(k, 0)$ and $0 \leq i$ and $i < n$ and $i \geq k\}$. This index domain simplifies to $\lambda(\langle n, k\rangle).\{\langle i'\rangle \mid 0 < i' < min(n - k, n)\}$, which corresponds to D_t =`d1{min(n-k,n)}`. This equivalence is checked using the ISL library [138]. We now check that t^{-1} exists by inverting a matrix M that represents the lens transformation, that is, the index transformation t and the mapping of parameters. Since piece-wise linear sub-expressions (e.g. *max* and *min* expressions) cannot be represented by a single matrix, we make these values available for the matrix by adding them as filter parameters to the input vector. The transformation of these extra parameters is the identity. In case a non-linear expression e is not a sub-expression, the matrix has co-linear rows (since a parameter and e map to e) and is not invertible. In this case we add $e + n$ instead of e as additional parameter. In the example, we add $\kappa_1 = min(n - k, n) + n$, $\kappa_2 = max(k, 0)$ where the $+n$ is due to $min(n - k, n)$ being not a sub-expression.

The matrix M transforms the parameters and indices from `drop` to `d1` so that $M \cdot \langle i, n, k, \kappa_1, \kappa_2\rangle^T = \langle i - max(k, 0), min(n - k, n), k, \kappa_1, \kappa_2\rangle^T$ holds:

$$M \triangleq \begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

If the left-inverse M^{-1} exists, the lens transformation is bijective. Let N (resp. N^{-1}) denote the first $\mathsf{dim}(D_t)$ (resp. $\mathsf{dim}(D_s)$) rows of M (resp. M^{-1}). Now, $t(i) = N \cdot \langle i, n, k, \kappa_1, \kappa_2\rangle^T = i - max(k, 0)$ and $t^{-1}(i) = N^{-1} \cdot \langle i, min(n - k, n), k, \kappa_1, \kappa_2\rangle^T = i + max(k, 0)$. Note that $t^{-1}(t(\vec{i})) = \vec{i}$ holds only if t is linear in the indices $\vec{i}$. Therefore, we reject lenses that use indices in piece-wise linear functions so that linearity of t in $\vec{i}$ follows.

Finally, the functions can be type checked. The type of `swapK''` is parametric in `N`, `A` and `B` and a call to `swapK''` guarantees the constraints on the output type. The definition of `swapK''` in lines 9–12 is type correct if the body of `swapK''`, namely an **`array`**-expression, evaluates to an array over indices `d1{M:M=N}` over `int{U:U=A,V:V=B}` as required by the return type. In order to check this, we first discuss the type-checking of the **`array`**-expression.

Recall that the computations inside an **`array`**-expression are tuples of an index domain and a λ-expression. The λ-expression is evaluated for every index in the index domain. In order to type check the first tuple in line 11, we first compute the index domain defined by `d1{N}.take(N-K)->d1` before checking that all accesses within the λ-expression are in bounds.

The definition in line 3 states that `take` can only be applied to a domain `d1`, which is satisfied here since it is applied to `d1{N}`. The result of `d1{N}.take(N-K)` is `take{N}(N-K)`. The next step is to check the lens application `take{N}(N-K)->d1`. Line 6 defines a matching lens from `take` to `d1`. By type checking this lens earlier on, we have shown that for any parameters n and k `take{n}(k)` is transformed to `d1{min(n-k,n)}`. Thus, the lens application `take{N}(N-K)->d1` reduces to `d1{min(N,N-K)}` by substituting n and k with N and $N - K$.

Given that `d1{min(N,N-K)}` defines the indices that the λ-expression `\(i)->xs.drop(k)->d1[i]` is applied to in line 11, we can now check that the accesses within the expression are in bounds.

The only access `xs.drop(k)->d1[i]` is in bounds if the domain of the index i (given by `d1{min(N,N-K)}`) is a subset of the index domain of `xs.drop(k)->d1`. We replace xs in this term by the index domain `d1{N}` of its declared type `int{A,B}[d1{N}]`. The resulting type `d1{N}.drop(k)->d1` reduces to the index domain `d1{min(N,N-K)}`.

We use the ISL solver to check that the domain of the index i is a subset of the latter. In the example, this is trivially the case since the index domains are identical. In general, given $\lambda(\vec{X}).A \subseteq \lambda(\vec{Y}).B$, the solver checks if the inclusion holds for all $\vec{X}$, $\vec{Y}$, that is, $\forall \vec{x} \in \mathbb{Z}^{|\vec{X}|} : \forall \vec{y} \in \mathbb{Z}^{|\vec{Y}|} : \lambda(\vec{x}) \subseteq \lambda(\vec{y})$.

Finally, we must check that the elements defined by the tuples are non-over-lapping and cover the complete result domain `d1{N}` of the **array**-expression which guarantees that every element is defined exactly once.

Note that the domain `d1{min(N,N-K)}` computed from the first tuple component in line 11 does not generally denote the indices that are written since lenses provide transparent views and thus, the effect of the lenses used to construct `d1{min(N,N-K)}` must be undone before writing. Therefore, we apply the inverse t^{-1} of the lens transformation from `take` to `d1` to the indices of `d1{min(N,N-K)}`. In the example, t^{-1} is the identity so that the written indices for the first tuple is given by the same domain `d1{min(N,N-K)}`. Let $Tgt_1 = \lambda(\langle N, K\rangle).\{\langle i\rangle \mid 0 \leq i < min(N, N-K)\}$ denote the internal representation of this domain.

This concludes the type checking of the first tuple. Type checking the second tuple proceeds analogously. The domain for index i is `d1{min(K,N)}`. The domain of the array access `xs.take(K)->d1[i]` evaluates to `d1{min(K,N)}`, so that these accesses are also in bounds. In order to compute the written indices, we apply the inverse lens transformation from `d1` to `drop`, namely $t^{-1}(i) = (i + max(k, 0))$, to `d1{min(K,N)}`. The result is $\lambda(\langle N, K\rangle).\{\langle i'\rangle \mid \exists i : \exists k : i' = i + max(k, 0)$ and $0 \leq i$ and $i < min(K, N)$ and $k = N - K\}$ which reduces to $\lambda(\langle N, K\rangle).\{\langle i\rangle \mid max(N-K, 0) \leq i < min(K, N)+max(N-K, 0)\}$. Let Tgt_2 denote this index domain.

In order to check that each element is specified at least once, we use the ISL solver to check that the join of all target domains is equal to the domain `d1{N}` of the result array, i.e. $Tgt_1 \sqcup Tgt_2 = \lambda(\langle N\rangle).\{\langle i\rangle \mid 0 \leq i < N\}$. In order to check that no element is defined more than once, $Tgt_1 \sqcap Tgt_2 = \emptyset$ must hold. In general, for an **array**-expression with more than two computations, the pairwise meet must be empty and the join of all Tgt_i domains must be equal to the target domain.

Note that even for our simple example, three cases are considered: For $K < 0$ the test $Tgt_1 \sqcup / \sqcap Tgt_2$ simplifies to $\lambda(\langle N, K\rangle).\{\langle i\rangle \mid 0 \leq i < N\} \sqcup / \sqcap \lambda(\langle N, K\rangle).\emptyset$. For $0 \leq K \leq N$ the test simplifies to
$\lambda(\langle N, K\rangle).\{\langle i\rangle \mid 0 \leq i < N - K\} \sqcup / \sqcap \lambda(\langle N, K\rangle).\{\langle i\rangle \mid N - K \leq i < N\}$. For $N < K$ the test simplifies to $\lambda(\langle N, K\rangle).\emptyset \sqcup / \sqcap \lambda(\langle N, K\rangle).\{\langle i\rangle \mid 0 \leq i < N\}$. Thus, in all cases, the meet is empty and the join of the domains is equal to `d1{N}`.

Moreover, the same three cases must be considered when translating the **array**-expression to code. Manually writing code that correctly and efficiently handles the case distinction and computes the correct indices for each case is error prone. Our language uses the Cloog library [13] that, given a domain d and a function $f(\vec{i})$, generates loops and conditionals that have the semantics $(\vec{X}) \rightarrow \forall\, \vec{i} \in d(\vec{X}) : f(\vec{i})$ and thus automatically handles all cases correctly. Furthermore, efficient code can be generated for read accesses defined through lenses and filters thereby alleviating the user from deriving complex index transformations. Note that several lens transformations are combined into a single transformation, as shown in Sec. 4.6.6.

For the example, the following C++ code is generated (slightly simplified, vectorization and parallelization omitted):

```
for (i=0;i<min(n,n-k);i++) {
  res[i]=xs[i+max(k,0)];
}
for (i=max(0,n-k);i<n;i++) {
  res[i]=xs[i-max(k,0)];
}
```

In order to complete the type checking of the **array**-expression, the value domain is checked. This is computed as the join of the result of all λ-expressions. For linear expressions, the type is computed by structural induction over the expression. Non-linear expressions are over-approximated by `int`, that is, the unconstrained integer type. For instance, `xs.take(K)->d1[i]` has the value domain of xs which is $\lambda(\langle A, B\rangle).\{\langle v\rangle \mid A \leq v < B\}$. Note that this deduction is always valid because we have shown that all accesses are in bounds.

In summary, the type of the **array**-expression is given by its value domain $\lambda(\langle A, B\rangle).\{\langle res\rangle \mid A \leq res < B\}$ and its index domain $\lambda(\langle N\rangle).\{\langle i\rangle \mid 0 \leq i < N\}$. It remains to check that the type of the **array**-expression satisfies the annotated return type of `swapK''`. Whenever an expression is type checked against a type annotation (which is the case for let-expression, function application and function bodies), three properties are checked: First, the value domain of the expression must be a subset of the value domain of the annotation. Second, the domain template of the expression's index domain must match that of the annotation. This ensures that both domains use the same parameters and can be implemented with the same data structure. Third, the index domain of the expression must be a subset of the annotation's index domain. Since these properties are satisfied here, `swapK''` is type correct.

Now, we type check the `main` function whose declaration on line 14 requires to show that it maps two integers with arbitrary values to an integer in $[0, 3]$.

The local variable `m` is declared to be an array over integers in $[0, 9]$. The index domain of `m` has a parameter `X` that is bound based on the type of the right-hand side expression. Here, the **array**-expression in line 17 uses an instance of domain template `d1` with parameter `x` which thereby provides the value for `X=x`. In order to prove that the value of each array element lies in the interval $[0, 9]$, we compute the type of the expression `max(0,min(i+j,9))` to `int{0,9}` which satisfies the value type of `m`.

Now consider line 16 of `main`. For each **if**-statement, the type system creates a program variable in each branch with a type that reflects the information in the guard, effectively collecting relational constraints in the type environment. In the example, the value domain of this variable is $\lambda\langle X\rangle.\{\langle\rangle \in \mathbb{Z}^0 \mid X > 9\}$ for the then-branch and $\lambda\langle X\rangle.\{\langle\rangle \in \mathbb{Z}^0 \mid \neg(X > 9)\}$ for the else-branch. The constraints of this variable are collected to refine the type of `m` to `int{0,9}[d1{X:X>9}]` when it is passed to `swapK''`. The call is proven correct by checking that `4` is a subtype of `int` (which is true because $[4, 4] \subseteq [-\infty, \infty]$) and the type of `m` is a subtype of `int[d1{X}]`. The latter is satisfied since for the value domain $[0, 9] \subseteq [-\infty, \infty]$ and `d1{X:X>9}` describes a subset of `d1{X}` and the domain template `d1` is identical.

Given that the result type of `swapK''` is the same as the input type, `swapK'' 4 m` has type `int{0,9}[d1{X:X>9}]`. Thus, the access $[X/2]$ is in bounds if $\langle \frac{X}{2}\rangle \in$ `d1{X:X>9}`. Inserting the definition of `d1` results in the tautology $0 \leq \frac{X}{2} < X \wedge X > 9$, thus proving the access to be safe. The accessed value (in $[0, 9]$) is divided by 3, so that the value of the then-branch in line 18 is in $[0, 3]$.

Finally, the result from the then-branch is joined with the result of the else-branch giving $[0,3] \cup [4,4] = [0,4]$ which is a subtype of result type `int{A:A=0,B:B=4}` of `main`. This concludes the informal description of type checking.

4.4 Generic and Efficient Array Code

In this section we will show how sub arrays as first class citizens enable the programmer to write array code that is generic and efficient at the same time. Consider the C++ function `dot` in Figure 4.4 which performs a scalar product between two vectors. The function is generic in the sense that it can be used to compute the scalar product of any two vectors of length `TSIZE` but it is not efficient because in order to compute the scalar product of a matrix row and a matrix column the row and the column must be copied into a flat vector structure (unless the elements happen to be stored continuously in memory).

```
template<int TSIZE>
float dot(float r1[TSIZE],float r2[TSIZE])
{
  float accum=0;
  for(int i=0;i<TSIZE;i++)
    accum+=r1[i]*r2[i];
  return accum;
}

//scalar product of row and column
float dotRowCol(float *r1,float *r2,int row, int col,int size)
{
  float accum=0;
  for(int i=0;i<size;i++)
    accum+=r1[row*size+i]*r2[i*size+col];
  return accum;
}
```

Figure 4.4: Either generic or efficient implementation of a scalar product.

In contrast, the C++ function `dotRowCol` in Figure 4.4 does not require to copy elements, given that the pointers `r1` and `r2` point to a matrices stored as arrays in row major form. The code inside the function explicitly iterates over the correct portions of the matrices. Nevertheless, the efficient implementation of the scalar product requires that the function is reimplemented for every possible data layout and type of object the scalar product may be applied to because the data layout must be taken into account explicitly, thus generating a lot of code duplication which makes maintaining and bug fixing problematic. In addition, the required index transformations inside the function obfuscate the simple computation and accidentally passing a data structure with incorrect data layout into the function is not detected by the type system.

The only way to factor out the index computation from `dot` is to have the programmer define and pass functions into `dot` which map a one dimensional index into the respective array offsets. Finding efficient versions of such functions is actively being researched [79] and non-trivial for higher dimensional arrays with possibly non-convex domain shapes that may depend on dynamic parameters. Offloading this complex task on the programmer seems unreasonable. Furthermore, our experiments show that a function based generic

scalar product implemented in C++ using templates and lambdas produces much slower code with current compilers than a specialized version like `dotRowCol` in Fig. 4.4.

By using sub arrays as first class citizens, our language allows to implement a version of the scalar product that is generic, efficient and type safe at the same time. Line 1 in Fig. 4.5 shows the type of our scalar product. Instead of specifying the concrete domain type (e.g. `d1{X}`), **funkyImp** allows to specify that any sub-type of domain d is acceptable by using *extends* d as domain. A domain d_1 is a sub-type of a domain d_2 iff a lens transformation from d_1 to d_2 exists or $d_1 = d_2$.

```
dot: T[extends d1{X}]->T[extends d1{Y:Y=X}])->T
dot va vb = reduce (va->d1) 0 (\(i,acc)->acc+va->d1[i]*vb->d1[i])
```

Figure 4.5: Generic and efficient implementation of a scalar product.

Values `va` and `vb` in line 2 cannot be used directly (as their concrete type is unknown). Instead, the available lens transformation to `d1{X}` is applied to access the elements of the two input vectors which are guaranteed to have the same length by the constraints on `X` and `Y`, making the function safe. The function is generic, because any array with domain type that can be transformed to `d1{X}` can be used for `va` and `vb`, irrespective of the data layout in memory. For example, `va` or `vb` could be a row, column, trace of a matrix or a plain vector just as long as a lens transformation from the array subset to `d1{X}` is defined. Figure 4.7 shows how our generic scalar product can be applied in a plain and a cache blocked matrix multiply. Our definition of the scalar product is efficient, because our `dot` template is instantiated for every combination of domain types (and fixed domain sizes) the function is applied on, so that each instance performs the required iterations inside the array without copying any values. For example, the application of `dot` in line 26 of Fig. 4.7 generates the same code as function `dotRowCol` in Fig. 4.4 because the `dot` function is applied on a `row` and a `col` array subset and the matrix is stored internally in row major order. In line 32 `dot` is applied to a row of a block, which generates the code shown in Fig. 4.6 line 5.

Function `mm` in line 26 defines a plain matrix multiplication computing the scalar product of column and row vectors.

Function `blocked` in line 36 computes the matrix product of a matrix and a transposed matrix using one level cache blocking. It decomposes the matrix into blocks of size 32^2 and computes the matrix product for each block by slicing the matrix into 32 wide stripes in the first dimension and reducing over all 32 wide slices in the second dimension of each stripe (again yielding 32^2 blocks). Function `mulAddBlock` in line 30 performs the scalar product of all rows in two blocks. The >> operator in line 38 computes the domain that contains all possible indices $\langle i, j \rangle$ that yield a non-empty result when applying the filter `ma.block3232(i,j)`.

A simplified version of the C++ code generated by our compiler for `mm`, `mulAddBlock` and `blocked` is shown in Fig. 4.6.

Function `semi_mmT` uses a different decomposition than `blocked` to compute the matrix product of a matrix and a transposed matrix. The reduction is applied to full rows (rather than blocks) but the result is computed in groups of 32^2 blocks, which yielded the best results.

```
1  const dim_t dim = 4096;
2  const dim_t block_width = 32;
3  const dim_t num_blocks = dim / block_width;
4
5  float dotBlock(float *r1,float *r2,int start)
6  {
7   float accum=0;
8   for(int i=0;i<block_width;i++)
9    accum+=r1[start+i]*r2[start+i];
10  return accum;
11 }
12
13 float* mm(float* first, float* second)
14 {
15  float sum = 0;
16  float* multiply=
17   (float*)malloc(sizeof(float)*dim*dim);
18 #pragma omp parallel for //auto for simple mm
19  for ( int c = 0 ; c < m ; c++ )
20   for ( int d = 0 ; d < q ; d++ )
21   {
22    for ( int k = 0 ; k < p ; k++ )
23     sum = sum + first[c*dim+k]*second[k*dim+d];
24    multiply[c*dim+d] = sum;
25    sum = 0;
26   }
27  return multiply;
28 }
29
30 void mulAddBlock(float *out,float *a,float *b
31  ,int br,int bc, int col)
32 {
33  for(dim_t r=0;r<block_width;r++)
34   for(dim_t c=0;c<block_width;c++)
35   {
36    out[(br*block_width+r)*dim+bc*block_width+c]
37    +=dotBlock(&a[(br*block_width+r)*dim],
38    &b[(bc*block_width+c)*dim],col*block_width);
39   }
40 }
41
42 void blocked(float *X,float *Y)
43 {
44  float *Z=(float*)malloc(sizeof(float)*dim*dim);
45 #pragma omp parallel for //not auto detected
46  for (dim_t br = 0; br < num_blocks; br++)
47   for (dim_t bc = 0; bc < num_blocks; bc++)
48    for (dim_t col = 0; col < num_blocks; col++)
49     mulAddBlock(Z,X,Y,br,bc,col);
50  for ( int c = 0 ; c < dim ; c++ )
51   printf("trace[%d]=%f\n",c,Z[c*dim+c]);
52 }
```

Figure 4.6: C++ version of plain and fully cache blocked matrix multiply.

```
(*define array index domains:*)
dom d1{x} = { (i) : i < x } (*i>=0 is implicit*)
(*following domains are compatible to d1*)
dom d2{x,y} = { (j,k) : j < x and k < y }
(*following domains are projections from d2*)
dom row{x,y}(j) = { d2{x,y}(a,b) : a=j}
dom col{x,y}(j) = { d2{x,y}(a,b) : b=j}
dom trace{x,y} = { d2{x,y}(a,b) : a=b and x=y }

(*domains to slice d2:*)
dom block3232{x,y}(i,j) ={ d2{x,y}(a,b)
: a>=i*32 and a<i*32+32 and b>=j*32 and b<j*32+32 }
dom slicex32{x,y}(j) ={ d2{x,y}(a,b) : a>=j*32 and a<j*32+32 }
dom slicey32{x,y}(i) ={ d2{x,y}(a,b) : b>=i*32 and b<i*32+32 }
(*lenses, with clauses are checked when lens is applied*)
lens (i) in d1{x} <- (j,i) in row{x,y}(j:0<=j and j<x)
lens (i) in d1{y} <- (i,j) in col{x,y}(j:0<=j and j<y)
lens (i) in d1{x} <- (i,i) in trace{x,y}
lens (i,j) in d2{32,32} <- (32*br+i,32*bc+j)
in block3232{x,y}(br:0<=br and br<x%32,bc:0<=bc and bc<y%32)
lens (i,j) in d2{32,y} <- (32*br+i,j) in slicex32{x,y}(br:0<=br and
br<x%32)
lens (i,j) in d2{x,32} <- (i,32*bc+j) in slicey32{x,y}(bc:0<=bc and
bc<y%32)

(*simple mat mult:*)
mm T[d2{X,Y}]->T[d2{Y,X}])->T[d2{X,X}]
mm ma mb=array d2{X,X}(\(r,c)->dot(ma.row(r))(mb.col(c)))

(*blocked mat mult helper:*)
mulAddBlock T[d2{X,Y}]->T[d2{X,Y}]->T[d2{X,Y}]->T[d2{X,Y}]
mulAddBlock tmp ra rb T = (*reduction over block*)
array tmp (tmp->d2,
\(r,c)->tmp->d2[r,c]+dot (ra->d2.row(r)) (rb->d2.row(c)))

(*blocked mat*mat^T mult:*)
blocked T[d2{X:X%32=0,X}]->T[d2{X,X}]->T[d2{X,X}]
blocked ma mb =
 let res:T[d2{X,X}]= array d2{X,X}
 (ma>>block3232, (*for every 32*32 block in ma:*)
 \(br,bc)->
  reduce (ma.slice32(br)->d2.slicex32) 0
  \(col,accom)->mulAddBlock accum (ma.block3232(br,col))
     (mb.block3232(bc,col))
 )

semi_mmT:int[d2{X:X%32=0,X}]->int[d2{X,X}])->int[d2{X,X}])
semi_mmT ma mb =
 array ma
 (
  (ma>>block3232), (\(br,bc)-> array d2{32,32}
  (
   ma.block3232(br,bc)->d2,
   \(r,c)->dot(ma.row(br*32+r),mb.row(bc*32+c))
  ))
 )
```

Figure 4.7: Plain, semi and fully cache blocked matrix multiply.

Note that the lenses in lines 16-22 are declared backwards using <- because lenses must always be specified so that the arguments of the higher dimensional domain are a function of the arguments of the lower dimensional domain to keep the transformation invertible. Furthermore, these lenses carry constraints on the filter parameters which must hold when the lens is applied. For example, the row of a matrix is empty unless the row index is in the valid range. These constraints are verified whenever such a lens with constrained parameters is applied (e.g. in line 26 $0 \leq r < X$ must hold so that the lens $d2 \rightarrow row$ can be applied).

In line 36 of Figure 4.7 we show the one level fully cache blocked matrix multiply and in line 49 the semi blocked implementation which were used to generate matrix multiplication benchmarks presented in the next section. For comparison, we show equivalent C++ implementations of the plain matrix multiply `mm` and the blocked version `blocked` in C++ in Fig. 4.6.

An additional version of the scalar product for blocked rows (`dotBlock` in line 5) is required. When comparing the C++ version of `mulAddBlock` in line 30 with the **funkyImp** version in Fig. 4.7 it is apparent that the lack of sub arrays as first class citizens forces the programmer to manually implement unsafe index transformations which obfuscate the actual meaning of the code in order to achieve good performance. In contrast, the **funkyImp** code is very concise (especially when taking into account that the domain and lens definitions in line 1-23 of Fig. 4.7 are reusable declarations and can be part of the standard library), a lot safer and as fast.

4.5 Type System

In this section we present our type system. Although it provides no type inference, the task of type checking is still non-trivial in the sense that types track which array elements are accessible by describing the defined indices using a disjunction of polyhedra. For brevity, we restrict the presentation to integers and arrays of integers.

4.5.1 Universe of Types

The type of an expression is written as $\alpha[D]$ with the C-like interpretation of an array with element type α with valid indices defined by the index domain $D \in dom$.

For our presentation, we restrict the element types to integers. In this setting, we can define $\alpha \in dom$ as another domain representing the possible values of the elements in the array. For example, the type $\lambda(u, v).\{\langle i \rangle \in \mathbb{Z} | l \leq i \leq u\}[D]$ describes an array where every element of the array is in the interval $[l, u]$. Integer elements are especially interesting when used as indices into arrays, as shown in Fig. 4.3 in line 32.

For brevity, we have omitted the handling of integer overflow. Nevertheless, our type system and the soundness and completeness proof can be easily modified to model overflows correctly.

The type of an integer value (rather than an array of values) is given by $\alpha[D^0]$, where the index domain is the set containing the 0-tuple. Let $\bot\alpha \triangleq \lambda().\emptyset$ denote the empty value domain.

Fig. 4.8 shows the type checking rules for our type system. Here, Γ is the type environment mapping identifiers to types.

All domains constructed by the type system are constructed syntactically using the operations defined in Sec. 4.2. We use the isl [138] constraint solver to perform symbolic

subset tests between the index sets represented by different syntactic domains so that the actual index set represented by a syntactic domain from *dom* is never computed explicitly. In addition, we define $\mathsf{env}_\Gamma : dom \to dom$ which enriches a given domain with the relational information from the type environment. This function is applied to both domains before applying the subset test to include the relational information.

The function env constructs a domain which enriches the domain D by all relational information transitively related to the domain D available in the environment

$$\mathsf{env}_\Gamma(D) \triangleq D[\![\bigotimes_{\{v \in id | x \in \Gamma(v)_1\{\} \wedge x \in \mathsf{env}_\Gamma(D)\{\}\}} v]\!]. \tag{4.2}$$

where $v_1 \otimes v_2 \triangleq \Gamma(v_1)_1[\![]\!]$ *and* $\Gamma(v_2)_1[\![]\!]$. env transitively adds the formulas of all value types that share parameters with $\mathsf{env}_\Gamma(D)$.

4.5.2 Subset Relation

The subset test verifies that the index set represented by D_1 enriched by the relational information available about the parameters of D_2 is a subset of the index set represented by D_2 for all possible parameters values

$D_1 \sqsubseteq_\Gamma D_2 \triangleq \forall \vec{x} \in \mathbb{Z}^n : \forall \vec{x'} \in \mathbb{Z}^m : \mathsf{env}_\Gamma(D_1\{D_2\{\}\})(\vec{x}) \subseteq \mathsf{env}_\Gamma(D_2)(\vec{x'})$ where $n = |(D_1\{D_2\{\}\})\{\}|$ and $m = |D_2\{\}|$. Adding the parameters from D_2 to D_1 causes $\mathsf{env}_\Gamma(D_1\{D_2\{\}\})$ to add all the constraints from the environment that are transitively related to the parameters of D_2.

Finally, we define a subset operation $\precapprox_\Gamma$ which verifies that D_1 is smaller or equal to D_2 and that the domain templates of both domains are equal: $D_1 \precapprox_\Gamma D_2 \triangleq D_1 \sqsubseteq_\Gamma D_2 \wedge \Gamma(\mathsf{name}(D_1)) =_\Gamma \Gamma(\mathsf{name}(D_2))$ where $D_1 =_\Gamma D_2 \triangleq D_1 \sqsubseteq_\Gamma D_2 \wedge D_2 \sqsubseteq_\Gamma D_1$ where $\mathsf{name} : dom \to id$ gives the `name` of the domain declaration **`dom`** `name{p_1,..,p_n}` each domain type D_i was constructed from. It is necessary to verify that the domain templates $\Gamma(\mathsf{name}(D_i))$ are equal because the parameter layout computed by the concrete semantics given in Sec. 4.6 is not generally compatible for different domain templates. Only the **`array`**-expression can be used to convert between domains based on different templates.

4.5.3 Type Rules

Next, we discuss the type rules from Fig. 4.8 from top to bottom and introduce the necessary definitions.

Syntactic Construction For most of the domains computed by the type system, our definition gives the gist of the syntactic construction in terms of a mathematical function that defines the elements of the index set represented by the domain rather than explicitly giving the syntactic construction because the mathematical definition is easier to understand and the corresponding syntactic construction can be easily derived from the given definition. The actual syntactical constructions must retain the names of variables so that no relational information is lost.

Domain/Filter Declaration The rule (DomainDecl) syntactically constructs the index set as defined in Sec. 4.2.1. Valid domain declarations must have a cardinality smaller infinity (given that all domain parameters are finite). Here $|D|$ denotes the number of elements inside domain D. In general, the number of elements is computed symbolically as parametric piece-wise polynomials using the method from [11].

$$\frac{|\lambda(\vec{x})\,.\,\{\vec{i} \in \mathbb{Z}^m | f\}| < \infty \quad \Gamma, name : \lambda(\vec{x})\,.\,\{\vec{i} \in \mathbb{Z}^m | f\} \vdash p : t}{\Gamma \vdash dom\ name\{x_1, .., x_n\} = \{< i_1, .., i_m >: f\}\ p : t}\ (DomainDecl)$$

$$\frac{\Gamma(parent) = \lambda(\vec{x})\,.\,\{\vec{i} \in \mathbb{Z}^m | f\} \quad d = \lambda(\vec{x})\,.\,\{\vec{i} \in \mathbb{Z}^m | f\ and\ f'\}(\vec{x'} \to \vec{x})(\vec{i'} \to \vec{i}) \quad |d| < \infty \quad \Gamma, name : d \vdash p : t}{\Gamma \vdash dom\ name\{x'_1, .., x'_n\}(l_1, .., l_k) = \{(i'_1, .., i'_m)\ in\ parent\{x'_1, .., x'_n\} : f'\}\ p : t}\ (FilterDecl)$$

$$\frac{\Gamma \vdash e_i : \alpha_i[D^0] \quad \alpha_i \leq int[D^0] \quad d' = \mathsf{inst}_\Gamma(d\{e_1, .., e_{|d\{\}|}\})}{\Gamma \vdash d\{e_1, .., e_{|d\{\}|}\} : \bot_\alpha[d']}\ (DomExp)$$

$$\frac{\Gamma \vdash e : \alpha[d]}{\Gamma \vdash \backslash(v_1, .., v_n) \text{->} e : int^n \to \alpha[d]}\ (Lambda)$$

$$\frac{\Gamma \vdash e_t : \alpha_t[d_t] \quad \Gamma, v_i : (d^i \| _i)[D^0] \vdash f_i : int^{\mathsf{dim}(d_i)} \to \alpha_i[D^0] \quad \Gamma \vdash e_i : \alpha_i[d^i] \quad f_i = \backslash(\vec{v}) \to e'_i \quad |\bigcap_i D^R[\![e_i]\!](d^i)| = 0 \quad d_t =_\Gamma \bigsqcup_j D^R[\![e_j]\!](d^j)}{\Gamma \vdash array\ e_t\ (e_1, f_1)\ ...\ (e_n, f_n) : \bigsqcup_i \alpha_i[d_t]}\ (Array)$$

$$\frac{\Gamma \vdash e_i : \alpha^s_i[d^s_i] \quad \Gamma(id) = \alpha^t_1[d^t_1] \to\ ..\ \to \alpha^t_n[d^t_n] \quad \alpha^s_i \sqsubseteq_\Gamma \alpha^t_i \quad d^s_i \lessapprox_\Gamma d^t_i}{\Gamma \vdash id\ e_1\ ..\ e_{n-1} : \alpha^t_n((\bigtimes_i d^t_i)\{\} \oplus (\bigtimes_i \alpha^t_i)\{\} \to (\bigtimes_i d^s_i)\{\} \oplus (\bigtimes_i \alpha^s_i)\{\}))[d^t_n((\bigtimes_i d^t_i)\{\} \oplus (\bigtimes_i \alpha^t_i)\{\} \to (\bigtimes_i d^s_i)\{\} \oplus (\bigtimes_i \alpha^s_i)\{\}))]}\ (Apply)$$

$$\frac{\Gamma \vdash e : \alpha[d_1] \quad \Gamma \vdash e_i : \alpha_i[D^0] \quad d_1 \lessapprox_\Gamma \Gamma(\mathsf{parent}(d_2)) \quad \Gamma(d_2) = d'_2 \quad \alpha_i \leq int}{\Gamma \vdash e.d_2(e_1, ..., e_n) : \alpha[\mathsf{inst}_\Gamma(d_2\{\}(e_1, ..., e_n)).\mathsf{params}(d_1)]}\ (Filter)$$

$$\frac{\Gamma \vdash e : \alpha[d] \quad \Gamma \vdash e_i : \alpha_i \quad \alpha_i \leq int[D^0] \quad \bigtimes_i \Gamma(e_i)_1 \leq d}{\Gamma \vdash e[e_1, ..., e_{\mathsf{dim}(d)}] : \alpha[D^0]}\ (Access)$$

$$\frac{d'_1 = \mathsf{inst}_\Gamma(d_1\{p_1 : f_1, ..\}(k_1 : f'_1, ..)) \quad d'_2 = \mathsf{inst}_{\Gamma'}(d_2\{e'_1, .., e'_m\}(e''_1, .., e''_n)) \quad (\vec{e'} \oplus \vec{e} \oplus \vec{e''})^T = M \cdot (\vec{p} \oplus \vec{i} \oplus \vec{k})^T \quad \Gamma' \vdash e_i : t^i \quad d'_2 =_{\Gamma'} \bigtimes_i t^i_1 \quad M^{-1}M = \mathbb{1} \quad *}{\Gamma \vdash lens\ (i_1, .., i_{\mathsf{dim}(d_1)})\ in\ d_1\{p_1 : f_1, .., p_n : f_n\}(k_1 : f'_1, .., k_l : f'_l) \to (e_1, .., e_{\mathsf{dim}(d_2)})\ in\ d_2\{e'_1, .., e'_m\}(e''_1, .., e''_n)\ p : t}\ (LensDef)$$

where $* = \Gamma \odot d_1\{p_1 : f_1, ..\}(k_1 : f'_1, ..), d_1 \to d_2 : \langle M^{-1}, M \rangle \vdash p : t$ and $\Gamma' = \Gamma, p_i : \mathsf{params}(d'_1)\|_i, k_i : \mathsf{filters}(d'_1)\|_i, i_i : d'_1\|_i$

$$\frac{\Gamma \vdash e : \alpha[d'_1] \quad \Gamma(d_1 \to d_2) = \langle M^{-1}, M \rangle \quad d'_1 \sqsubseteq_\Gamma d_1}{\Gamma \vdash e \text{->} d_2 : \alpha[d_2]}\ (Lens)$$

$$\frac{\Gamma \vdash f : t \quad t \sqsubseteq_\Gamma \lambda().\{\langle i \rangle\, | 0 \leq i \leq 1\} \quad \Gamma, V : (D^0[\![f]\!])[D^0] \vdash e_1 : \alpha_1[d] \quad \Gamma, V : (D^0[\![not\ f]\!])[D^0] \vdash e_2 : \alpha_2[d]}{\Gamma \vdash if(f)\ e_1\ else\ e_2 : \alpha_1 \cup \alpha_2[d]}\ (If)$$

$$\frac{\Gamma \vdash e : \alpha[d_1] \quad \Gamma(d_2) = d_2' \quad d_1 \precsim_\Gamma \Gamma(\mathsf{parent}(d_2)) \quad |\mathsf{filters}(d_2'.\mathsf{params}(d_1))| < \infty}{\Gamma \vdash e \text{»} d_2 : \bot_\alpha[\mathsf{filters}(d_2'.\mathsf{params}(d_1))]} \; (FilterRange)$$

$$\frac{\Gamma \vdash e' : \alpha'[d'] \quad \alpha_i'' = \mathsf{inst}_\Gamma(\alpha''\{V_1 : f_1, ..\}) \quad d_i'' = \mathsf{inst}_\Gamma(d''\{V'_1 : f'_1, ..\}(V_1'' : f_1'', ..)) \quad \Gamma' = \Gamma \odot t_v \odot t_d, id : rn(\alpha_i'', id)[d_i''] \quad \alpha' \sqsubseteq_{\Gamma'} \alpha_i'' \quad d' \precsim\!\approx_{\Gamma'} d_i'' \quad \Gamma' \vdash e : \alpha[d]}{\Gamma \vdash let\ id : \alpha''\{V_1 : f_1, ..\}[d''\{V'_1 : f'_1, ..\}(V_1'' : f_1'', ..)] = e'\ in\ e : \alpha[d]} \; (Assign)$$

$$\frac{\alpha_i'[d_i'] = \mathsf{inst}_\Gamma(\alpha_i\{V_i^1 : f_i^1, ..\})[\mathsf{inst}_\Gamma(d_i\{V'^1_i : f'^1_i, ..\}(V''^1_i : f''^1_i, ..))] \quad \Gamma \odot \alpha_1\{..\} \odot d_1\{..\} .. \odot \alpha_{n-1}\{..\} \odot d_{n-1}\{..\}, id : rn(\alpha_1', id_1)[d_1'] \rightarrow \ .. \ \rightarrow rn(\alpha_n', id_n)[d_n'] \vdash p : t}{\Gamma \vdash id : \alpha_1\{V_1^1 : f_1^1, ..\}[d_1\{V'^1_1 : f'^1_1, ..\}(V''^1_1 : f''^1_1, ..)] \rightarrow .. \rightarrow \alpha_n\{V_n^1 : f_n^1, ..\}[d_n\{V'^1_n : f'^1_n, ..\}(V''^1_n : f''^1_n, ..)]\ p : t}$$

where id is declared as $id\ id_1\ ..\ id_{n-1} = e$ $(FunType)$

$$\frac{\Gamma(id) = \alpha_1'[d_1'] \rightarrow .. \rightarrow \alpha_n'[d_n'] \quad \Gamma \vdash e : \alpha_n[d_n] \quad \alpha_n \sqsubseteq_\Gamma \alpha_n' \quad d_n \precsim_\Gamma d_n' \quad \Gamma \vdash p : t}{\Gamma \vdash id\ id_1\ ..\ id_{n-1} = e\ p : t} \; (FunDef)$$

$$\frac{\Gamma \vdash e_1 : \alpha_1[D^0] \quad \Gamma \vdash e_2 : \alpha_2[D^0] \quad \alpha = \lambda(\alpha_1\{\} \oplus \alpha_2\{\}).\{i \in \mathbb{Z} | true\}[\![i = j\ op\ k\ and\ j \in \alpha_1\ and\ k \in \alpha_2]\!]}{\Gamma \vdash e_1\ op\ e_2 : \alpha[D^0]} \; (IntExp)$$

Figure 4.8: Type rules for our language where Γ is a type environment mapping names to a type. $\mathsf{parent}(dom)$ denotes the parent domain of a filter as defined in Sec. 4.2.1. $\mathsf{dim}(d) = |d\,\langle\rangle\,|$ denotes the dimension of a domain d. $D^R[\![e]\!](d)$ (Eq. 4.8 in the Appendix) symbolically applies the combined reverse transformation of all lenses in e to d. $\Gamma \odot d\{V_1 : f_1, .., V_{|d|} : f_{|d|}\}(V_{|d|+1}, ..) \triangleq \Gamma, V_i : (\lambda().\{\langle V_i \rangle \in \mathbb{Z}^1 | true\}[\![f_i]\!])[D^0]$ adds the new values V_i to the environment. The following functions are defined in Sec. 4.5, where the type rules are discussed. $inst_\Gamma$ denotes the instantiation of a domain template with the given parameters. $\mathsf{filters}(d)$ denotes the domain that represents all possible values of the filter parameters of d, $\mathsf{params}(d)$ denotes the domain that represents all possible parameters of d. The application of all values in a domain d_2 as parameters for d_1 is denoted as $d_1.d_2$. Domain subset tests are denoted as $d_1 \sqsubseteq_\Gamma d_2$. Domain subset tests where the template domains must be equal are denoted as $d_1 \precsim\!\approx_\Gamma d_2$. The cartesian product is denoted as $d_1 \times d_2$ and $d\|_i$ denotes the domain obtained by projecting the i-th index out of d. Renaming is defined as $rn(D, id) \triangleq D(D\,\langle\rangle \rightarrow id)$.

Domain Instance The (DomExp) rule computes the type of a domain instance which represents a domain template where the parameters are substituted by program expressions.

The substitution is performed using the following function, where the type information of the arguments is added to the domain.

$$\begin{aligned}\mathsf{inst}_\Gamma(d\{V_1 : f_1, .., \}(V_1' : f_1', ...)) &\triangleq d'(\vec{X} \to \vec{V})(\vec{F} \to \vec{V'}) \\ \mathsf{inst}_\Gamma(d\{e_1, .., e_n\}(e_1', .., e_m')) &\triangleq d'[\![X_i = e_i \text{ and } F_i = e_i']\!],\end{aligned}$$

where $d'\{\} = \vec{X} \oplus \vec{F}$ and $d' = \Gamma(d)$.

Lambda For brevity, our language is first order but the build-in functions *array* and *reduce* accept lambda expressions. The lambda type rule (Lambda) and the type rule for **`array`**-expression (Array) is shown. The type rules for the *reduce* operation are omitted for brevity but can be easily deduced from the given rules.

Array For every tuple (e_i, f_i) passed as argument to *array*, the type of e_i defines the set of (index) values f_i will be applied to. Therefore, f_i is typed in an environment where the type of the argument $\vec{v}$ of f_i is given by the index domain of e_i. Thus, the type of the i-th component v_i is given by projecting the i-th dimension out of the index domain of e_i using

$$D\|_i \triangleq \lambda(\vec{X}).\{I_i | \vec{I} \in D(\vec{X})\}.$$

The lambda is typed in this environment, so that usages of $\vec{v}$ in the lambda (e.g. array accesses) can obtain information about the possible values of $\vec{v}$ (e.g. out of bounds checks where the index is computed using $\vec{v}$).

As lenses provide transparent views rather than actual coordinate transformations, $f_i\ \vec{v}$ generates an element α at index $\vec{v'} = R[\![e_i]\!] \cdot \vec{v}$ where $R[\![e_i]\!]$ (defined in Eq.4.6) is a linear transformation that combines the reverse of all lens transformations in e_i.

$D^R[\![e]\!](d)$ (defined Eq. 4.8) symbolically applies the combined reverse transformation $R[\![e]\!]$ to d so that $D^R[\![e_i]\!](d_i)$ gives the domain that contains all actual index tuples generated by applying f_i to all indices in d_i.

The (Array) rule verifies that the meet of all reverse transformed index domains is empty and the join is equivalent to the result domain d_t.

Apply In order to verify that the relational constraints defined by the declared function parameters hold for the given function arguments, (Apply) compares the Cartesian product of all argument domains and declared parameter domains. The Cartesian product of two domains $D_1 \times D_2$ merges two domain definitions into a single, higher dimensional domain so that

$$D_1 \times D_2 \triangleq \lambda(\vec{X} \oplus \vec{X'}).\{\vec{v} = \vec{v_1} \oplus \vec{v_2} | \vec{v_1} \in D_1(\vec{X}) \wedge \vec{v_2} \in D_2(\vec{X'})\}.$$

The type calculated by (Apply) is the declared return type of the function where the declared parameters are renamed to the actually supplied parameters so that the relational information from the supplied parameters is not lost. Relational information is collected in the (If), (FunType) and (Assign) rules in the variables V_i which are added to the type environment.

Filter The parameters from d_1 and the filter parameters are substituted into the filter domain d_2.

The domain describing the possible parameter values of the domain D is defines as

$$\mathsf{params}(D) \triangleq \lambda(\vec{F}).\{\vec{x} \in \mathbb{N}^{|\vec{x}|} | D(\vec{x} \oplus \vec{F}) \neq \emptyset\}.$$

Domain application $D_1.D_2$ constructs a domain where the values from D_2 are applied as parameters to D_1,

$$D_1.D_2 = \lambda(\vec{X}'). \bigcup_{\vec{v} \in D_2(\vec{X}')} D_1(\vec{v}).$$

Access Rule (Access) verifies that the index is in bounds and gives the value domain of the array as result.

Lenses The (LensDef) rule verifies that the transformation of parameters and indices is linear and surjective. In addition, the transformation must be reversible, so a left-inverse of M must exist. This implies that lens transformations declarations must always transform from lower to higher dimensional space. This may require to specify the transformation with a backward arrow, as shown in Fig. 4.7. Generally, a lens can be applied only in direction of the arrow.

The domain describing the possible filter parameter values of the domain D is defines as

$$\mathsf{filters}(D) \triangleq \lambda(\vec{X}).\{\vec{l} \in \mathbb{N}^{|\vec{l}|} | D(\vec{X} \oplus \vec{l}) \neq \emptyset\}.$$

If The relational information available through the conditions in if expressions are added to the type environment using fresh variables (that cannot be addressed by the programmer). Non-linear (in)equalities are disregarded when added to the corresponding domains as defined in $D[\![f]\!]$.

Filter Range As shown in Sec. 4.2.1, filter parameters allow to address specific subsets of a parent domain like the different rows of a matrix. Often it is also useful to address higher level objects, for example all rows or blocks in a matrix. An example of this is shown in Fig. 4.7. For this purpose, we introduce the filter range operation which constructs a domain that contains all possible filter parameter values.

The filter range operation is performed using the » operator and no filter arguments are given so that **array** `d2{4,4} (d2{4,4},\(i,j)->0)>>row())` denotes the domain of the filter parameters of *row* which is equal to $\lambda().\{\langle I \rangle | 0 \leq I < 4\}$. Now, a new array can be constructed row wise: **array** `d2{4,4} (d2{4,4}>>row,\(i)->f(m.row(i)))` where `m` has type `int[d2{4,4}]` and `f` computes a new row from the given one so that `f` is applied to all rows in `m` to define the new array.

We define a special case for domain application $D^0.D \triangleq D$ so that filter range can be seamlessly added to the existing type rules.

To include filter ranges in the type rule (Array), d_i must be replaced by domain application $d_i'.d_i$ everywhere in the type rule, so that a domain that represents the set of indices for which an argument (e_i, f_i) will generate values is constructed, where the possible types of f_i are extended so that $\Gamma \vdash f_i : \alpha_i[d_i']$.

Domain application distinguishes two kinds of lambdas f_i, depending on its type. A single application of a lambda f_i may either compute an array of values ($d_i' \neq D^0$) or it computes a single value ($d_i' = D^0$).

So given a function f_i that computes an array of values and a filter range domain d_i and the corresponding filter domain d_i', $d_i'.d_i$ computes the domain which applies all possible filter values to the filter. So the set of all indices f_i generates values for, when f_i is applied to all indices in d_i, is given by $d_i'.d_i$. If f_i computes a single value rather than an array of values, then $d_i' = D^0$ and $d_i'.d_i = d_i$ so again $d_i'.d_i$ gives the set of index tuples generated by f_i when it is applied to all index tuples in d_i.

Assignment Type rule (Assign) verifies that the given type is a subtype of the target type given by a domain instance. In addition, the index of the value domain of the assigned value id is renamed to id so that the formula $f = \Gamma(id)_1[\![]\!]$ describes relational information on id and can be used to refine the domains of other values that depend on id. Furthermore, the variables V_i declared inside the domain instance `d{V_1:f_1,...}` are added to the environment using the $\odot$ operator which defined at the bottom of Fig. 4.8.

FunType/FunDef The (FunType) rule uses inst_Γ to compute value and index domains of all arguments, renames the index variables of the value domains of variable id_i to id_i and adds the variables from the domain instantiation to the type environment using the $\odot$ operator before adding the type of id. The (FunDef) rule verifies that the function body e is a subtype of the declared function result type.

IntExp The constructed result domain represents the result of the operation e_1 *op* e_2, if the operation is linear. Otherwise, a unrestricted integer domain is constructed.

4.6 Semantics

A language without semantics has no meaning, so in this section we define the formal (denotational) semantics of our language from Fig. 4.2. Afterwards, the semantics is used to prove that the our type system is sound. In addition, we prove that the type system is complete if all expressions are linear and branch decisions as well as array accesses are randomized.

Program State

The program state is a partial function $(id \hookrightarrow \mathbb{N}^n \times \mathcal{P}(\mathbb{N}_{\mathbb{N}^m}))$ mapping value names to a tuple where the second component is a set of natural numbers indexed by a vector $\vec{i} \in \mathbb{N}^m$ which represent an array and the first component is a vector that contains the dynamic parameter values of the index domain type used to construct the array. The set of indexed numbers represents the array elements associated with a variable name so that $v_{\langle i_1,..,i_m\rangle} \in \rho(name)$ represents the element denoted by the array access $name[i_1,..,i_m]$ if the index exists.

The semantics of the update operator of a program state $\rho : id \hookrightarrow \mathbb{N}^n \times \mathcal{P}(\mathbb{N}_{\mathbb{N}^m})$ is defined as an overwrite, if the name was defined in ρ, or as an extension to the partial map when it was not previously defined.

$$\rho[x \to V] = \rho' : \rho'(x) = V, \\ \forall y \in \mathsf{dom}(\rho) : y \neq x \Rightarrow \rho(y) = \rho'(y)$$

To shorten the notation we also define

$$\rho[\vec{x} \to \vec{V}] = \rho[x_1 \to V_1]..[x_{|\vec{x}|} \to V_{|\vec{x}|}]$$

If x is not defined in ρ then we write $\rho(x) = \bot$.

$$
\begin{array}{rcll}
\text{Expression} & \triangleq & \text{Semantic} & \text{Type} \\
V[\![x]\!]^\rho & \triangleq & \rho(x) & \preceq \gamma(x) \text{ since } \forall\, e : V[\![e]\!]^{\rho'} \preceq \gamma(e) \wedge \rho(x) = V[\![e]\!]^{\rho'} \\
V[\![const]\!]^\rho & \triangleq & \left\langle \langle\rangle, \{const_{\langle\rangle}\} \right\rangle & \approx \gamma(const) = \lambda().\{I \in \mathbb{Z} | I = const\}[D^0] \\
V[\![e_1\ op\ e_2]\!]^\rho & \triangleq & V[\![val(V[\![e_1]\!]_2^\rho)\ op\ val(V[\![e_2]\!]_2^\rho)]\!]^\rho = \left\langle \langle\rangle, \{(e_1\ op\ e_2)_{\langle\rangle}\} \right\rangle & \preceq \gamma(e_1\ op\ e_2)_1 \text{ over-approx of non-lin. exp} \\
V[\![e.f(e_1,..,e_n)]\!]^\rho & \triangleq & \left\langle \vec{p} \oplus val(V[\![\vec{e}]\!]_2^\rho) \oplus \vec{f} \oplus val(V[\![\vec{e}]\!]_2^\rho), \{v_{\vec{i}} | v_{\vec{i}} \in V[\![e]\!]_2^\rho \wedge \vec{i} \in \gamma(e.f(e_1,..,e_n))\} \right\rangle & \preceq \gamma(e.f(e_1,..,e_n)) \text{ due to removed values} \\
 & & \text{where } \vec{p} \oplus \vec{k} \oplus \vec{f} = V[\![e]\!]_1^\rho & \\
V[\![e\text{->}d]\!]^\rho & \triangleq & \left\langle \vec{p'} \oplus \vec{k'} \oplus \vec{f}, \{v_{\vec{i'}} | v_{\vec{i}} \in V[\![e]\!]_2^\rho\} \right\rangle \text{ where } \gamma(e\text{->}d) = \left\langle M^{-1}, M \right\rangle & \approx' \gamma(e\text{->}d) \text{ as } M \text{ is bijection on } \vec{i} \text{ from } d_1 \text{ to } d_2 \\
 & & \text{and } (\vec{p'} \oplus \vec{i'} \oplus \vec{k'})^T = M \cdot (\vec{p} \oplus \vec{i} \oplus \vec{k})^T \text{ and } \vec{p} \oplus \vec{k} \oplus \vec{f} = V[\![e]\!]_1^\rho & \\
V[\![e\text{»}d]\!]^\rho & \triangleq & V[\![e]\!]^\rho & V[\![e\text{»}d]\!]_2^\rho \text{ is never used} \\
V[\![e[e_1,..,e_n]]\!]^\rho & \triangleq & \begin{cases} V[\![v]\!]^\rho & : v_{\vec{i}} \in V[\![e]\!]_2^\rho \text{ where } \vec{i} = val(V[\![\vec{e}]\!]_2^\rho) \\ \top & : \text{else} \end{cases} & \text{see Lemma 5 } (Access) \\
V[\![if(b)\ e_1\ else\ e_2]\!]^\rho & \triangleq & \begin{cases} \top & : \exists \vec{i} : \neg(\alpha_{\vec{i}} \in V[\![e_1]\!]_2^\rho \wedge \alpha'_{\vec{i}} \in V[\![e_2]\!]_2^\rho) \\ V[\![e_1]\!]^\rho & : val(V[\![b]\!]_2^\rho) = true \\ V[\![e_2]\!]^\rho & : val(V[\![b]\!]_2^\rho) = false \end{cases} & \begin{cases} \top & : \gamma(e_1)_2 \neq_\Gamma \gamma(e_2)_2 \\ \preceq \gamma(e_1)_1 \sqcup \gamma(e_2)_1[\gamma(e_1)_2] & : \text{else} \end{cases} \\
V[\![d\{e_1,..,e_n\}]\!]^\rho & \triangleq & \langle val(V[\![\vec{e}]\!]_2^\rho), \emptyset \rangle & V[\![d\{e_1,..,e_n\}]\!]_2^\rho \text{ is never used} \\
V[\![let\ id : \alpha[d\{V_1,..,V_n\}] = e'\ in\ e]\!]^\rho & \triangleq & V[\![e]\!]^{\rho'} \text{ where } \rho' = \rho[\vec{V} \to V[\![\vec{p}]\!]^\rho][id \to V[\![e']\!]^\rho] \text{ where } (\vec{p} \oplus \vec{k} \oplus \vec{f}) = V[\![e']\!]_1^\rho & \text{see Lemma 6 } (Let) \\
V[\![id\ e_1\ ..\ e_{n-1}]\!]^\rho & \triangleq & V[\![e]\!]^{\rho'} \text{ where } id\ id_1\ ..id_{n-1} = e \text{ and } \gamma(id) = \alpha'_1[d'_1] \to\ ..\ \to \alpha'_n[d'_n] \text{ and} & \text{see Lemma 7 } (Apply) \\
 & & \vec{V_i} = d'_i\{\} \text{ and } \rho' = \rho[\vec{V}^1 \to V[\![\vec{p^1}]\!]^\rho][id_1 \to V[\![e_1]\!]^\rho]..[\vec{V}^{n-1} \to V[\![\vec{p^{n-1}}]\!]^\rho][id_{n-1} \to V[\![e_{n-1}]\!]^\rho] & \text{and } (\vec{p^i} \oplus \vec{k^i} \oplus \vec{f^i}) = V[\![e_i]\!]_1^\rho \\
V[\![array\ e_t\ (e_1,f_1)\ ..\ (e_n,f_n)]\!]^\rho & \triangleq & \left\langle \vec{p_t} \oplus \vec{k_t}, \bigsqcup_j \left\{ v_{\vec{i'}} | \begin{cases} \top : |\bigcap_i D^R[\![e_i]\!](d_i)| \neq 0 \\ \top : d \neq_\Gamma \bigsqcup_j D^R[\![e_j]\!](d_j) \\ \vec{i} \in d_j(\vec{p_t} \oplus \vec{k_t}) \wedge (\vec{p'} \oplus \vec{i'} \oplus \vec{f})^T = \\ \quad R[\![e_j]\!](\vec{p^j} \oplus \hat{l} \oplus \vec{f^j}) \wedge \{v_{\vec{l}}\} = V[\![f_j(\vec{i})]\!]_2^\rho & : \text{else} \end{cases} \right\} \right\rangle & \text{see Lemma 8 } (Array)
\end{array}
$$

where $\alpha_j[d_j] = \gamma(e_j)$ and $\alpha[d] = \gamma(e_t)$ and $\vec{p^j} \oplus \vec{k^j} \oplus \vec{f^j} = V[\![e_j]\!]_1^\rho$ and $\gamma(f_i) = \left\langle int^{\mathbf{dim}(d_i)} \right\rangle \to \alpha_i[d_i]$ and $\vec{p_t} \oplus \vec{k_t} \oplus \vec{f_t} = V[\![e_t]\!]_1^\rho$ and $\hat{l} \triangleq \begin{cases} \vec{i} : \vec{l} = \langle\rangle \\ \vec{l} : \text{else} \end{cases}$

Figure 4.9: Recursive definition of the value semantics on exp $V[\![e]\!] : (id \hookrightarrow \mathbb{N}^n \times \mathcal{P}(\mathbb{N}_{\vec{i}})) \hookrightarrow \mathbb{N}^n \times \mathcal{P}(\mathbb{N}_{\vec{i}})$.

4.6.1 Expression Semantics

Before giving the expression semantics of our language we define helper functions to extract values from the program state and compare the program state to types of expressions.

Given a set with a single element, the partial function $val : \mathcal{P}(\mathbb{N}_{\vec{i}}) \hookrightarrow \mathbb{N}$ extracts the element $val(\{s_{\vec{i}}\}) \triangleq s$ and $val(\vec{X}) \triangleq \left\langle val(X_1), .., val(X_{|\vec{X}|}) \right\rangle$.

The binary relation $\preceq$: $(\mathbb{N}^n \times \mathcal{P}(\mathbb{N}_{\mathbb{N}^m})) \times (dom \times dom)]$ defined as

$$\langle \vec{p}, V \rangle \preceq d^{val}[d^{ind}] \triangleq \forall v_{\vec{i}} \in V : v \in d^{val} \wedge \{\vec{i} | v_{\vec{i}} \in V\} = d^{ind}$$

contains all pairs where the value domain of an expression over-approximates the concrete values $V[\![e]\!]_2^\rho$ and the index domain precisely represents the indices in $V[\![e]\!]_2^\rho$.

The binary relation $\approx$: $(\mathbb{N}^n \times \mathcal{P}(\mathbb{N}_{\mathbb{N}^m})) \times (dom \times dom)$

$$\langle \vec{p}, V \rangle \approx d^{val}[d^{ind}] \triangleq \forall v_{\vec{i}} \in V : \{v\} = d^{val} \wedge \{\vec{i} | v_{\vec{i}} \in V\} = d^{ind}$$

contains all pairs where the type of an expression precisely represents the concrete value $V[\![e]\!]^\rho$.

Finally, Fig. 4.9 defines the value semantics on expression e,
$V[\![e]\!] : (id \hookrightarrow \mathbb{N}^n \times \mathcal{P}(\mathbb{N}_{\mathbb{N}^m})) \hookrightarrow \mathbb{N}^n \times \mathcal{P}(\mathbb{N}_{\mathbb{N}^m})$ where $V[\![\langle x_1, .., x_n \rangle]\!]^\rho$ is a short hand notation for $\langle V[\![x_1]\!]^\rho, .., V[\![x_1]\!]^\rho \rangle$.

4.6.2 Soundness and Completeness Proof

To shorten the notation, we define $\gamma(e) \triangleq \langle \mathsf{env}_\Gamma(\alpha), \mathsf{env}_\Gamma(d) \rangle$ where $\Gamma \vdash e : \alpha[d]$.

Our type system is sound, given that

Lemma 2. $\forall\ e \in program : V[\![e]\!]^\rho = \top \Rightarrow \gamma(e) = \top$.

4.6.3 Γ approximates ρ

We use structural co-induction to prove soundness of the type system in the sense that any array out of bounds access or array application with over or under specified result domain results in a type error, i.e. that $\forall\ e \in program : V[\![e]\!]^\rho = \top \Rightarrow \gamma(e) = \top$. First, we prove that our type system over-approximates the concrete program state. The binary relation $\Delta \subseteq (id \hookrightarrow \mathbb{N}^n \times \mathcal{P}(\mathbb{N}_{\mathbb{N}^m})) \times (id \hookrightarrow dom \times dom])$ defines the condition that must hold so that the type system over-approximates the concrete program state:

$$\forall\ x \in id : \rho(x) \preceq \gamma(x) \tag{4.3}$$

Given the definitions, $V[\![e]\!]^\rho \approx \Theta(e) \Rightarrow V[\![e]\!]^\rho \preceq \Theta(e)$ must hold.

We define conditional precise approximation to simplify the completeness proof.

$$V[\![e]\!]^\rho \approx' \Theta(e) \triangleq \begin{cases} V[\![e]\!]^\rho \approx \Theta(e) : \forall\ e_i \in e : V[\![e_i]\!]^\rho \approx \Theta(e_i) \\ V[\![e]\!]^\rho \preceq \Theta(e) : \text{else} \end{cases} \tag{4.4}$$

$V[\![e]\!]^\rho \approx' \Theta(e)$ indicates precise approximation iff all sub-expressions e_i syntactically contained in e are approximated precisely.

Start of the Induction

In order to show that $\rho\Delta\Gamma$ holds we prove (4.3) by structural induction over the language semantics. The start of the induction is the state where $\forall x \in id : \rho(x) = \bot$ so that $\rho\Delta\Gamma$ trivially holds.

Induction Step

First, we show that constrained domain parameters in `d{V_1:f_1,..}` matched to a domain instance `d{e_1,..,e_n}` are correctly approximated by the type system.

Lemma 3. $(V[\![d\{e_1,..,e_n\}]\!]_1)_i \preceq \Theta(V_i)$ *where* `d{V_1:f_1,..}` *is matched to* `d{e_1,..,e_n}`

Proof. The only type rules that allow matching are (Apply) and (Assign). Both rules very that $\gamma(e_i)_1 \sqsubseteq_\Gamma (\lambda().\{\langle V_i\rangle \in \mathbb{Z}^1 | true\}[\![f_i]\!])$. □

Next, we show that the env function used inside $\gtrapprox_\Gamma$ and $\leq_\Gamma$ operators on domains retains the over-approximation of the concrete values.

Lemma 4. $V[\![e]\!]^\rho \preceq \Theta(e) \Rightarrow V[\![e]\!]^\rho \preceq \gamma(e)$

Proof. $\mathsf{env}_\Gamma(\Theta(e)_i)$ only adds (relational) constraints to $\Theta(e)_i$ which are known to hold for $V[\![\vec{e}]\!]^\rho$: Lemma 3 shows that all matchings from domain instances `d{e_1,..,e_n}` are correct. The type rules (Apply) and (Assign) verify that a matched domain instance d_s can only be assigned to a bigger matched instance d_b so that $d_s \sqsubseteq_\Gamma d_b$ must hold. Only type rule (If) allows to add constraints to the environment that make a domain smaller, but these constraints are verified dynamically. □

The trivial parts of the co-induction over the expression semantics are shown on the right hand side of Fig. 4.9. More involved cases from Fig. 4.9 reference the arguments presented in the following.

Lemma 5. *(Access)*

$$\begin{cases} V[\![\alpha]\!]^\rho & : \alpha_{\vec{i}} \in V[\![e]\!]_2^\rho \\ \top & : else \end{cases} \preceq \begin{cases} \Theta(e)_1[D^0] & : \bigtimes_i \Theta(e_i)_1 \sqsubseteq_\Gamma \Theta(e)_2 \\ \top & : else \end{cases}$$

Proof. Due to the induction we have $V[\![e]\!]^\rho \preceq \Theta(e) \wedge V[\![e_i]\!]^\rho \preceq \Theta(e_i)$. Due to Lemma 4, $V[\![\vec{e}]\!]^\rho \preceq \mathsf{env}_\Gamma(\bigtimes_i \Theta(e_i)_1) \sqsubseteq_\Gamma \bigtimes_i \Theta(e_i)_1$ must hold unless $V[\![e[e_1,..,e_n]]\!]^\rho = \top = \Theta(e[e_1,..,e_n])$. □

Lemma 6. *(Let)* $V[\![let\ id : \alpha[d\{V_1,..,V_n\}] = e'\ in\ e]\!]^\rho \triangleq V[\![e]\!]^{\rho[\vec{V}\to V[\![\vec{p}]\!]^\rho][id\to V[\![e']\!]^\rho]} \preceq (\Gamma, \vec{V} : \Gamma(\vec{V}), id : \Gamma(d\{\vec{V}\}))(e)$ *where* $(\vec{p} \oplus \vec{k} \oplus \vec{f}) = V[\![e']\!]_1^\rho$

Proof. Due to Lemma 3, $p_i \in \Gamma(V_i)_1$ and thus $\forall\ i : V[\![V_i]\!]^\rho \preceq \Gamma(V_i)$. From the induction, we have $V[\![e']\!]^\rho \preceq \Theta(e')$. Rule (Assign) verfies that $\Theta(e')_1 \sqsubseteq_\Gamma \mathsf{inst}_\Gamma(\alpha\{\vec{V}\}) \wedge \Theta(e')_2 \sqsubseteq_\Gamma \mathsf{inst}_\Gamma(d\{\vec{V'}\})$. Therefore, $V[\![e]\!]^{\rho[\vec{V}\to V[\![\vec{p}]\!]^\rho][id\to V[\![e']\!]^\rho]} \preceq t$ where $(\Gamma \oplus \alpha\{\vec{V}\} \oplus d\{\vec{V'}\}, id : \mathsf{inst}_\Gamma(\alpha\{\vec{V}\})[\mathsf{inst}_\Gamma(d\{\vec{V'}\})]) \vdash e : t$. □

Lemma 7. *(Apply)* $V[\![e]\!]^{\rho'} \preceq \Gamma(id\ e_1\ ..\ e_{n-1})$ *where* $\rho' = \rho[\vec{V^1} \to V[\![\vec{p^1}]\!]^\rho][id_1 \to V[\![e_1]\!]^\rho]..[\vec{V^{n-1}} \to V[\![\vec{p^{n-1}}]\!]^\rho][id_{n-1} \to V[\![e_{n-1}]\!]^\rho]$ *and* $(\vec{p^i} \oplus \vec{k^i} \oplus \vec{f^i}) = V[\![e_i]\!]_1^\rho$ *and* $id\ id_1\ ..id_{n-1} = e$ *and* $\Gamma(id) = \alpha'_1[d'_1(\vec{V^1})] \to\ ..\ \to \alpha'_n[d'_n(\vec{V^n})]$

Proof. $V[\![id\ e_1\ ..\ e_{n-1}]\!]^\rho$ can be written as $V[\![e']\!]^\rho$ where $e' = let\ id_1 : \alpha'_1[d'_1\{\vec{V^1}\}] = e_1\ in\ ..\ let\ id_{n-1} : \alpha'_{n-1}[d'_{n-1}\{\vec{V^{n-1}}\}] = e_{n-1}\ in\ let\ res : \alpha'_n[d'_1\{\vec{V^n}\}] = e\ in\ res$. The type rules (FunDef) together with (Apply) perform the same actions as the type rule for e' except for the parameter renaming in type rule (Apply) so that $\Theta(id\ e_1\ ..\ e_{n-1}) = \left\langle \Theta(e')_1(\vec{X} \to \vec{X'}), \Theta(e')_2(\vec{X} \to \vec{X'}) \right\rangle$. e' consists only of *let* expressions and the renaming only adds provably true constraints, which together with Lemma 6 yields that $V[\![e]\!]^{\rho'} \preceq \Gamma(id\ e_1\ ..\ e_{n-1})$. This also holds without the renaming, the renaming adds the relational information of the input variables to the result type and removes nothing since the input variables are a subtype of the declared parameters due to type rule (Apply). □

Lemma 8. *(Array)* $V[\![e^{array}]\!]^\rho \preceq \bigsqcup_j \alpha_j[d]$ *where* $e^{array} = array\ e_t\ (e_1, f_1)\ ..\ (e_n, f_n)$ *and* $\Theta(e_i) = d_i$ *and* $\Gamma(f_i) = \left\langle int^{\mathsf{dim}(d_i)} \right\rangle \to \alpha_i[d'_i]$

Proof. Type rule (Array) requires the same conditions for $\top$ as the concrete semantics so that $V[\![e^{array}]\!]^\rho = \top \Leftrightarrow \Theta(e^{array}) = \top$ follows.

Due to the induction, $V[\![f_j\ \vec{i}]\!] \approx' \Gamma(f_j\ \vec{i})$ so that $V[\![\alpha]\!] \approx' \Gamma(f_j\ \vec{i})$ holds. Since this holds for all j, $\forall\ \alpha_{\vec{i'}} : \alpha_{\vec{i'}} \in \bigsqcup_i \alpha_i$ must hold. Thus, the value domain of $\Theta(e^{array})$ over-approximates the concrete values.

Next we must prove that $\{\vec{i'} \| \alpha_{\vec{i'}} \in V[\![e^{array}]\!]^\rho_2\} = \Theta(e^{array})_2$

If $d'_j \neq D^0$ (so $d_j(\vec{p})$ is a filter range and $f_j\ \vec{i}$ calculates an array for domain $d'_j(\vec{p} \oplus \vec{i})$) then, due to the definition of $d'_j.d_j$ where all possible filter parameters $\vec{i} \in d_j$ are applied to d'_j , $\{\vec{i''} | \vec{i''} \in d'_j(\vec{i})\} = d'_j.d_j$ must hold. Otherwise, $d'_j = D^0$ (so $d_j(\vec{p})$ is an index set and $f_j\ \vec{i}$ calculates a single value and $d'_j.d_j = d_j$), $\{\vec{i''} | \vec{i''} = \vec{i}\} = d'_j.d_j = d_j$ must also hold. We define $D^0(\vec{i}) = \vec{i}$ so that the value(s) calculated by f_j can be written as $f_j\ \vec{i} = \{\alpha_{\vec{i''}} | \vec{i''} \in d'_j(\vec{i})\}$. Due to type rule (Array) where $d = \bigsqcup_j D^R[\![e_j]\!](d'_j.d_j)$ it is known that the join of the reverse transformed $d'_j.d_j$s is equal to d. Due to type rule (LensDef), the reverse transformation must be bijective so that the $\vec{i'}$ must cover all elements in d as $\vec{i''}$ covers all elements $d'_j.d_j$.

□

Result of the Induction

Due to the induction we have $\forall\ e \in program : V[\![e]\!]^\rho \preceq \Theta(e)$ and the first rule in Fig. 4.9 gives $\rho\Delta\Gamma$.

4.6.4 Soundness

Now, the induction from the previous section is used to prove soundness.

Lemma 9. $\forall\ e \in program : V[\![e]\!]^\rho = \top \Rightarrow \gamma(e) = \top$

Proof. The induction proves together with Lemma 4 that $\forall\ e \in program : V[\![e]\!]^\rho \preceq \gamma(e) \sqsubseteq_\Gamma \Theta(e)$ which implies $\forall\ e \in program : V[\![e]\!]^\rho = \top \Rightarrow \gamma(e) = \top$. □

4.6.5 Completeness

In this section we prove that our type system is complete in the sense that any array out of bounds access or array application with over or under specified result domain

in the semantics Fig. 4.9 leads to an error in the type system and vice versa so that $V[\![e]\!]^\rho = \top \Leftrightarrow \gamma(e) = \top$.

As shown in the previous section (Lemma 9) the type system is sound, i.e. it detects all such errors. The type system is complete under the assumption that all expressions which are concerned with array access and construction are linear and that the programmer always annotates the most precise type possible. In addition, we assume that the branch decisions as well as array accesses (reads and writes) are randomized, i.e. the concrete semantics randomly returns the result of the true or false branch, irrespective of the condition and a valid array access returns a random value from the index domain. The randomized assumptions are required because our universe of types abstracts all values of an array and the result of branches into a single domain (range of values). It is possible to handle these cases precisely by extending the universe of types to disjunctions of mappings from input domains to output domains rather than using a single domain per value. All methods presented in this chapter would still be applicable but in this scenario types may grow exponentially and type checking would become expensive so we have restricted the universe of types to a single domain.

Using the given assumptions, all $\preceq'$ in the induction Sec. 4.6.3 become $\approx$. Branches and array accesses become precise due to the randomization and non-linear relations that are disregarded in the type system do not exist. $\gamma(e)$ precisely contains all information (and relations) available for expression e making the type system sound and complete under the given assumptions, because env collects all available linear relations in e.

4.6.6 Combined Lens Transformation

In this section we define the helper functions required to compute a single linear transformation that combines all lens transformations in an expression e. As seen in rule (LensDef) of Fig. 4.8, each individual lens transformation where $\Gamma(d_1 \to d_2) = \langle M^{-1}, M\rangle$ and $M \in \mathbb{Q}^{|\vec{p'}\oplus\vec{i'}\oplus\vec{k'}|\times|\vec{p}\oplus\vec{i}\oplus\vec{k}|}$ transforms the parameters $\vec{p}$, indices $\vec{i}$ and filter parameters $\vec{k}$ from d_1 to the primed variables in d_2. The individual transformation expects as input only those filter parameters it depends on ($\vec{k}$). The combined transformation of all lenses depends on all filter parameters $\vec{f}$ that occur in the expression e. Thus, we construct the input vector for the combined transformation as $\vec{p}\oplus\vec{i}\oplus\vec{f}$. The ins functions defined below constructs a matrix that copies filter parameters from the end of the vector behind the indices, so that $\vec{p}\oplus\vec{i}\oplus\vec{k}\oplus\vec{f}$ for an individual lens can be constructed by matrix multiplication. The aug function defined below, augments the linear transformation M of an individual lens to apply the identity operation on the additional filter parameters $\vec{f}$, so that $(\vec{p'}\oplus\vec{i'}\oplus\vec{k'}\oplus\vec{f})^T = M^{aug}\cdot(\vec{p}\oplus\vec{i}\oplus\vec{k}\oplus\vec{f})^T$. Afterwards, the rem function defined below is used to construct a matrix that removes the individual output filter parameters yielding $\vec{p'}\oplus\vec{i'}\oplus\vec{f}$.

To shorten the notation, we define $\Theta(e) \triangleq t$ where $\Gamma \vdash e : t$.

The functions $F_c[\![e]\!]^f$ resp. $R_c[\![e]\!]^f$ use ins, aug and rem to combine all lens transformations in e into a single linear transformation transforming forward from $\Theta(B[\![e]\!])_2$ to $\Theta(e)_2$ resp. backwards from $\Theta(e)_2$ to $\Theta(B[\![e]\!])_2$. Here, the partial function $B[\![e]\!] : \mathtt{exp}$

computes the syntactic base expression of e be removing all lenses and filters.

$$\begin{aligned} B[\![e.f(l_1,..,l_n)]\!] &\triangleq B[\![e]\!] \\ B[\![e{->}d_2]\!] &\triangleq B[\![e]\!] \\ B[\![e{\gg}d]\!] &\triangleq B[\![e]\!] \\ B[\![e]\!] &\triangleq e. \end{aligned} \tag{4.5}$$

The aug function augments a given matrix $M \in \mathbb{Q}^{n\times|\vec{p}\oplus\vec{i}\oplus\vec{k}|}$ with additional rows and columns so that the identity operation is applied on the last $|\vec{f}|$ elements (appended filter parameters $\vec{f}$) of the augmented input vector $\vec{p}\oplus\vec{i}\oplus\vec{k}\oplus\vec{f}$

$$\begin{aligned} \mathsf{aug} &: \mathbb{Q}^{n\times m}\times\mathbb{N}\to\mathbb{Q}^{n+k\times m+k} \\ \mathsf{aug}((m_{ij}),k) &\triangleq (a_{ij}) \text{ where} \\ a_{ij} &= \begin{cases} m_{ij} & : i<n\wedge j<m \\ 1 & : i\geq n\wedge j\geq m\wedge i=j \\ 0 & : \text{else} \end{cases}. \end{aligned}$$

Before applying an individual lens transformation, the required filter parameters must be copied from the end of the input vector to the proper position in the input vector. The add function constructs a matrix that copies the k-th element of the input vector to position l before applying matrix M

$$\begin{aligned} \mathsf{add} &: \mathbb{Q}^{n\times m}\times\mathbb{N}\times\mathbb{N}\to\mathbb{Q}^{n\times m-1} \\ \mathsf{add}((m_{ij}),k,l) &\triangleq (m_{ij})\cdot(a_{ij}) \text{ where} \\ (a_{ij}) &= \begin{cases} 1 & : i<l\wedge i=j \\ 1 & : i=l\wedge j=k \\ 1 & : i>l\wedge i-1=j \\ 0 & : \text{else} \end{cases}. \end{aligned}$$

The ins function constructs a matrix that inserts c consecutive elements from the input vector starting at position k to position l before applying m

$$\begin{aligned} \mathsf{ins} &: \mathbb{Q}^{n\times m}\times\mathbb{N}\times\mathbb{N}\times\mathbb{N}\to\mathbb{Q}^{n\times m-c} \\ \mathsf{ins}((m_{ij}),k,c,l) &\triangleq \begin{cases} (m_{ij}) & : c=0 \\ \mathsf{add}((m_{ij}),k+1,c-1,l+1) & : \text{else} \end{cases}. \end{aligned}$$

The rem function constructs a matrix that removes c consecutive elements from the input matrix starting at position k before applying m

$$\begin{aligned} \mathsf{rem} &: \mathbb{Q}^{n\times m}\times\mathbb{N}\times\mathbb{N}\times\mathbb{N}\to\mathbb{Q}^{n-c\times m} \\ \mathsf{rem}((m_{ij}),k,c) &\triangleq (a_{ij})\cdot(m_{ij}) \text{ where} \\ (a_{ij}) &= \begin{cases} (m_{ij}) & : k<i \\ (m_{(i+c)j}) & : \text{else} \end{cases}. \end{aligned}$$

The comb function combines the previous functions to insert, augment and remove the appropriate filter arguments

$$\mathsf{comb}_c(M, f, d_1, d_2) \triangleq$$
$$\mathsf{rem}(\mathsf{aug}(\mathsf{ins}(M, f, |\vec{F}|, |\vec{X}| + |d_1\ \langle\rangle\ |), f, |\vec{F}|), c), |\vec{X'}| + |d_2\ \langle\rangle\ |, |\vec{L'}|)$$

where $\vec{X} \oplus \vec{F} = d_1\{\}$ and $\vec{X'} \oplus \vec{L'} = d_2\{\}$.

The join function constructs a matrix that moves the last elements (indices) to the index position before applying m

$$\mathsf{join} : \mathbb{Q}^{n\times m} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{Q}^{n-|\vec{i'}|\times m}$$
$$\mathsf{join}((m_{ij}), |\vec{X}|, |\vec{r}|, |\vec{f}|, |\vec{i'}|) \triangleq (m_{ij}) \cdot (a_{ij}) \text{ where}$$
$$(a_{ij}) \cdot (\vec{X} \oplus \vec{r} \oplus \vec{f} \oplus \vec{i'})^T = (\vec{X} \oplus \vec{i'} \oplus \vec{f})^T.$$

The partial function $R_c[\![e]\!]^f : \mathbb{Q}^{n\times m} \hookrightarrow \mathbb{Q}^{n'\times m'}$ defines a matrix that maps parameters and indices of $\Theta(e)_2$ to the parameters and indices of $\Theta(B[\![e]\!])_2$ by combining the reverse transformation that the lenses in e apply.

$$R_c[\![e{-}{>}d_2]\!]^f_A \triangleq R_c[\![e]\!]^{f-m}_{\mathsf{comb}_c(M^{-1},f-m,d_1,d_2)\cdot A} \tag{4.6}$$
$$\text{where } \Gamma \vdash e : \alpha[d_1] \text{ and } \Gamma(d_1{-}{>}d_2) = \left\langle M^{-1}, M \right\rangle$$
$$R_c[\![e.f(l_1,..,l_n)]\!]^f_A \triangleq R_c[\![e]\!]_A$$
$$R_c[\![e{\gg}d]\!]^f_A \triangleq R_c[\![e]\!]_A$$
$$R_c[\![e]\!]^f_A \triangleq A. \tag{4.7}$$

$R[\![e]\!](\vec{p} \oplus \vec{i} \oplus \vec{f}) \triangleq R_{|\vec{p}\oplus\vec{i}\oplus\vec{f}|}[\![e]\!]^{|\vec{p}\oplus\vec{i}\oplus\vec{f}|}_{\mathbb{1}} \cdot (\vec{p} \oplus \vec{i} \oplus \vec{f})^T$ shortens the notation.

The partial function $F_c[\![e]\!]^f : \mathbb{Q}^{n\times m} \hookrightarrow \mathbb{Q}^{n'\times m'}$ defines a matrix that maps parameters and indices of $\Theta(B[\![e]\!])_2$ to the parameters and indices of $\Theta(e)_2$ by combining the transformation that the lenses in e apply.

$$F_c[\![e{-}{>}d_2]\!]^f_A \triangleq F_c[\![e]\!]^{f+m}_{A\cdot\mathsf{comb}_c(M,f,d_1,d_2)}$$
$$\text{where } \Gamma \vdash e : \alpha[d_1] \text{ and } \Gamma(d_1{-}{>}d_2) = \left\langle M^{-1}, M \right\rangle$$
$$F_c[\![e.f(l_1,..,l_n)]\!]_A \triangleq F_c[\![e]\!]^f_A$$
$$F_c[\![e{\gg}d]\!]_A \triangleq F_c[\![e]\!]^f_A$$
$$F_c[\![e]\!]_A \triangleq A.$$

$F[\![e]\!](\vec{p}\oplus\vec{i}\oplus\vec{f}) \triangleq F_{|\vec{p}\oplus\vec{i}\oplus\vec{f}|}[\![e]\!]^{|\vec{p}\oplus\vec{i}|}_{\mathbb{1}} \cdot (\vec{p}\oplus\vec{i}\oplus\vec{f})^T$ shortens the notation. So, $(\vec{p'}\oplus\vec{i'}\oplus\vec{f})^T = R[\![e]\!](\vec{p}\oplus\vec{i}\oplus\vec{f})$ and $(\vec{p}\oplus\vec{i}\oplus\vec{f})^T = F[\![e]\!](\vec{p'}\oplus\vec{i'}\oplus\vec{f})$.

The partial function $L[\![e]\!]_{\vec{f}} : \mathtt{exp}^n \hookrightarrow \mathtt{exp}^m$ defines a vector that contains all expressions used as filter paramater in the order they occure in e.

$$L[\![e.f(l_1,..,l_n)]\!]_{\vec{f}} \triangleq L[\![e]\!]_{\vec{f}\oplus\langle(l_1),..,(l_n)\rangle}$$
$$L[\![e{-}{>}d_2]\!]_{\vec{f}} \triangleq L[\![e]\!]_{\vec{f}}$$
$$L[\![e{\gg}d]\!]_{\vec{f}} \triangleq L[\![e]\!]_{\vec{f}}$$
$$L[\![e]\!]_{\vec{f}} \triangleq \vec{f}.$$

$T[\![e]\!](d) = (\Gamma(B[\![e]\!])_2\{\} \oplus \vec{I} \oplus L[\![e]\!]_{\langle\rangle})^T$ gives the symbolic transformation (vector of expressions) from d to $\Gamma(B[\![e]\!])_2$ in $\vec{I}$:

$T[\![e]\!](d) \triangleq R[\![e]\!]_{\mathbb{1}} \cdot join(F[\![e]\!]_{\mathbb{1}}, \vec{X}, d\,\langle\rangle\,, L[\![e]\!]_{\langle\rangle}, d\,\langle\rangle) \cdot (\Gamma(B[\![e]\!])_2\{\} \oplus \Gamma(B[\![e]\!])_2\,\langle\rangle \oplus L[\![e]\!]_{\langle\rangle} \oplus d\,\langle\rangle)^T$

$D^R[\![e]\!] : dom \to dom$ defines the domain obtained by applying the reverse lens transformations in e to the domain given as argument.

$$D^R[\![e]\!](d) \triangleq \bigtimes_i \Gamma(I_i)_1 \tag{4.8}$$

where $(\Gamma(B[\![e]\!])_2\{\} \oplus \vec{I} \oplus L[\![e]\!]_{\langle\rangle})^T = T[\![e]\!](d)$.

4.7 Implementation

We have implemented the data-parallel sub language presented in this chapter inside the **funkyImp** compiler and run-time system, which translates the sub language into task and data-parallel C++ code.

The code generation of our compiler produces code with the semantics shown in Sec. Fig. 4.9 in the following way:

- all index and value domains used in the type system and semantics are constructed syntactically and checks are performed symbolically using the Barvinok/ISL library [11, 138]
- arrays are implemented as structs containing a vector of dynamic array parameters and a pointer to memory that can hold all elements contained in the smallest hyper cube that contains the indices obtained by applying the dynamic domain parameters to the domain template. The usual mapping from hyper cube coordinate to linear memory is applied, so that items from the lowest array dimension are linear in memory and data-parallel computations (vectorization) can be applied.
- the **funkyImp** run-time uses the Boehm garbage collector [19] to perform memory management.

Further details about the compiler implementation can be found in Chap. 7.

4.7.1 Optimizations

Our compiler applies the following optimizations to the generated code.

- instead of applying filters f_i and lenses d_i in an expression `d.f_1->d_1.f_2->d2` individually, the compiler transformes them into a single loop-nest that combines the effect of all filters and a single linear transformation that combines the effect of all lenses
- all values and arrays are unboxed, functions with parametric type are specialized for each type instance they are applied on
- the immutability in our functional language allows to safely parallelize the loop-nests using vectorization and multi-threading.

- we use a simple heuristic so that only large arrays or arrays that contain complex data structures as elements are task-parallelized to promote vectorization over parallelization which often yields better results

- the compiler tries to have nested *array* and *reduce* operations write directly into the result array of the outer most operation rather than copying intermediary results

4.8 Experimental Results

In this section we present an evaluation of the performance of our language. First, we compare several versions of a declarative matrix multiply written in our language to Repa, SAC, parallel C++ and MKL. The latter defines an absolute reference frame of performance of our language and shows that our high level declarative specification can be used to produce code that is competitive even with the hand optimized assembly code from MKL. Afterwards, we evaluate the relative performance impact of replacing branches by filters and removing array out of bounds checks.

4.8.1 Absolute Performance Comparison

Figure 4.10 shows the average execution time of a matrix times transposed matrix product for several relevant languages. The benchmark creates two matrices (of the size given in the table) with entries of type float, transposes the second matrix and performs a plain matrix multiply for those entries marked with (plain). (full) refers to a fully one level cache blocked implementation (the C++ version is shown in function `blocked` in Fig. 4.6, the **funkyImp** version of `blocked` in 4.7 line 36); (semi) refers to the partially blocked version shown in Fig. 4.7 line 46; (asm) refers to the hand tuned Math Kern Library from Intel. For each language it was attempted to find the best compiler settings to give optimal results.

The Repa (3.2.3.3 with ghc 7.6.3 LLVM 3.4) generated code uses a single SIMD register inside a simple conditional jump based loop. It scales linearly on the test system, indicating that the memory is not utilized to its maximum rate, which is probably caused by the low usage of SIMD instructions.

For the SAC (sac2c-1.00-17729-beta-linux-x86_64) generated code, the innermost loop is a simple conditional jump accumulating several SIMD registers. Overall, SAC performs close to the C++ version.

The C++ code (icpc 12.1.2, using full optimizations but no vectorization or parallelization pragmas) uses a simple highly vectorized inner loop which is memory bound. The sequential C++ version outperforms the parallel Repa version for the 2048^2 matrix. The Intel icpc compiler can automatically parallelize the plain matrix multiply (using the -parallel switch). The compiler fails to automatically parallelize the blocked version but state of the art tools like Pluto [12] may be able to achieve this.

Pluto [12] (0.11.1) with switches –parallel –tile (denoted as (par) in the table) achieves roughly the same performance as the automatically parallelized version from the C++ compiler. Applying the recommended switches (denoted as (all)) to get best performance according to the Pluto developers (-tile –l2tile –parallel –unroll –ufactor=4 and specifying blocking sizes) fails, as the –unroll switch triggers an error in the Pluto pipeline. The –unroll switch is broken since at least version 0.9 of Pluto, so we had to perform the experiments without the –unroll switch. Using the currently available best switches produces

Code/Size	2048^2	2048^2	4096^2	8192^2
parallel	no	yes	yes	yes
Repa [76] (plain)	13.2	3.3	26.9	235.9
SAC [124] (plain)	3.2	1.5	14.3	173.8
C++ (plain)	2.4	1.5	14.3	130.3
Pluto [12] (par)	-	1.5	12.7	131.8
Pluto (all)	-	3.9	error	error
funkyImp (plain)	-	1.3	13.6	127.3
funkyImp (full)	-	1.0	7.8	66.2
funkyImp (semi)	-	0.6	5.4	52.4
MKL [66] (asm)	-	0.4	1.9	15.7

Figure 4.10: Execution time of matrix times transposed multiplication benchmark in seconds. The benchmark creates two matrices (of the size given in the table) with entries of type float, transposes the second matrix and performs a plain matrix multiply for those entries marked with (plain), so no loop restructuring optimizations are applied by the programmer. Entries not marked with (plain) contain parallel decompositions applied by the programmer as explained in Sec. 4.8.1. All times were measured on an Intel Core i7 CPU 860 @ 2.80GHz with ca. 12 GB/sec memory throughput and hyper threading and speedstep disabled.

a parallel version that is slower than the sequential C++ (plain) version. This indicates that automatic cache blocking as performed by Pluto is currently not able to match the speed of manually decomposed parallel computations as shown by e.g. **funkyImp** (semi). Generally, Pluto is an impressive tool that can apply some of the optimizations that are enabled by **funkyImp** automatically but at the cost of control over the chosen optimization and decomposition. Unsurprisingly, full control over the parallel decomposition as introduced by the **funkyImp** language allows to achieve much better performance even in simple applications like a matrix multiply. For matrices of size 4096 the result for Pluto was omitted because the result matrix is incorrect, indicating a bug in the Pluto implementation.

The plain **funkyImp** version is slightly faster than the plain C++ version. This is the case because **funkyImp** is an implicitly parallel language that automatically parallelizes at the task level. So the construction of the two matrices and potentially the transposition of the second matrix run in parallel. Without the task-parallelism, **funkyImp** is as fast as the C++ version.

The remaining (non plain) benchmarks cannot be directly compared to the previous ones because they use other (more cache friendly) algorithms to perform the same matrix multiply.

The semi blocked **funkyImp** version (code shown in Fig. 4.7 line 36) outperforms the fully blocked version (Fig. 4.7 line 46) and comes quite close to MKL. As **funkyImp** generates C++ code, the performance also depends on the used backend compiler. Although the fully blocked version should show better cache behaviour, inspection of the innermost loops of both versions reveals that the icpc compiler generates sub optimal code for the Nehalem platform for the fully blocked code (too many register write backs per instruction) which generate stalls in the pipeline and make the fully blocked version slower.

The Math Kernel Library (MKL) from Intel uses fine tuned assembly code that does

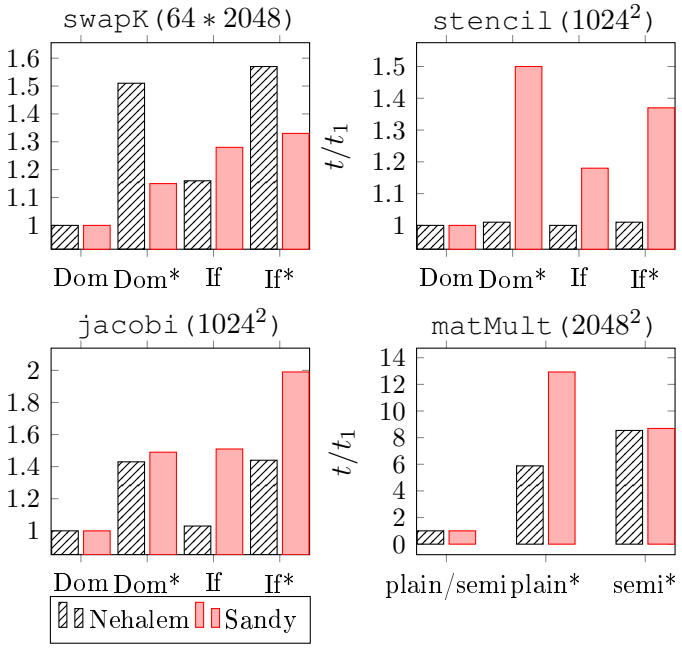

Figure 4.11: Each figure compares the relative performance of the different implementations of the program given in the figure caption (the bars are normalized relative to the first bar), where the problem size is given in brackets.

not suffer from unnecessary pipeline stalls and unsurprisingly outperforms all other implementations. Nevertheless, the simple and high level declarative code from the **funkyImp** (semi) version comes close and it is conceivable that improvements in the code generation of icpc reducing unnecessary pipeline stalls will reduce the gap between the two.

4.8.2 Relative Performance Comparison

Figure 4.11 shows the relative performance penalty when switching from a filter and **`array`** -expression based to a branch based array composition and when turning on dynamic out of bounds checks. The title of each figure gives the name of computational problem and its size. The bars in each figure show the slowdown of the different implementations relative to the implementation labeled Dom which uses **`array`**-expressions, filters and, thus, no dynamic out of bounds checks. The bars in the figure labeled If* show the slowdown that using a language which does not support **`array`**-expressions, filters and cannot statically prove the safety of array accesses would incur. For problems swapK, stencil and jacobi, the penalty is in the order of 30% to 100% for the matrix multiply in the order of 700% showing that using our language has the potential to produce notable performance improvements while giving strong safety guarantees. The performance of Nehalem version of the stencil problem shows little sensitivity to the implementation, indicating that the icpc compiler generally generates inefficient code for this problem on the Nehalem platform.

4.9 Related Work

In this section we discuss previous work on pure data-parallelism and languages supporting polytopes as well as parallel programming languages in general.

Stencil compilers like [135] have shown that high performance code can be generated for stencil based data-parallel computations. Yet, stencil operations are just a subset of the declarative array composition enabled by our language.

Our language extends slicing to generic array decomposition where the index set shapes are specified as parametric polytopes rather than simple hyper cubes. Furthermore, our language allows to combine arrays with non-overlapping index sets into larger arrays thus providing a general mechanism to de- and recompose arrays. A limited form of hyper cubic array decompositions is offered via slicing [38] in many previous languages like Fortran.

Previously, dependent types (e.g. [146]) and abstract interpretation (e.g. [137]) have been used to perform static bounds checking on arrays. Dependent types place strong restriction on the programming language and have not yet been applied successfully to imperative languages like C/C++. Dependent types are not part of the language so they cannot be used to construct arrays declaratively. The abstract interpretation based methods require an inter-procedural fix-point computation (due to aliasing) to find the proper memory areas and shapes. This leads to false positives and relatively long analysis times (e.g. 1 hour for 75k loc [137]). Our type checker is precise for accesses linear in the iteration indices (so no false positives and no missed errors), which are common. Furthermore, our analysis is intra-procedural, which makes our analysis more scalable and suitable to be part of the type system. In addition, previous bounds checking analysis cannot generically handle multi-dimensional, dynamic arrays precisely. Methods to recover linear relations for some instances of dynamic, multi-dimensional arrays are currently being researched [53].

For arbitrarily computed array accesses that cannot be well approximated by polytopes, abstract interpretation based methods may yield more precise results than our type system but these methods remain computationally too expensive to be used in a type system.

For certain problems Pluto [12] can move branches out of loops into the loop header and apply cache blocking optimizations to loop nests. As shown in Sec. 4.8, the control over the parallel decomposition enabled by **funkyImp** can help the programmer to match the automatic optimizations applied by Pluto. Moreover, in some cases the automatic transformation from Pluto are sub-optimal and may even lead to slowdowns. Thus, for consistently good performance, programmer needs full control over the decomposition. Moreover, the new syntax introduced by **funkyImp** enables the programmer to safely express generic and efficient computations using polytope-shaped decompositions of arrays (i.e. rows, columns and trace from the same matrix). This new concept of first class sub arrays extends the set of expressible pure programs and thus goes beyond the transformations performed by Pluto.

Leissa et al. [85] have proposed a language that automatically adapts the data layout of arrays to a hybrid between array of structures and structures of arrays, which is a commonly accepted best practice for vector operations on the CPU. Our language could benefit from this approach, alleviating the programer from choosing the best data layout.

X10 from [27] uses the partitioned global address space model to program distributed memory systems. Their type system supports unions of hyper cubes and relies on either run-time checks or dependent types with the associated problems. The language is imperative and the parallelization they offer is not automatic.

Chapel [26] allows to specify domains separately from their usage like **funkyImp** but without hierarchy or declarative array construction. The parallelization is non-automatic and no static safety guarantees comparable to funkyimp for array access are given.

These languages, along with C++11, python and many others allow to perform slicing

operations but no polyhedral decomposition of arrays.

Imperative languages can auto-parallelize only simple **array**-expressions as proving non-aliasing of different pointers is not generally possible. More complex operations must often be parallelized by the programmer (e.g. by placing openmp pragmas). Again, due to aliasing, these pragmas cannot be statically verified so that the produced code may contain data-races and dead-locks.

An old yet notable language is Alpha from [119]. The languages allows to declare arrays over polytopal domains but is limited to affine access relations. Due to these restrictions, several semantic preserving optimizations can be applied to the code. Still, the array structure (composition) is non-declarative and separated from types and the given safety guarantees beyond the optimizations as well as the performance in comparison to imperative languages is unclear.

[92] discussed several ways of incorporating data-parallelism into functional languages, including polytopal predicates. These polytopes are not part of the type system and can be seen as an extension to the way SAC handles iteration domains.

To the best of our knowledge, no existing language (pure or imperative) offers safe automatic paralellization of generic data-parallel operations in combination with the static in bounds safety guarantees for statically and dynamically sized arrays while achieving performance that is on par with parallel C++.

4.10 Conclusion and Outlook

We have presented the data-parallel subset of **funkyImp**, a pure functional language that introduces arrays subsets as first class citizens and allows to declaratively specify new arrays as a set of computations, each defining one part of the new array. **funkyImp** generalizes the view of an array being a hypercube of values to programmer-centric partitionings of the index set such as rows or upper/lower triangles defined by a set of polytopes that are parameterized by program variables.

We have introduced filters and lenses that allow for a declarative handling of index domains, thereby leading to generic, yet efficient and statically verifiable code. Furthermore, we have present a novel type system that employs the power of polyhedral solvers to statically ensure the absence of out-of-bound accesses and that all elements are defined exactly once.

We have presented experimental work that compares **funkyImp** with other relevant languages and highlights the performance benefits achievable by using our type system in a functional setting.

As a result, **funkyImp** makes it safe and easy for non-experts in C++ and data-parallel programming to write very concise, declarative code that is automatically transformed into correct parallel code. The produced code yields performance comparable to hand written data-parallel C++ code while the declarative source language is not subject to the high development effort and the pitfalls of parallel programming in C++.

Note that the data-parallel **array**-expressions introduced in this chapter are side-effect free and thus seamlessly integrate into the task graph analysis presented in Chap. 2. Furthermore, the task graph analysis can be applied to the lambdas passed into the **array**-expressions to find task-parallelism inside the data-parallel construct. If the task graph execution time prediction from Chap. 3 predicts a viable speedup from this task-parallelism then vectorization may be forfeited for a task-parallel execution.

The data-parallel language introduced in this chapter lends itself very well to automatically generate code for discrete data-parallel processors (like GPGPUs) by turning of array computations into OpenCL [54] kernels. In the presence of a discrete data-parallel processor, the processing environment becomes non-homogeneous and the cost of computing a task graph node becomes a function of the processor type it is computed on. Existing, static heterogeneous scheduling algorithms like [7] suffer from the lack of a realistic performance model which relates the theoretical schedule to the real world execution time. In his master's thesis Pöppl [118] has created a performance model that allows to predict data copy times (from host system to GPU and back) as well as execution times of OpenCL kernels on GPGPUs that is sufficiently precise to make realistic static scheduling decisions. As part of future work, the scheduling algorithm introduced in Chap. 3 should be extended to cope with heterogeneous environments based on Pöppl's performance model.

Chapter 5

Destructive Updates

In this chapter we will discuss side-effects in functional languages before we introduce uniqueness types into the **funkyImp** language in order to support referentially transparent, destructive updates on non-aliased data structures. Referentially transparent refers to the notion that all program values are immutable, so that no modification of any value may be observed [134] by a program. Note that referential transparency does not prohibit modification of values for optimizations or other purposes, as long as this modification is invisible to the program.

For a general purpose language, some form of side-effects and mutable state are required in order to communicate with the stateful hardware a program is running on. The C-program in Fig. 5.1 shows how the state of variable f is related to the file system and eventually the hard disk.

```
1 void main()
2 {
3  FILE* f=fopen("file.txt","w");
4  fwrite(f,"line1\n");
5  fwrite(f,"line2\n");
6  fclose(f);
7 }
```

Figure 5.1: Example program performing file I/O.

In the example, f is an opaque pointer that internally represents the state of a file on the hard disk together with the current read or write position. Any write operation necessarily modifies the state of the file and effectively destroys the previous state of the file. In the C-program, this is reflected by side-effects on the data f points to. Since there is no explicit flow between the write and close calls, our task graph construction from Chap. 2 would parallelize these calls. This would destroy the semantics of the program, because the program output may be modified by the parallelization which may reorder the side-effecting calls. Hence, generic side-effects would invalidate Lemma 1 and thus make semantic preserving, automatic parallelization hard.

Instead of unrestricted side-effects, which break referential transparency, Haskell uses monads. Monads [73] are abstract data structures which guarantee sequential execution while manipulating state. The I/O monad implicitly passes the state of the world, i.e. information describing the state of the universe including the state of the system the program is running on, into I/O performing functions which implicitly output the new state of the world and possibly some value. This scheme is referentially transparent in

principle. As there is no way to copy the state of the world, changes to the world are invisible to all functions that do not take the world state as input. Functions that take the world state as input must be primitives in the language, otherwise copies of the world state would become accessible to the programmer. There is no way to call any of these functions twice with the same input values, as the programmer cannot compare different world states and generally has no control over the state of the world. Therefore, these functions become referentially transparent by transforming the side-effects into an explicit computation on the world state and enforcing that computations on the world state are executed in the programmer specified order without giving explicit access to the world state.

One disadvantage of monads is that they specify a total order on **all** I/O operations, even if some of these operations are practically independent because the parts of the world state they access are mutually independent so that these independent operations would benefit from parallelization. Since the whole world state is treated monolithically, independent and thus parallelize-able operations cannot be inferred without precise knowledge about the effects generated by the I/O primitives.

In the following, we will discuss how uniqueness types [141] can be added to a side-effect free language so that mutations of non-aliased values can be implemented without violating referential transparency.

This chapter is structured as follows. In Sec. 5.1 we illustrate uniqueness types by applying them to file I/O. In Sec. 5.2 we discuss the effect of uniqueness types on the automatic parallelization provided by the core language from Chap. 2 and possible performance gains introduced by destructive updates. Afterwards, in Sec. 5.3, we formally introduce *uniqueness types* and *object orientation* into the **funkyImp** language. This allows us to define unique data structures which are derived from immutable data structures, so that casting inside the class hierarchy, as discussed in Sec. 5.4, can be used to switch between the mutable and the immutable view of a given data structure. In Sec. 5.5 we present the type hierarchy enabled by the immutable base type in combination with the unique and volatile attributes before we compare the performance of building a binary search tree using either destructive or non-destructive methods in Sec. 5.6. Finally, in Sec. 5.8 we summarize the effects of enhancing **funkyImp** with uniqueness types and object orientation.

5.1 File I/O using Uniqueness Types

Uniqueness is a type attribute which, for example, allows the programmer to define a `unique<String>` in addition to the usual, immutable `String`. The type system guarantees that at any time no more than one reference to a unique object exists and that a reference to a unique object can be used exactly once in the whole program before it becomes invalid. Fig. 5.2 illustrates a uniqueness type based version of the program from Fig. 5.1.

Here, in line 6, *fopen* returns a file handle h that is unique. Copying h into f in line 7 does not violate uniqueness because the reference h is invalidated by its use and h cannot ever be used in another expression without triggering a type error. Therefore, line 7 performs a move rather than a copy operation. In combination, line 8 and the uncommented line 9 would violate uniqueness because the unique handle f would be used twice. As the type system statically guarantees that a unique reference is used at most once, it is impossible for the programmer to access the value of a unique reference after

```
1 fopen : String->String->unique<FILE>
2 fclose : unique<FILE>->()
3 fwrite : unique<FILE>->String->unique<FILE>
4
5 test : ()
6 test =  let h:unique<FILE> = fopen "file.txt" "w" in
7    let f:unique<FILE> = h in
8    let f1:unique<FILE> = fwrite f "line1\n" in
9    (*let q:unique<FILE> = fwrite f "\n" in*)(*error, f referenced
     twice*)
10   let f2:unique<FILE> = fwrite f1 "line2\n" in
11   fclose f2
```

Figure 5.2: File I/O with uniqueness types

it has been used. This, in turn, allows to apply mutating operations to unique references without violating referential transparency. In order to apply a mutating operation, the only existing reference to the unique object must be used. After applying the operation the old state of the unique object is inaccessible because the only existing reference to it has been invalidated by applying the operation. Therefore, the *fwrite* operations in lines 8 and 10 can safely mutate the file handle.

5.2 Automatic Parallelization and Destructive Updates

Unlike monads, which handle side-effects implicitly, uniqueness types transform the side-effects into explicit data flow. Instead of a monolithic world state, each file operation depends explicitly on at most one previous file operation and there are no artificial dependencies on unrelated I/O operations that change the state of the world.

Since uniqueness types make all side-effects explicit, we can use the data flow analysis and parallelization methods from the previous chapters without any modification to correctly exploit all static task-parallelism including independent I/O operations. Note that uniqueness types cannot be used to implement communication between different threads or tasks because this would require multiple references (one per communicating task) to mutable memory. This problem will be discussed in Chap. 6.

Besides allowing a limited form of side-effects in a referentially transparent manner, uniqueness types allow to speedup computations in side-effect free languages. For example, unique arrays can be mutated rather than copied. This is implemented in Clean [116] which uses uniqueness types instead of monads in a language otherwise very similar to Haskell. In addition to arrays, they allow to define unique algebraic data structures, and the compiler tries to reuse invalidated unique objects instead of allocating new objects automatically. This can reduce or even eliminate the cost of garbage collection for unique objects. In Clean, the ability to mutate unique objects is not exposed to the programmer who must rely on the compiler optimizations.

For our **funkyImp** language we will take a different approach. In the following we will extend our language by uniqueness types and introduce the **upd** $x = e$ *in* y construct which allows to explicitly specify mutations by updating the unique field y in object x with the value of e. In object oriented languages these updates are usually denoted as $x.y = e$. Furthermore, we will introduce object oriented programming and classes (rather than algebraic data types that are not amenable to explicit casting). Using a

class hierarchy where the unique, mutable version of a data structure is derived from the immutable version, we allow the programmer to specify destructive updates and parallel, non-destructive updates on the same data structure. Casting inside the class hierarchy can be used to switch between the mutable and the immutable implementation. No observable side-effects are introduced into the language, as the mutable implementation can only be used on provably unique objects. After introducing the new language features we will present benchmarks showing the performance benefits.

Note that several methods to weaken the uniqueness requirements and thus to allow more operations on unique variables have been proposed in the literature, especially in the context of object oriented languages. Boyland [22] discusses methods to allow unique fields inside non-unique objects. As the object containing the field is not unique, multiple references to it may exist so that mutations of the unique field may be visible without additional constraints. We restrict our implementation of uniqueness types so that unique fields may exist only inside unique objects because we will introduce actors in Chap. 6 which allow to express mutations on shared state safely and without the cumbersome type annotations required for the weakened uniqueness requirements.

Another restriction of uniqueness types is the inability to express data structures with graph shape where each node is unique (e.g. a doubly linked list of unique nodes) since this requires multiple inbound references to a unique node which is not permitted by uniqueness. Clarke [29] discusses methods to extend uniqueness by capabilities which allow to specify external uniqueness. External uniqueness guarantees that there is at most a single reference to a graph shaped data structure.

We restrict our language to tree shaped recursive data structures by implementing plain uniqueness types as discussed in the following section. In addition, we will introduce shared mutable state in Chap. 6 that can be used to safely build and mutate graph shaped data structures.

5.3 Uniqueness Types vs. Object Oriented Programming

In this section, we discuss how uniqueness types are integrated into the **funkyImp** language. The type system needed to verify correct usage of unique values performs an intra-procedural, control flow sensitive analysis to count the syntactic occurrences of unique values. If any unique value is referenced more than once after it was initialized an error is reported. In addition, read-only occurrences of unique values that flow only into the result of the condition of an if expression are not counted inside the corresponding branch. This is permissible because the evaluation of the condition happens strictly before the branch is evaluated and the read-only access to the unique value is guaranteed to finish together with the evaluation of the condition. This is an example of borrowing [22], where multiple read-only references to a unique value can exist as long as it is guaranteed that the unique value cannot be modified during the life-time of these read-only references. The type rules required for uniqueness types have been described in detail by Baker [9].

The syntax of our language is extended as shown in Fig. 5.3. In addition to the *unique* type attribute, we introduce another form of temporal borrowing using the *volatile* type attribute which is introduced in Sec. 5.4.1 to transiently declare multiple read-only references to a single unique value. Finally, we introduce classes to allow the user to derive unique objects from immutable objects and thus switch between the mutable and immutable view of the same data structure using type casts.

$$
\begin{aligned}
\text{program } p &::= c^+ \\
\text{qualified value } q &::= x : qt \\
\text{qualified type } qt &::= (t \mid \textbf{unique}\!<\!t\!>) \\
\text{volatile type } vt &::= (qt \mid \textbf{volatile}\!<\!t\!>) \\
\text{type } t &::= \mathit{basetype} \mid \mathit{classname} \\
\text{class } c &::= \textbf{class } qt\ (\textbf{extends } \mathit{classname})\{q^*(qt\ q^* = e)?f^*\} \\
\text{fun type } ft &::= x : vt(\rightarrow vt)^* \\
\text{fun } f &::= ft\ x^+ = e \\
\text{exp } e &::= \textbf{let } x : vt(, x : vt)^* = e\ \mathit{in}\ e \mid \textbf{upd } x = e\ \mathit{in}\ x \mid e\ \mathit{op}\ e \mid \\
&\quad\ \ \textbf{if}(e)\ e\ \textbf{else}\ e \mid (e(, e)^*) \mid x.x\ e^* \mid v \mid \textbf{null} \mid \textbf{this} \mid qt\ e^* \\
\text{id } x &::= \ldots
\end{aligned}
$$

Figure 5.3: The language grammar extends the languages from Fig. 2.1 (and Fig. 4.2) by classes and uniqueness.

To keep the language specification small, we introduce default access modifiers so that all values inside a class are implicitly *protected* and all methods are implicitly *public*. Furthermore, we omit the C++ style template mechanism available in the full language implementation and enforce that non-primitive types are options in the sense that a value of class type is either *null* or a reference to an instance of the type. The full language implementation is discussed in Chap. 7. Finally, we introduce tuples into the language so that functions can return multiple values without having to declare a class encapsulating the values for every instance.

Similarly to Java, a class is declared by the *class* keyword followed by a qualified type. The class body allows to declare a set of qualified fields followed by an optional constructor which must have the same name as the class and a set of methods. Note that in contrast to most object oriented languages, all fields inside a class are immutable unless the class itself is unique and contains unique fields. Non-unique classes cannot contain unique fields. Analogously to Java, a class is constructed by applying the required arguments to the constructor and methods are called using the . operator so that $e_0.meth\ e_1\ ..\ e_n$ applies the arguments $e_0\ ..\ e_n$ to the function *meth* where e_0 is passed as implicit first argument called *this*.

The **upd** $x = e\ in\ y$ syntax is introduced to allow the programmer to specify explicit updates on unique objects. Here y must be a reference to a unique class instance that must contain a field with name x. The expression evaluates to y where the field x in the result is overwritten with the value of e. Note that all non-unique objects remain immutable. Fig. 5.4 shows an a non-unique binary tree implementation to illustrate the class syntax.

5.3.1 Immutable Binary Tree Example

The constructor in line 8 can use **upd** $x = e\ in\ y$ to set the values of the fields because freshly constructed objects are (implicitly) always unique. It is impossible that

```
1  class BinTree
2  {
3    left:BinTree
4    right:BinTree
5    val:int
6
7    BinTree: BinTree->int->BinTree->BinTree
8    BinTree l v r =
9      upd left=l in
10     upd right=r in
11     upd val=v in
12     BinTree
13
14   sum: int
15   sum =
16     let ls=left.sum in
17     let rs=right.sum in
18     ls+rs+val
19
20   add: int->BinTree
21   add v =
22     if (v = val) this
23     else if(v<val)
24       if(left=null) BinTree (BinTree null v null) val right
25       else BinTree (left.add v) val right
26     else if(right=null) BinTree left val (BinTree null v null)
27       else BinTree left val (right.add v)
28 }
```

Figure 5.4: Immutable binary tree.

any references exist to a freshly constructed object. The method *sum* sums all values inside the tree and *add* performs the usual construction of an immutable binary tree by creating a new spine from the old tree while reusing the leaves. On the one hand, *sum* can be efficiently task-parallelized with the methods introduced in the previous chapters. On the other hand, *add* offers no static task-parallelism and building a tree by chaining multiple applications of *add* performs many copy operations to unnecessarily retain the intermediate versions of the tree.

5.3.2 Uniqueness and Sub-Typing

We define a unique type `unique<T>` to be a subtype of type `T`. A unique type can be safely up-casted to an immutable type because this operation consumes the only unique reference available on the object so that it is impossible to perform destructive updates on the objects after the cast was performed. A down-cast from an immutable to a unique object cannot be performed by a simple update of the object's type because multiple references to the immutable object may exist, thus violating uniqueness. In order to guarantee uniqueness for a down-casted object, an expensive deep copy of the object would need to be created. By forcing the programmer to manually implement a deep copy rather than automating this process via down-casts, we protect the programmer from accidentally sacrificing performance by using an expensive down-cast which has a innocuous appearance.

5.3.3 Mutable Binary Tree Example

In order to avoid the unnecessary copying inherent to the non-destructive construction of an immutable tree, we introduce a unique, mutable binary tree. We exploit the sub-typing relation between unique and immutable types to allow the programmer to deal with the same data structure from both the unique and the immutable point of view. Therefore, the unique binary tree is derived from the non-unique class as shown in Fig. 5.5.

```
1  class unique<BinTree> extends BinTree
2  {
3    (*may overwrite attributes: (only unique classes may contain unique
       attributes)*)
4    left:unique<BinTree> (*must be same type, can change qualifier from
       immutable<t> to unique<t>, if class is unique!*)
5    right:unique<BinTree>
6    val:int (*optional (not modified, inherited from BinTree)*)
7    (*could add additional (possibly unique) attributes*)
8
9    (*methods from super are not available if class is unique*)
10   unique<BinTree>:unique<BinTree>->int->unique<BinTree>->unique<
     BinTree>
11   unique<BinTree> l v r=
12     upd left=l in
13     upd right=r in
14     upd val=v in
15     unique<BinTree>
16
17   (*must also return this, or unique ref is lost*)
18   sum: (int,unique<BinTree>)
19   sum res =
20     let (ls:int,l:unique<BinTree>)=left.sum in (*must update left (
       unless borrowed)*)
21     let (rs:int,r:unique<BinTree>)=right.sum in (*must update right (
       unless borrowed)*)
22     (ls+rs+val, let left=l in let right=r in this)
23
24   add: int->unique<BinTree>
25   add v =
26     if(v = val) this
27     else if(v<val)
28       if(left=null) upd left = (unique<BinTree> null v null) in this
29       else upd left = (left.add v) in this
30     else if(right=null) upd right = (unique<BinTree> null v null) in
       this
31       else upd right = (right.add v) in this
32 }
```

Figure 5.5: Unique binary tree.

When a non-unique class inherits from another non-unique class we apply the inheritance rules from Java [112], so that non-private fields and methods from the base class are usable by the derived class. In order to preserve the relation between immutable and unique types in the class hierarchy, we use different inheritance rules for shadowed fields. Shadowing of fields refers to the scoping rules used to resolve the ambiguous access of fields with identical names in a class hierarchy. Given a base class B which defines a field f and a derived class D which also defines a field f, then D inherits the field f from

B and thus contains two fields named f. This ambiguity is usually resolved by name shadowing. Inside the functions of class D, a reference to f resolves to the field defined in D because it is closer in the class hierarchy. In the context of class D, the field f from D is said to shadow the field f from class B. Usually (e.g. in Java and C++), the shadowing fields and the shadowed fields define separate memory locations which can be distinguished using the scoping rules or explicit addressing (e.g. using *super* in Java).

In our language we change this notion of shadowed fields. Our type system enforces that the shadowing field's type is a sub-type of the shadowed field's type and the memory locations of both fields are defined to be identical. Hence, the memory locations of fields $left, right$ and val in Fig. 5.5 are co-located with the memory location of the fields from the base class but their type may be changed to a sub-type. This is used to make fields $left$ and $right$ unique inside the class `unique<BinTree>` without changing their memory location.

In addition to redefining shadowing, we remove the methods from the base class from the scope of derived unique classes, which is also non-standard. The methods from the non-unique base class cannot be used inside the unique class because they do not respect uniqueness of the fields $left$ and $right$ and of the function parameter $this$. For example, the methods from the immutable `BinTree` may generate and store additional references to the unique parameter $this$ in another unique method parameter and thus violate uniqueness of $this$. Instead, the programmer can supply methods that exploit the uniqueness to destructively perform operations as shown by the *add* method in line 25 of class `unique<BinTree>`.

Now the same binary tree data structure can be manipulated using either destructive or non-destructive updates by casting. Note that in a unique class, the type system must enforce that all unique fields which are referenced in a function must be overwritten before the function returns in order to restore uniqueness of the fields when a function is left. This is illustrated in line 22 of Fig. 5.5, where the field $left$ and $right$ must be overwritten because they had been accessed in the function. Furthermore, note that uniqueness is transitive so that a reference to a unique class (e.g. using $this$) implicitly references all unique fields reachable from the class reference (e.g. $this.left.left$).

5.4 Casting

A typical use case of the class hierarchy we have defined for binary trees is shown in Fig. 5.6. Here, a unique tree is constructed destructively. After construction is finished, the uniqueness is up-casted away to allow multiple and potentially parallel read operations on the tree. In Sec. 5.6, we compare the performance of destructive and immutable tree construction.

In the example, the usage of the tree data structure can be separated in two phases, namely sequential destructive updates followed by parallel reads (including non-destructive updates).

After the up-cast no more destructive updates of the data structure are possible because, in general, down-casts are not permitted. Without further modification of the language, the only way to recover uniqueness for a non-unique object is to create a unique deep copy, which is expensive. In the following we will introduce the *volatile* type attribute which allows safe borrowing [22], i.e. uniqueness is safely recovered after a phase where uniqueness was given up for transient, parallel readers.

```
1 class main
2 {
3  main: ()->()
4  main =
5    let ubt:unique<BinTree> = (((unique<BinTree> null 0 null).add 5).
     add -2) ... (*destructively build tree*)
6    let usum:int, ubt2:unique<BinTree> = ubt.sum in
7    let bt:BinTree = ubt2 in (*up cast removes uniqueness requirement
     *)
8    let res1:BinTree = g bt in *(process tree in parallel*)
9    let res2:int=h bt in (*cannot ever mutate ubt2 again*)
10   res1.sum+res2
11 }
```

Figure 5.6: Destructive tree construction followed by up-cast to an immutable tree.

5.4.1 Transient Immutability

The *volatile* type attribute allows the programmer to specify that he intends to transiently remove mutability to allow multiple parallel readers before recovering mutability, so the behavior is similar to a reader writer lock [102]. Hence, the *volatile* attribute allows to interleave phases of sequential mutation of a data structure (or even parallel mutation, see Chap. 6) with phases of parallel read-only access to the same data structure. In the read-only phase, the new data required to update the data structure can be computed in parallel, before it is applied to update the structure afterwards without the possibility of introducing data-races or dead-locks. Fig. 5.7 shows an example of transient immutability using the *volatile* attribute.

```
1 class main
2 {
3  main: ()->()
4  main =
5    let ubt:unique<BinTree> = (((unique<BinTree> null 0 null).add 1).
     add 2) ... (*destructively build tree*)
6    let (usum:int,ubt2:unique<BinTree>) = ubt.sum
7    let bt:volatile<BinTree> = ubt2 in (*up cast temporarily removes
     uniqueness requirement*)
8    let res1:volatile<BinTree>=g bt in (*data flow from bt to res1*)
9    let res2:int=h bt in (*parallel transient reader*)
10   let res1sum:int=res1.sum in (*cannot call res1.sum in line 11
     without creating cyclic dependency on down-cast*)
11   let capture:unique<BinTree> = bt in (*wait for all references to
     objects that are data flow dependent on bt to vanish*)
12   (capture.add (res1sum+res2)).sum
13 }
```

Figure 5.7: Transient immutability using the volatile type attribute.

Similar to the previous example up-casting is used in line 6 to remove the uniqueness requirement but the result is stored in a `volatile<BinTree>` instead of an immutable `BinTree`. Then, several *volatile* read-only references can be used before a *volatile* reference is down-casted to a unique reference in line 10.

5.4.2 Task Graph Extension for Volatile Types

Fig. 5.8 shows the part of the task graph obtained from the code in Fig. 5.7 which contains the volatile references. We must guarantee that all *volatile* readers have finished reading before the down-cast is performed. To achieve this, we augment the task graph with additional edges so that the down-cast becomes dependent on all expressions that use a *volatile* reference to the corresponding data structure in the following way.

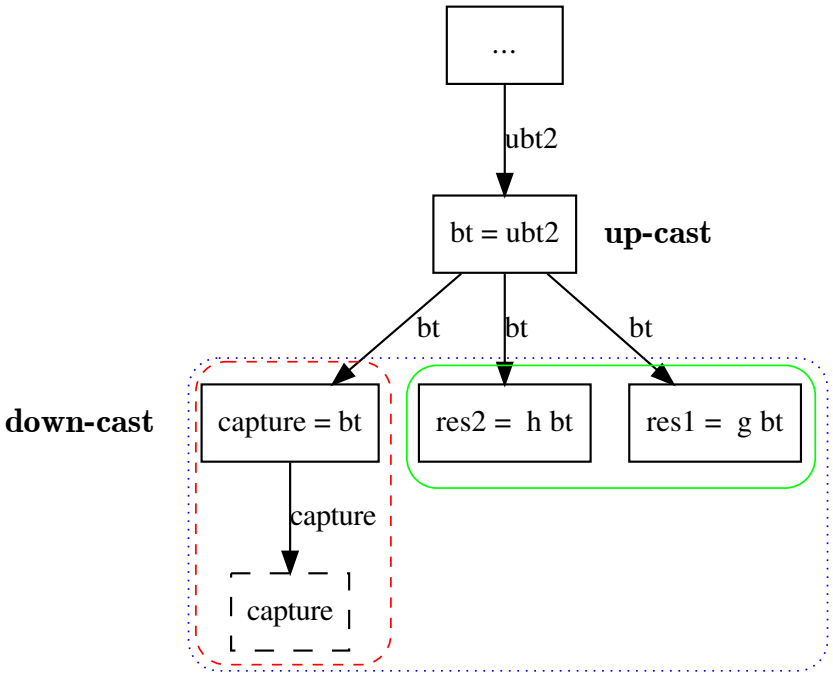

Figure 5.8: Partial task graph obtained from Fig. 5.7.

Effectively, we need to add edges from all nodes that depend on a volatile reference to the down-cast node so that all reads must finish before the down-cast. The nodes that depend on the volatile reference are those nodes which are reachable from the up-cast, shown in the dotted region in Fig. 5.8. From these reachable nodes, we have to remove those nodes which are exclusively reachable from the down-cast, shown as dashed region in Fig. 5.8, because these do not depend on the volatile reference any more, instead they depend on the unique reference generated by the down-cast. The remaining nodes, shown in the solid region in Fig. 5.8, are those nodes that really depend on the volatile reference generated by the up-cast. Therefore, we add an edge from each remaining node to the down-cast in order to guarantee that all reads on volatile references finish before the down-cast. The result of this operation on Fig. 5.8 is shown in Fig. 5.9.

Furthermore, we only allow the specific volatile reference created by the up-cast, i.e. *bt* in Fig. 5.7, to be down-casted and verify that it is down-casted at most once, so that only one unique reference to the data structure can be recovered. If the newly added edges create cycles in the task graph, then the program is rejected by the compiler because the down-cast depends on itself. For example, such a cyclic dependency could created by using $res1.sum$ instead of $res1sum$ in line 11. In this scenario, the dashed node representing line 11 in Fig. 5.8 would be directly connected to the node computing $res1$ and thus this node would not be exclusively reachable via the down-cast. As a consequence, the task graph in Fig. 5.9 would be extended by an edge from the dashed node to the down-cast node, introducing a cycle into the graph.

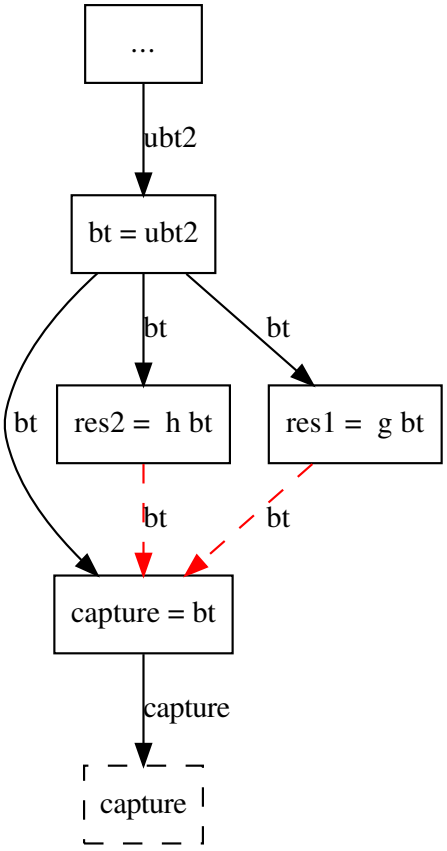

Figure 5.9: Partial task graph obtained from Fig. 5.7 including additional dependencies (dashed edges) from volatile references to the down-cast.

After extension of the task graph, as described in this section, it is guaranteed that the down-cast waits for all nodes that use a volatile reference to the unique object. A down-cast may be delayed infinitely, if any of the readers do not terminate. The compiler tries to protect the programmer from such a mistake by disallowing to pass volatile references into recursive functions, unless these are explicitly marked as terminating by the programmer, as discussed in Sec. 7.3.2.

Note that volatile references may be applied to any function accepting immutable references as long as all transitive flow inside the function from the volatile argument to the function result happens only via primitive types, which are copied rather than referenced, or control flow so that the function result cannot contain references into the volatile argument. This guarantees that the volatile argument is unreferenced when the outer most function returns so that the volatile argument can be safely down-casted later on.

5.5 Type Hierarchy

In this section, we summarize the relationship between the different type attributes introduced in this chapter. Fig. 5.10 shows the relation between immutable, volatile and unique types where black arrows indicate a sub-type relation and dashed arrows indicate that casts in the direction of the arrow are permissible. Note that volatile references can only be down-casted to a unique reference but not up-casted to an immutable reference. This is the case because we cannot regain uniqueness from an immutable reference, as

discussed in Sec. 5.3.2.

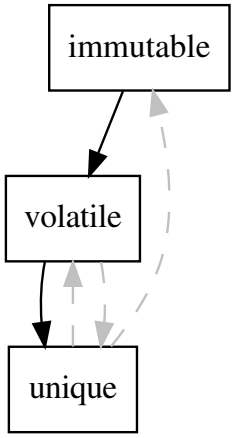

Figure 5.10: Relation between immutable, volatile and unique types.

In the next section, we will show the actual performance benefit of using destructive updates in contrast to non-destructive updates.

5.6 Experimental Results

Destructively building a binary tree as defined in Fig. 5.5 with $1 \cdot 10^6$ random nodes is about 500% faster than building the same tree using the non-destructive code from Fig. 5.4 on a Nehalem Intel Core i7 CPU 860 at 2.80GHz. Generally, uniqueness types reduce the pressure on the garbage collector because many objects are reused through mutation rather than discarded. Thus, strong performance gains may be achieved by using uniqueness types.

5.7 Double Buffering

Fig. 5.11 shows a typical loop working on array data iteratively. The **array** expression allocates a new array for every iteration although the old array is dead after the computation and could thus be reused.

```
1 iter:int->int->int{A,B}[d1{N}]->int{A,B}[d1{N}]
2 iter n m data =
3  if(n>=m) (*convergence criterion*)
4    data
5  else
6    let newdata:int{A,B}[d1{N}]=array d1{N} (d1{N},\(i)->data[i]/2) in
7    iter (n+1) m newdata
```

Figure 5.11: Iterative computation on an array unnecessarily reallocating arrays.

In order to avoid the superfluous array allocations, double (or higher order) buffering techniques can be implemented using uniqueness types in combination with an **array** expression that accepts a unique array as target and thus does not need to allocate a new array, as shown in Fig. 5.12.

```
1 iter:int->int->unique<int{A,B}[d1{N}]>->unique<int{A,B}[d1{N}]>->
  unique<int{A,B}[d1{N}]>
2 (*note: arrays are unique, content is not*)
3 iter n m b1 b2 = (*read b1, write result to b2 and swap, due to
  uniqueness b1!=b2 must hold*)
4  if(n>=m) (*convergence criterion*)
5    b1
6  else
7    let vb1:volatile<int{A,B}[d1{N}]>=b1 in
8    let alias:unique<int{A,B}[d1{N}]>=array b2 (d1{N},\(i)->vb1[i]/2)
9    in iter (n+1) m alias ((unique<int{A,B}[d1{N}]>)vb1)
10   (*b2alias is array b2 overwritten by the array expression*)
```

Figure 5.12: Iterative computation using double buffering.

Here, two unique arrays are allocated by the original caller of function `iter` and are used for in an alternating sequence for reading from and writing to.

5.8 Conclusion

In this chapter, we have shown that classes combined with uniqueness types are a powerful tool to express mutations of data structures which are unshared or shared only in a precisely defined scope without breaking referential transparency. Transient mutability creates the possibility to interleave program phases where multiple readers can access the unique data structure with phases where a single mutator can safely act on the data structure. These mechanisms allow to implement program patterns commonly found in imperative, deterministic parallel programs with similar performance. In Chap. 6, we will extend our language to allow non-deterministic, parallel computations on full graphs rather than the tree shaped data structures supported by basic uniqueness types.

Chapter 6

Shared Mutable State

In this chapter we will introduce shared mutable state into our **funkyImp** language. Shared mutable state is needed to communicate between concurrent program parts, i.e. process asynchronous messages, and adds non-determinism to the language. In order to cope with the non-determinism we will switch from denotational to collecting semantics and define the semantics of a program as the set of traces the program may generate. This will allow us to show that **funkyImp** programs, including shared mutable state, are dead-lock and data-race free.

This chapter is structured as follows. In Sec. 6.1, we revisit the cause of data-races by discussing a simple C program and some assembly fragments generated thereof. To do so, we use a simplified model of parallel execution to give some intuition into the relation between trace based semantics of assembly instructions and data-races. Afterwards, we discuss mutual exclusion and locks as methods to prevent data-races in Sec. 6.2. Readers familiar with these concepts may skip the first two sections. Note that we will later introduce traces and data-races for our language, extended by shared mutable state using actors, more formally yet analogously to the first two sections.

In Sec. 6.3 we introduce actors by contrasting them to shared memory as discussed in the first section, before we give some intuition into how actors interact with the rest of our language. Afterwards, in Sec. 6.4, we introduce our actor language grammar, before we use examples to discuss how our flavor of actors, which employs message barriers at the end of functions, can be used to communicate ordered messages between asynchronous tasks in Sec. 6.5. Next, in Sec. 6.6, we introduce the **sync** keyword, which allows to compose and synchronize messages on sets of actors while retaining the property that our language is dead-lock free. In Sec. 6.7, we discuss the formal semantics of our actor language before we present how this semantics can be efficiently implemented using different hardware synchronization mechanisms in Sec. 6.8. Finally, in Sec. 6.10, we formally define data-races analogously to Sec. 6.1 and show that our **funkyImp** language, extended by actors, is data-race free.

6.1 Data-Race Example

Before formally defining data-races in Sec. 6.10, we will discuss the following example in order to give some intuition into the problem. In the sequential version of the program, shown in Fig. 6.1, the global variable c is initialized with 0 and incremented twice so that the result is 2. In the parallel version, shown in Fig. 6.3, two threads are started, where each thread executes the function inc_fun which, in turn, increments the global variable

c. Since the state of variable c is mutable and accessible from more than one thread, the variable is said to have *shared mutable state*.

```
int c=0; //init global variable

int main()
{
  c=c+1; //increment (c=1)
  c=c+1; //increment (c=2)
  return c; //return result
}
```

Figure 6.1: Sequential counting.

Inside the sequential program, the statement `c=c+1;` has a clearly defined semantics. It increments the value stored in the memory location with name c by one. On most RISC [74] processors, this message will be implemented in a fashion similar to Fig. 6.2. Here, the content of the memory location named c is read into a register ebx, as shown in the first two lines of the assembly listing. Then, one is added to the value of the register before the result is written back to memory. The semantics of assembly code, especially of memory messages in the context of multiple threads, is quite complex and hardware dependent [69]. A formal presentation of these semantics would mostly distract from the actual problem. Therefore, we will assume the following, simplified execution model for assembly instructions, which is sufficient to discuss data-races. In this section, we assume that assembly programs are defined by a list of assembly instructions, which are executed atomically from top bottom. Concurrently running assembly programs are executed by interleaving the instructions from all threads into a single assembly program, so that instructions from a single thread are still executed from top to bottom but may be separated by instructions from other threads. This interleaving of instructions gives raise to many different execution orders of the instructions from different threads.

```
mov eax,0x62306c          ;move address of c into eax
mov ebx,DWORD PTR[eax]    ;load value of c from memory into ebx
add ebx,0x1               ;add 1
mov DWORD PTR[eax],ebx    ;store result to memory
```

Figure 6.2: Assembly listing of the source line $c = c + 1$;.

The programmer can ignore all assembly-level and hardware-level details when reasoning about the semantics of the statement in a sequential program. This changes, when the global variable c is incremented concurrently, as shown in Fig. 6.3.

In the current C++11 standard, the semantics of a program fragment that reads from a memory location which might be concurrently written is undefined [52]. Therefore, the semantics of the statement `c=c+1;` is not retained by the parallelization shown in Fig. 6.3. This is the case, because the hardware executes the statement in several smaller steps rather than in a single, un-interruptable step. Both threads execute the same code from Fig. 6.2 as before but due to the parallel execution, any interleaving of the instructions may be observed.

As illustrated in Fig. 6.4 (a), the code behaves as expected if the instructions of one thread happen to finish completely before the instructions from the other thread are executed. This interleaving is equivalent to a sequential execution of both statements.

```
int c=0;
void *inc_fun( void *ptr ){c=c+1; return null;}

int main()
{
 pthread_t thread1, thread2;

 pthread_create( &thread1, NULL, inc_fun, NULL);
 pthread_create( &thread2, NULL, inc_fun, NULL);

 //wait for both threads to finish
 pthread_join( thread1, NULL);
 pthread_join( thread2, NULL);
 return c;
}
```

Figure 6.3: Parallel execution of $c = c + 1$;.

In contrast, Fig. 6.4 (b) shows an interleaving where both threads read the value from memory into a separate register before any one thread has written the result of its computation back to memory. In this case, the side-effect of one of the statements is effectively lost because both threads are racing to read and write the data. This shows that, even under our simplified execution model, data-races can be produced by concurrent reads and writes to memory.

In order to give some intuition into the relation between data-races and traces, we

```
mov eax,0x62306c            ;T1:move address of c into eax
mov ebx,DWORD PTR[eax]      ;T1:ebx=0
add ebx,0x1                 ;T1:ebx=1
mov DWORD PTR[eax],ebx      ;T1:c=1
mov eax`,0x62306c           ;T2:move address of c into eax`
mov ebx`,DWORD PTR[eax`]    ;T2:ebx`=1
add ebx`,0x1                ;T2:ebx`=2
mov DWORD PTR[eax`],ebx`    ;T2:c=2
```

(a) Sequential

```
mov eax,0x62306c            ;T1:move address of c into eax
mov ebx,DWORD PTR[eax]      ;T1:ebx=0
add ebx,0x1                 ;T1:ebx=1
mov eax`,0x62306c           ;T2:move address of c into eax`
mov ebx`,DWORD PTR[eax`]    ;T2:ebx`=0
mov DWORD PTR[eax],ebx      ;T1:c=1
add ebx`,0x1                ;T2:ebx`=1
mov DWORD PTR[eax`],ebx`    ;T2:c=1
```

(b) Interleaved

Figure 6.4: Two single threaded programs emulating different interleavings of the assembly instructions from two threads T1 and T2 executing inc_fun from Fig. 6.2. Registers of thread T1 are called eax and ebx, registers of thread T2 are called eax' and ebx'. To simplify the discussion, we assume that the instructions in a assembly program are executed atomically and from top to bottom.

define a trace of assembly programs containing only the variable c as a list that initially contains as first item the value 0 and as further items the state of c after execution of each assembly instruction which modified the value of c. We define a trace set to be the set that contains all possible traces that the program may generate. In general, it can be computed by building all possible interleavings of the assembly instructions generated by the program. The trace set for the sequential program is given by $\{\langle 0, 1, 2\rangle\}$, because the program deterministically counts from zero to two. Since we assume atomic execution of assembly instructions, the trace set for the parallel program is given by $\{\langle 0, 1, 2\rangle, \langle 0, 1\rangle\}$. The additional trace is due to the potentially lost update illustrated in Fig. 6.4 (b). The difference in the trace set shows that the parallelization given in Fig. 6.3 introduces a data-race and thus does not retain the semantics of the sequential program.

Note that, in reality, assembly instructions are not guaranteed to execute atomically. On current Intel hardware, a single assembly instruction usually translates into several micro-messages performing the individual memory reads and writes that constitute a single assembly instruction. In general, these micro-messages may interleave in a similar fashion to assembly instructions. It is possible to replace line 2 and 3 from Fig. 6.2 by a single assembly instruction `add DWORD PTR[eax],0x1`. This would remove the data-race under our simplified execution model but the data-race would remain under the full execution model for assembly instructions because the instruction is not guaranteed to execute atomically.

6.2 Mutual Exclusion

Modern hardware supports a set of messages and methods to protect shared mutable state from data-races. The commonly available protection mechanisms enable the programmer to explicitly guard specific program fragments (called critical sections) from concurrent execution with other critical sections using a shared guard. Fig. 6.5 shows how one of these mechanisms called lock can be used as a shared guard to protect the statements `c=c+1;` from executing concurrently.

```
pthread_mutex_t mutex; //initialized else where

void *inc_fun( void *ptr )
{
  pthread_mutex_lock(&mutex); //enter critical section
  c=c+1; //increment global variable c
  pthread_mutex_unlock(&mutex); //leave critical section
  return null;
}
```

Figure 6.5: Shared memory protected by a lock.

One can imagine the lock to be a pointer to a flag which indicates (if set) that a thread is currently executing code that should be guarded from concurrent execution. The call to *pthread_mutex_lock* indicates the beginning of the critical section. *pthread_mutex_lock* checks the state of the flag. If it is not set, the thread sets the flag and continues execution of the guarded code. *pthread_mutex_unlock* indicates the end of the critical section and resets the flag. Note that the message that reads the flag and sets it in case it was not set must be executed as a single, atomic message to avoid

a data-race on the flag itself. On current Intel hardware this can be implemented by the available atomic compare and swap message [69]. In addition, it must be guaranteed that all memory messages from the critical section are guaranteed to execute between the flag set and unset messages. Both, compiler optimizations and the out-of-order execution [65] of the processor may reorder instructions. Therefore, the corresponding compiler optimizations must be restricted and special memory fence [18] instructions must be generated to restrict out-of-order execution.

If the flag is already set when a thread tries to enter the critical section using *pthread_mutex_lock* then the thread must wait until the flag becomes unset before it can continue so that concurrent execution of the guarded sections is effectively prevented. The semantics of the $c = c+1$; statements is restored by manually placing locks, as shown in Fig. 6.5. In other words, locks sequentialize the protected messages and thus remove parallelism. When comparing the data-race free parallel implementation in Fig. 6.5 to the sequential implementation in Fig. 6.1 it is apparent that this parallelization cannot gain any parallel speedup, instead the overhead from thread signaling and locking will make the parallel version slower than the sequential version. Basically, we have exchanged a fast sequential execution for a slower sequential execution where the execution order of the critical sections is non-deterministic.

Virtually all low-level concurrency protection methods available on current desktop hardware can be replaced by a variant of locks without changing the semantics of race free programs that use these protection methods, so from a conceptual point of view a discussion of locks should be sufficient. Nevertheless, the performance characteristics exposed by the different protection methods allow to expose varying levels of parallelism when operating on shared data structures. Therefore, we will discuss the different protection methods in detail in Sec. 6.9.

Programming with locks tends to be error prone. In an imperative language like C++, the programmer must manually find and explicitly lock all concurrent read/write messages on potentially shared memory. Any oversight may result in hard to detect, silent failures of the program which cannot be eliminated by testing alone.

Furthermore, nesting of critical sections can lead to non-deterministic non-termination through *dead-lock*. A dead-lock may occur when different threads acquire the same locks in different order. Since the sequence of acquire statements is non-atomic, each thread may acquire the respective first lock before both threads are stuck waiting for the other lock, which was already acquired by the respective other thread.

In order to minimize the sequentializing effect of locks on parallelism, a larger shared data structure may be decomposed into multiple connected sub-structures where each sub-structure is protected by its own lock and may be manipulated in parallel to the other sub-structures. Correctly orchestrating multiple subsets of related locks is even more complex and error prone than using a single, coarse-grained lock for the whole structure [39] thus exacerbating the shortcomings of explicitly using locks.

In the rest of this chapter we will define a variant of actors in order to introduce shared mutable state into our language and we will show that the thusly extended language remains data-race and dead-lock free.

6.3 Actors vs. Shared Mutable State

An actor can be seen as a mutable, stateful object that receives asynchronous messages. Different, concurrent processes may possess references to a shared actor but cannot di-

rectly access the state of that actor. Instead, they may indirectly read and write the state of the actor by sending messages to it, which are processed strictly sequentially. Therefore, each actor can be seen as a reference to shared mutable state that is implicitly protected from concurrent access.

6.3.1 Referential Transparency and Inter Thread Communication

Although actors guarantee that the access to their state is atomic, referential transparency is lost because changes of an actor's state may become visible through shared references to the actor. Since we intend actors to be used to communicate between different tasks and threads, it is indeed essential that changes of state can be communicated via shared references. Generally, actors introduce bounded (choice based) and unbounded non-determinism (e.g. fair merge [31]) into our language at the cost of referential transparency.

6.3.2 Actors, Object Orientation and Uniqueness

Since actors encapsulate state and tie the functions that operate on the state to an interface, actors are closely related to objects in object oriented programming. Actors allow asynchronous function calls but enforce stricter encapsulation since public data does not exist. Thus, actors unify the notion of objects with concurrency [3].

Uniqueness types, as introduced in Chap. 5, statically guarantee that a reference to a unique object exists only at a single program point and cannot be shared. In contrast, references to actors can be shared without restriction but the serializing interface of each actor guarantees dynamically that at any time at most one reference to the actor is used so that any user of such a reference always sees a consistent state. Actors may contain references to unique objects because the state inside an actor is private. Therefore, uniqueness is not violated by multiple external references and concurrent (non-unique) write messages are prevented by the sequentializing interface of the actor. Note that read-only operations on unique types do not exist. Just like uniqueness types, the actor may support multiple parallel read messages on non-unique fields.

Note that the compiler forbids to pass volatile references into any actor message since this would make down-casts, required to regain uniqueness from volatile references, unsafe. As discussed in Chap. 5, for the down-cast to be performed, all readers of a volatile reference must have finished reading. When such reads are deferred by sending the volatile reference to an actor, the down-cast may also be deferred arbitrarily because an arbitrary amount of time may pass before the message is processed.

In order to forbid passing of volatile references into actor messages, the compiler computes for every function if there is any data flow from an immutable parameter into an actor message, because, in general, the type system allows to apply volatile references as subtype of immutable ones. If any such flow is found, the function is flagged as unfit to be used with volatile references. The compiler rejects the application of a volatile reference to a function f, if any unfit function is reachable from f in the call graph.

6.3.3 Actors and Non-determinism

In general, messages sent to an actor are not guaranteed to be processed in any particular order. Therefore, actors introduce non-determinism into our language. Similar to the I/O

monad discussed in Sec. 1.4.3, actors effectively encapsulate non-determinism and side-effects. We will show in Sec. 6.10 that, unlike monads, our variant of actors does not violate the key property of our language, stated in Lemma 1, which keeps the semantics invariant under parallelization. We will show that this is possible due to the non-determinism concerning the order of actor messages. This non-determinism allows to reorder most messages on actors without changing the possible outcome and thus the semantics of the program. In this respect, actors can be seen as non-deterministic memory cells where all reads and writes are atomic. The state of the memory cell is known only during the time after initialization and before the first update message on the cell has been scheduled. It is guaranteed that the message will eventually commit, but it is undefined when (in absolute time or even in relation to other actions on the same actor). In general, the programmer cannot make any assumptions about the current state of an actor when the program execution is outside of an atomic message.

Several methods to reason about actors and actor networks have been proposed. For example, power domains over actor message diagrams [31] may be used to infer invariants about actor networks using abstract interpretation. Here, a fix-point computation is required to approximate all possible interleavings of causally dependent actor messages. So while actors are powerful and safe from data-races, as we will show in Sec. 6.10, the complexity of dealing with non-determinism (and thus actors) should be avoided unless an equally efficient, deterministic solution is unfeasible.

6.3.4 Actors and Dynamic Parallelism

Nevertheless, actors allow to express computations over shared mutable data structures. In principle, a network of actors can be interpreted as a graph shaped mutable data structure where each individual node in the graph has a consistent state but groups of nodes are not consistent. Invariants concerning more then one node must be carefully protected from user interference, e.g unwanted messages entering the network at the wrong time or place, and may hold only after a specific state has been reached. Thus, actors enable the potential performance benefits as well as most of the complexity from fine-grained locking of parallel data structures [39]. In this way, actors can be used to build dynamic task graphs in contrast to the static task graphs introduced in Chap. 2, where the control flow depends on some stateful history. These task graphs must be specified explicitly by the programmer since actors are created and messages are sent explicitly. Furthermore, actor networks cannot be statically optimized using the methods from Chap. 3.

6.3.5 Use Cases

Summarizing, we see that although actors are data-race and dead-lock free, actors create the need to perform explicit task management and expose some problems previously not existing in our language. Nevertheless, actors are required to make our language general purpose, so that the following use cases can be handled:

- Inter thread/task communication: For example, asynchronous keyboard or network messages flowing into a program via a callback shall be communicated from the callback to an asynchronous program part.
- Non-deterministic parallel algorithms (e.g. iterating the nodes of a directed acyclic graph in topological order)

- Mutable graphs: uniqueness types can express only unique trees since a graph would require multiple inbound references to a mutable node. Externally unique objects [29] can model a graph with only a single external reference but not multiple external references.

Before discussing examples, we will introduce the syntax of our variant of actors.

6.4 Actor Language

$$
\begin{array}{rcl}
\text{configuration } \sigma & ::= & \langle M; Q; C; \langle e, r\rangle\rangle \\
\text{memory } M & ::= & [m_1 \to \langle v_1, \lambda x \to e_1, \mathsf{true} \vee \mathsf{false}, \mathsf{true} \vee \mathsf{false}\rangle, .., \\
 & & m_n \to \langle v_n, \lambda x \to e_n, \mathsf{true} \vee \mathsf{false}, \mathsf{true} \vee \mathsf{false}\rangle] \\
\text{queue } Q & ::= & [m_1 \to \langle \mathcal{P}(e), .., \mathcal{P}(e)\rangle, .., m_n \to \langle \mathcal{P}(e), .., \mathcal{P}(e)\rangle] \\
\text{composition } C & ::= & [m_1 \to \langle m_1^1, .., m_{l_1}^1\rangle, .., m_l \to \langle m_1^l, .., m_{l_l}^l\rangle] \\
\text{expression } e & ::= & \textbf{update } e\ \lambda x \to e \mid \textbf{query } e\ \lambda x \to e_r \mid \textbf{put } e\ e \mid \textbf{get } e\ \lambda x \to e_r \mid \\
 & & \textbf{cancel } e \mid \textbf{resume } e\ e \mid \textbf{finally } e \mid \textbf{new } e\ \lambda x \to e_h \mid v \mid \\
 & & \textbf{unlock } m \mid \textbf{sync } e_1\ ..\ e_m \mid \textbf{raise } e\ e\ e \mid \langle e, .., e\rangle \mid x \mid e\ e \\
\text{value } v & ::= & () \\
\text{eval contexts } E & ::= & E\ e \mid e\ E \mid \textbf{query } E\ (\lambda x \to e_r) \mid \textbf{put } E\ e \mid \textbf{put } e\ E \mid \\
 & & \textbf{raise } E\ e\ e \mid \textbf{raise } e\ E\ e \mid \textbf{new } E\ e \mid \textbf{get } E \mid \langle .., E, ..\rangle \mid \\
 & & \textbf{cancel } E \mid E\ \textbf{finally } e \mid \textbf{sync } E \mid \textbf{resume } E\ e
\end{array}
$$

*Figure 6.6: Language grammar extending **funkyImp** by actors.*

The grammar of our actor language is shown in Fig. 6.6, the formal semantics will be introduced later in Sec. 6.7, after we have given some intuition into the capabilities of our actor language using examples.

Actors introduce state into our language, so that the evaluation of an expression e depends on an environment $M; Q; C$. Within the environment, M maps an actor reference to its state $\langle v, \lambda x \to e, f_1, f_2\rangle$ which consists of the value v, the continuation $\lambda x \to e$, the flag f_1 indicating whether the actor is locked for an update and another flag f_2 indicating whether the actor is locked for a continuation. Continuations will be discussed later, in Sec. 6.5.2. Q maps an actor reference to its scheduled asynchronous messages. Note that Q stores scheduled messages in tuples of sets rather than simple sets so that the messages in the sets can be separated by barriers which induce a partial order on the messages. Messages left of a barrier must execute before the messages on the right hand side of a barrier. Messages inside the individual sets are unordered with respect to each other. In Sec. 6.5 we will show how barriers can be used to guarantee a partial order between specific sets of scheduled messages. Finally, C describes compositions of several actors into a single actor by mapping an actor reference to a vector of actors. All messages on actors take a reference to an actor as input and perform a lookup in C, to retrieve the vector of actors the message should be applied on. Thus, new actors are created using **new**, which expands the domain of M by a new actor and creates a mapping from the

new actor to itself in C. Several actors can be composed into a single one by using **sync**, which creates a new actor and a mapping from this actor to the actors that shall be composed in C.

This setup allows to express the state of an actor program by a configuration $\langle M; Q; C; \langle e, r\rangle\rangle$. In addition to the environment and current expression e, a reference r is maintained within the configuration. r is a reference to a special actor that can be written with **cancel** or **resume** and read with **finally** to store and retrieve results via side-effects. The benefits and details of this mechanism are discussed in Sec. 6.4.1.

Given a reference to an actor, **update** and **query** can be used to schedule writes respectively reads on the target actors by placing the corresponding **put**, **get** and $\lambda x \to e$ expressions into the queue of the corresponding actor inside Q. Note that, unlike all other actor operations from Fig. 6.6, the **put**, **get** and **unlock** functions are used internally only, they are not part of the actor interface and cannot be used by the programmer. **update** e $\lambda x \to e'$ schedules a message into the queue of actor e, which will apply $\lambda x \to e'$ to the actor's state to compute the new state of the actor, when executed. **query** e $\lambda x \to e'$ schedules a message to actor e which will apply $\lambda x \to e'$ to the actor's state.

The language introduces evaluation contexts to allow the extraction and parallel evaluation of sub-expressions from an expression e.

In the following, we will discuss the features of our actor language in more detail using specific examples.

6.4.1 Non-deterministic Result Values using Finally

Before investigating generic programming with actors in Sec. 6.5, we will introduce **cancel** e, **resume** e and e **finally** e' that allow to access the special actor r in the environment $\langle M; Q; C; \langle e, r\rangle\rangle$ in order to non-deterministically store a result in a parallel computation. e **finally** e' creates a new actor r that exists inside the scope of e. Within e, **cancel** e and **resume** e can be used to store a result in r. The e **finally** e' expression evaluates e and waits until either a result was stored in r or it is guaranteed that no result can ever be stored in r. Finally, e **finally** e' evaluates to the result stored in r or to e' if no result was stored. Hence, e **finally** e' allows the programmer to react to expressions which may or may not generate a result. An instructive example is searching for the occurrence of a string in an array, as shown in Fig. 6.7.

```
find : T[d1{N}]-> T -> boolean (*search for s in ar*)
find ar s =(array()(ar,\(i)->if(ar[i]=s) cancel true else ()))
finally false
```

*Figure 6.7: Parallel search of array ar for element s. If an entry s is found, the search can be aborted and the result true is stored via side-effect in the actor r, which is part of the configuration. Later, the result can be retrieved from r using e **finally** e′, which blocks until all side-effects on r generated by e have been evaluated. If no result is generated by the expression e, e **finally** e′ evaluates to e′ which is equal to false in the example.*

Using the array notation from Chap. 4, a map operation is used to generate a number of data-parallel tasks which string-compare array elements with the search string. Each task produces either a result true, indicating that the string was found, or no result at all. Note that the remaining parallel computations could be canceled as soon as a first result

was found. In the absence of side-effects, the result of all parallel tasks would need to be combined into a single result value, so that all tasks must be evaluated completely, even if their result will be discarded.

In order to overcome this limitation, we introduce **cancel** e, **resume** e and e **finally** e'. Similar to a return statement in C++, **cancel** and **resume** use a side-effect to store a result value in an actor r. e **finally** e' waits until all **cancel** and **resume** in e have been evaluated. Note that **finally** does not need to wait for **cancel** or **resume** that are nested in another **finally** in e. Afterwards, e **finally** e' evaluates to the value stored in r or to e' if no value was stored in r. If a result value is stored using **cancel** then all other computations in e may be disregarded. If the value is stored using **resume** then the remaining computations must be evaluated. In order to avoid unnecessary computations, the code in Fig. 6.7 compares the array entries in parallel and uses **cancel** to abort all other comparisons as soon as one matching entry is found. The **finally** clause evaluates to the stored result (true) if a match was found or false if not. As shown in the semantics in Fig. 6.12, the first evaluation of concurrent executions of **cancel** e or **resume** e define the result for **finally**, so that the result is non-deterministic if several **cancel** or **resume** expressions try to store different results concurrently.

Note that **cancel** and **resume** expressions have no effect without **finally** so that in the absence of **finally** the type system guarantees a consistent result value with type () for these expressions. In the presence of **finally**, a consistent result type is guaranteed by the semantics of e **finally** e'. Given that the type system has verified that the type of e' is equal to the type of the values stored in r using **cancel** and **resume** in e, then the type of e **finally** e' must be equal to the type of e'.

The usual semantics of *return* in imperative languages cannot be used in our language because we have no statement sequentializing operator ; so that waiting for non data flow dependent computations cannot be realized without using **finally**. In this sense, **finally** transforms the temporal dependency of the result value on some previous **cancel** or **resume** into an explicit side-effect via r.

6.5 Communication and Ordered Messages

In this section we will discuss ordered messages in the context of actors. The actor model from Clinger [31] gives no guarantees on message order except for causality. This means that the programmer has no control over the order messages are processed in unless he can construct a causality between messages in the sense that a preceding message must initiate the sending of a succeeding message. This represents a problem, when actors are used to communicate ordered events, like key presses, between processes. Even if the key presses are sent to an actor in a specific order, the actor may destroy the order of the events by processing the messages in any order.

Some actor based languages, like Erlang [8], give stronger guarantees on the order of messages sent to an actor but these stronger guarantees would prohibit automatic parallelization because messages inside functions could not be reordered without changing the semantics of the function.

In this section we will introduce the concept of *barriers*, which allow to reorder events inside a function without changing its semantics while retaining control over the order of events between different, dependent functions.

We will use an actor to communicate ordered messages (key presses) from a callback to

another, asynchronous function. This communication has two sides. Firstly, the ordered messages entering the program via the callback must be transmitted to the actor without losing the order on the messages. Secondly, the asynchronous function must receive the messages in order.

In order to guarantee a total order on the incoming messages, the OS (or library) that invokes the callback must enforce that the callback is invoked strictly non-concurrently. In other words, a call to the callback may be performed only if the last call to the callback is guaranteed to have finished beforehand. Otherwise, different messages could overtake each other and the order would be lost. We assume that this is enforced for all external messages provided by the system which require ordering.

We will use the **update** message to send the key events from the callback to an actor. The semantics of **update** (see Fig. 6.12) defers the write operation by placing it into a set inside the actor's queue. Therefore, the semantics for **update** guarantees no order with respect to subsequent update messages unless it is combined with the rule for function evaluation, which places a barrier for all messages in the queues of all actors after the function was fully evaluated. This means that exclusively those messages that are generated inside a function are protected by the barrier from those messages which depend on the function result by control or data flow, all other messages remain unaffected. The following examples illustrate push and pull based communication of ordered events using actors.

6.5.1 Pull Based Communication using Query

The code shown in Fig. 6.8 installs a callback that forwards the message in order to an actor which stores the message. Note that `actor<T>` denotes the type of an actor with state $\langle v, \lambda x \to e, f_1, f_2 \rangle$ where v must have type `T`. We assume that the system guarantees that the callback is not invoked concurrently so that new key messages cannot be passed into the program before the callback has returned. The update for the current key message is scheduled strictly before the barrier at the end of the callback and any future calls to the callback wait for the callback to return so that any future key messages are scheduled strictly after the barrier. Therefore, the actor maintains the order of the key messages when processing the updates.

The *main* function creates a new actor with initial state $v =$ ‘ ‘ and a continuation $\lambda x \to x$ before installing the callback and running *pull*. Note that according to Sec. 7.3.2, *pull* is a blocking, recursive function which will be executed separately from other tasks.

The blocking function *pull* constantly queries the actor state and reacts to it if necessary. Note that the pulling function only samples the state of the actor and some messages may be lost if the pulling frequency is too low. Buffering can be used to avoid missed updates. Nevertheless, the busy waiting strategy implemented with *pull* is not very efficient, instead a notification based, pushing strategy is introduced next.

6.5.2 Push Based Communication using Raise and Asynchronous Continuations

The actor used in Fig. 6.9 forwards the received message using **raise** to the asynchronous continuation specified in the constructor of the actor. All computations performed in the (mutually exclusive) update of an actor must terminate without blocking so that the actor is not continuously blocked from processing further messages. Hence, an actor cannot evaluate a blocking continuation inside an update. Instead, **raise** can be used to

```
callback : char -> actor<char> -> () (*send key state change to
actor*)
callback key a = update a \x->key (*barrier in lambda retains key
order*)

pull : actor<char> -> () (*continously pull actor state*)
pull a = let _ = query a \x->printf "last␣key␣%c" x in pull a

main : () -> ()
main =
  let a = new ' ' \x->x in (*new actor that stores key state*)
  let _ = install_key_callback callback a in (*install callback with
  arguments key, a*)
  pull a
```

Figure 6.8: Pull based inter-task communication. The type of an actor that contains a value v with type T is denoted as `actor<T>`. Here, install_key_callback is a built in function that accepts a callback of the given type and an actor that will be passed through to the callback which will be evaluated every time the user presses a key.

defer control flow to the actor's asynchronous continuation which was specified as second parameter to **new**. This continuation is evaluated asynchronously by placing it in the actor's queue.

```
callback : char -> actor<char> -> () (*send key state change to
actor*)
callback key a = update a (\x->raise a key key) (*store key and
raise*)

main : () -> ()
main =
  let a = new ' ' (\x->printf "last␣key␣%c" x) in (*new actor that
  stores key state and outputs key on raise*)
  let _ = install_key_callback callback a in (*install callback with
  arguments key, a*)
  (*asynchronous rest of the program*)
```

Figure 6.9: Push based inter-task communication.

raise $e\ e'\ e''$ evaluates to e'' after scheduling a message to actor e which will compute $\lambda x \to e_h\ e'$ where $\lambda x \to e_h$ is the continuation stored in the actor's state at construction time so that `raise a key key` from line 2 evaluates to *key* after scheduling $e\ \lambda x \to eprintf$ "*last key %c*" x from line 6.

The semantics in Fig. 6.12 uses the flag f_1 in an actors state $\langle v, \lambda x \to e, f_1, f_2 \rangle$ to enforce that update messages on an actor are mutually exclusive. The flag f_2 is used to enforce that the continuations $\lambda x \to e$, scheduled via **raise**, are also mutually exclusive.

In Fig. 6.9, all raise messages are separated by the barrier generated at the end of the lambda defining the asynchronous continuation and all updates are separated by the barrier at the end of the callback. Since both, update and raise messages, are non-overlapping due to the flags and fully ordered by barriers, the order of the messages flowing into the system via the callback is maintained.

This concludes our discussion on how ordered message communication can be per-

formed with our flavor of actors. In the next section we will introduce the **sync** keyword which allows to compose actors.

6.6 Actor Composition and Dead-Lock Freedom

As discussed in the beginning of this chapter, actors can be seen as consistently locked shared memory. Locks allow to synchronize several different regions of shared memory (each protected by an individual lock) simply by acquiring multiple locks before entering a critical section. This usage pattern may be applied, for example, to build a database application where the data is stored in a two dimensional table and both, rows and columns, may need to be locked independently, as shown in Fig. 6.10. This could be achieved by assigning a lock to every table entry and then all elements on a row or column can be locked as needed. Care must be taken that multiple locks are always acquired in the same order to avoid dead-locks.

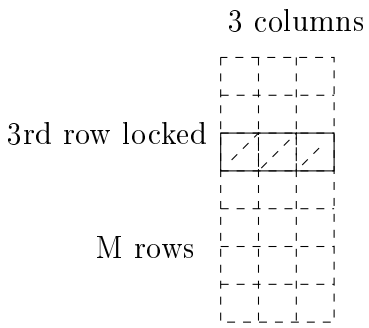

Figure 6.10: A database implemented as an array of actors. All fields can be updated asynchronously. Consistent updates on rows or columns can be implemented by synchronized locking rows or columns.

In order to enable the synchronization of actors, we enhance the actor model by the **sync** keyword which allows to create a new actor that is defined as the composition of a set of actors as shown in Fig. 6.11.

```
(*each actor stored in the N rows times 3 columns table contains an
int*)
updateRow : actor<int>[d2{N,3}]-> int -> (<int,int,int>-><int,int,
int>) -> ()
updateRow table r f = let s = new sync table.row(r) in update s f
```

Figure 6.11: Synchronized update of a row of actors in a two dimensional array. Here, `sync table.row(r)` is syntax sugar for `sync table[r,0] ... table[r,N-1]`

As shown in the semantics in Fig. 6.12 in rule exec-atomic-update, an **update** of a composed actor synchronously updates all component actors. This allows to implement the database by creating an actor for every table entry and building compositions of actors for every table row or column.

In contrast to traditional locks, dead-locks cannot be constructed when using **sync** because the implementation of **sync** guarantees that locks are always acquired in a correct order. Thus, our language is *dead-lock free* because **sync** is the only way to combine locks in our language.

Note that all other lock-like constructs that can be built using actors or imported using the foreign function interface (introduced in Chap. 7) are not effective locks.

As all state changing updates on actors are deferred, it is impossible for the programmer to enforce that a read or write access to an actor has finished at any specific program point so that any lock like construct trying to protect these reads and writes can never be safely released.

Note that barriers only guarantee that messages before and after the barriers do not mix. Messages scheduled strictly before a barrier may execute at any time before or after the barrier was reached, just as long they execute before those messages that are scheduled strictly after the barrier. Therefore, barriers cannot be used to enforce that a message has been processed before a specific program point.

This concludes our discussion of the new actor language features shown in Sec. 6.4.

6.7 Semantics

In this section we will discuss the semantics of actors as shown in Fig. 6.12. Note that we must design the semantics of our actors carefully. On the one hand, we need to be able to reorder messages inside a function with little restriction so that we can apply automatic parallelization via our task graph analysis from Chap. 2. On the other hand, some control over the order of messages is needed so that ordered events, like the key presses discussed in Sec. 6.5, can be communicated between threads without losing the order.

New actors are created with rule (new) which stores an initial value v, an asynchronous continuation $\lambda x \to e_h$ that can be scheduled via **raise** and two flags that are used to lock an actor so that **update** and **raise** are executed atomically. Rule (add-update) places an expression that reads the current state via **get**, applies a function and writes the result using **put** into the queue of an actor. **update** tests and sets the lock f_1 to guarantee atomicity and **put** resets f_1.

Rule (add-raise) schedules the execution of the asynchronous continuation and (exec-raise) executes the continuation after testing and setting f_2 in the state. **unlock** is used to reset the lock f_2 after the continuation has finished, guaranteeing that continuations of a single actor run strictly sequentially. **query** creates an asynchronous read but without any locking.

6.7.1 Barriers

The queue associated with an actor is a tuple of sets where the separation of two adjacent sets is a barrier that enforces a partial order between all elements in the queue. Queue elements within a single set are unordered with respect to each other. Queue elements from different sets are ordered so that elements in the tuple components with smaller index are processed strictly before components from sets with higher index. This can be seen in all (exec-*) rules that remove and process elements only from the first set in the tuple whereas all rules (add-*) add only to the last set in the tuple.

Rule (beta) computes the result of a function application and places a new barrier into the queues of all actors by adding a new set to the end of each tuple. Therefore, all messages scheduled inside a function are protected by the barrier from all messages that depend on the result of the function. Rule (remove-barrier) removes the first set in a tuple if it is empty and not the last in the tuple so we can progress across barriers as soon as all messages from before the barrier have been processed.

6.7.2 Cancel, Reduce and Finally

e *finally* e' sets up a frame for e, within this frame an actor r' is made accessible as part of the configuration that may store a result that is retrieved by the (finally-*) rules after e was evaluated. r' may be written using rules (resume) and (cancel) where both try to update r' (all but the first update are discarded). The only difference between both rules is that (cancel) may skip computations in E if the **cancel** expression is not nested in another **finally**. (finally-some) evaluates to the result stored in r'. Rule (finally-none) must wait for all messages that could update r' in order to guarantee that no updates on r' are possible and evaluates to the alternative result e'.

6.7.3 Trace and Trace Set

Since we have added non-determinism to our language, it is no longer guaranteed that different runs of the same program always yield the same result. Therefore, we must extend our program semantics from denotational semantics, which maps a program to a single result value, to a collecting semantics. To this end, we define program traces analogously to Sec. 6.1. A trace is defined as a list of snapshots of the configuration $\langle M, Q, C; \langle e, r\rangle\rangle$ where a new snapshot is added to the list when a rule from the semantics was applied to the last list element. The list is initialized with the single element $\langle [], [], []; \langle \mathit{program}, \bot\rangle\rangle$. The set of all possible traces is generated by taking the initial set $\{\langle\langle [], [], []; \langle \mathit{program}, \bot\rangle\rangle\rangle\}$ and building the trace for every list in the set. Originally, there is only one list in the set. Every time multiple rules from the semantics could be applied to one of the lists in the set, this list is copied so that each possible rule can be applied to an individual copy of the list. In this way, the trace set contains traces representing all possible interleavings that may be generated by evaluating the program.

6.8 Implementation

In this section we will discuss the implementation of the actor semantics from Fig. 6.12. Especially an efficient implementation of the message ordering constraints, i.e. the barriers at the end of functions (rule (beta)) and the waits on messages in rule (finally-none), may be non-obvious.

Clinger [31] notes that the implementation of a semantics does not need to generate all possible traces of a given program. Indeed, our implementation can produce any trace from the trace set of a given program. Any fairness or ordering guarantees given by the semantics must hold for any trace, so these guarantees are unaffected.

6.8.1 Barriers

We can exploit this fact in order to achieve an efficient implementation of our semantics in the following way. We implement a semantics which is stronger, yet compatible with the semantics we have defined in Fig. 6.12.

$$\frac{C(m) = \langle m_1, .., m_n \rangle \quad M(m_i) = \langle v_i, \lambda x \to e_h^i, l_1, l_2 \rangle}{\langle M; Q; C; \langle \textbf{get}\ m, r \rangle \rangle \hookrightarrow \langle M; Q; C; \langle \langle v_1, .., v_n \rangle, r \rangle \rangle} \text{(get)}$$

$$\frac{C(m) = \langle m_1, .., m_n \rangle \quad M(m_i) = \langle v_i', \lambda x \to e_h^i, \mathsf{true}, l \rangle}{\langle M; Q; C; \langle \textbf{put}\ m\ \langle v_1, .., v_n \rangle, r \rangle \rangle \hookrightarrow \langle M[m_i \to \langle v_i, \lambda x \to e_h^i, \mathsf{false}, l \rangle]; Q; C; \langle (), r \rangle \rangle} \text{(put)}$$

$$\frac{M(m) = \langle v, \lambda x \to e_h, l, \mathsf{true} \rangle}{\langle M; Q; C; \langle \textbf{unlock}\ m, r \rangle \rangle \hookrightarrow \langle M[m \to \langle v, \lambda x \to e_h, l, \mathsf{false} \rangle]; Q; C; \langle (), r \rangle \rangle} \text{(unlock)}$$

$$\frac{m \notin dom(M)}{\langle M; Q; C; \langle \textbf{new}\ v\ \lambda x \to e_h, r \rangle \rangle \hookrightarrow \langle M[m \to \langle v, \lambda x \to e_h, \mathsf{false}, \mathsf{false} \rangle]; Q[m \to \langle \emptyset \rangle]; C[m \to \langle m \rangle]; \langle m, r \rangle \rangle} \text{(new)}$$

$$\frac{}{\langle M; Q; C; \langle \textbf{query}\ m\ (\lambda x \to e_r), r \rangle \rangle \hookrightarrow \langle M; Q[m \to \langle Q(m)_0, .., Q(m)_{-1} \cup \{(\lambda x \to e_r)\ (\textbf{get}\ m)\} \rangle]; \langle (), r \rangle \rangle} \text{(add-query)}$$

$$\frac{}{\langle M; Q; C; \langle \textbf{update}\ m\ \lambda x \to e_r, r \rangle \rangle \hookrightarrow \langle M; Q[m \to \langle Q(m)_0, .., Q(m)_{-1} \cup \{\textbf{put}\ m\ (\lambda x \to e_r\ (\textbf{get}\ m))\} \rangle]; \langle (), r \rangle \rangle} \text{(add-update)}$$

$$\frac{M(m) = \langle v', \lambda x \to e_h, l_1, l_2 \rangle}{\langle M; Q; C; \langle \textbf{raise}\ m\ v\ e, r \rangle \rangle \hookrightarrow \langle M; Q[m \to \langle Q(m)_0, .., Q(m)_{-1} \cup \{(\lambda x \to e_h)\ v\} \rangle]; \langle e, r \rangle \rangle} \text{(add-raise)}$$

$$\frac{\langle M; Q; C; \langle e_r[x := v], r \rangle \rangle \hookrightarrow \langle M'; Q'; C'; \langle v', r \rangle \rangle}{\langle M; Q; C; \langle (\lambda x \to e_r)\ v, r \rangle \rangle \hookrightarrow \langle M'; Q'[m_i \to \langle Q'(m_i)_0, .., Q'(m_i)_{-1}, \emptyset \rangle]; C'; \langle v', r \rangle \rangle} \text{(beta)}$$

$$\frac{\langle M; Q; C; \langle e, r \rangle \rangle \hookrightarrow \langle M'; Q'; C'; \langle e', r \rangle \rangle}{\langle M; Q; C; \langle E[e], r \rangle \rangle \hookrightarrow \langle M'; Q'; C'; \langle E[e'], r \rangle \rangle} \text{(eval-ctxt)}$$

$$\frac{Q(m)_0 = \emptyset \quad |Q(m)| > 1}{\langle M; Q; C; \langle e, r \rangle \rangle \hookrightarrow \langle M; Q[m \to \langle Q(m)_1, .., Q(m)_{-1} \rangle]; C; \langle e, r \rangle \rangle} \text{(remove-barrier)}$$

$$\frac{m \notin dom(M) \quad C(m_i) = \langle m_1^i, .., m_k^i \rangle}{\langle M; Q; C; \langle \textbf{sync } \langle m_1, .., m_n \rangle, r \rangle \rangle \hookrightarrow \langle M; Q[m \to \langle \emptyset \rangle]; C[m_i \to \langle m_1^i, .., m_k^i \rangle]; \langle m, r \rangle \rangle} \text{ (sync)}$$

$$\frac{e_l \in Q(m)_0 \quad e_l = (\lambda x \to e_h)\ (\textbf{get } m)}{\langle M; Q; C; \langle e, r \rangle \rangle \hookrightarrow \langle M; Q[m \to \langle Q(m)_0 \setminus e_l, Q(m)_1, .., Q(m)_{-1} \rangle]; \langle (\lambda \_ .e)\ e_l, r \rangle \rangle} \text{ (exec-query)}$$

$$\frac{e_l \in Q(m)_0 \quad e_l = (\lambda x \to e_h)\ v \quad M(m) = \langle v', \lambda x \to e_h, l, \mathsf{false} \rangle}{\langle M; Q; C; \langle e, r \rangle \rangle \hookrightarrow \langle M[m \to \langle v', \lambda x \to e_h, l, \mathsf{true} \rangle]; Q[m \to \langle Q(m)_0 \setminus e_l, Q(m)_1, .., Q(m)_{-1} \rangle]; \langle (\lambda \_ .e)\ ((\lambda \_ .\textbf{unlock } m)\ e_l), r \rangle \rangle} \text{ (exec-raise)}$$

$$\frac{e_l \in Q(m)_0 \quad e_l = \textbf{put } m\ (\lambda x \to e_r\ (\textbf{get } m)) \quad C(m) = \langle m_1, .., m_n \rangle \quad M(m_i) = \langle v_i, \lambda x \to e_h^i, \mathsf{false}, l \rangle}{\langle M; Q; C; \langle e, r \rangle \rangle \hookrightarrow \langle M[m_i \to \langle v_i, \lambda x \to e_h^i, \mathsf{true}, l \rangle]; Q[m \to \langle Q(m)_0 \setminus e_l, Q(m)_1, .., Q(m)_{-1} \rangle]; C; \langle (\lambda \_ .e)\ e_l, r \rangle \rangle} \text{ (exec-atomic-update)}$$

$$\frac{r' \notin dom(M) \quad \langle M[r' \to \langle \langle (), \mathsf{false} \rangle, id, \mathsf{false}, \mathsf{false} \rangle]; Q; C; \langle e, r' \rangle \rangle \hookrightarrow \langle M'; Q'; C'; \langle v', r' \rangle \rangle \quad M'(r') = \langle \langle v, \mathsf{true} \rangle, id, \mathsf{false}, \mathsf{false} \rangle}{\langle M; Q; C; \langle e\ \textbf{finally } e', r \rangle \rangle \hookrightarrow \langle M'; Q'; C'; \langle v, r \rangle \rangle} \text{ (finally-some)}$$

$$\frac{r' \notin dom(M) \quad \langle M[r' \to \langle \langle (), \mathsf{false} \rangle, id, \mathsf{false}, \mathsf{false} \rangle]; Q; C; \langle e, r' \rangle \rangle \hookrightarrow \langle M'; Q'; C'; \langle v', r' \rangle \rangle \quad M'(r') = \langle \langle (), \mathsf{false} \rangle, id, \mathsf{false}, \mathsf{false} \rangle \quad Q(r') = \langle \emptyset \rangle}{\langle M; Q; C; \langle e\ \textbf{finally } e', r \rangle \rangle \hookrightarrow \langle M'; Q'; C'; \langle e', r \rangle \rangle} \text{ (finally-none)}$$

$$\frac{\textbf{finally} \notin E}{\langle M; Q; C; \langle E[\textbf{cancel } v], r \rangle \rangle \hookrightarrow \langle M; Q; C; \langle \textbf{update } r\ \lambda x \to \textbf{if } x = \langle \_, \mathsf{false} \rangle\ \textbf{then } \langle v, \mathsf{true} \rangle\ \textbf{else } x, r \rangle \rangle} \text{ (cancel)}$$

$$\frac{}{\langle M; Q; C; \langle \textbf{resume } v, r \rangle \rangle \hookrightarrow \langle M; Q; C; \langle \textbf{update } r\ \lambda x \to \textbf{if } x = \langle \_, \mathsf{false} \rangle\ \textbf{then } \langle v, \mathsf{true} \rangle\ \textbf{else } x, r \rangle \rangle} \text{ (resume)}$$

Figure 6.12: Semantics for our actor language as defined in Fig. 6.6. Here, v_i extracts i-th component t_i from the tuple $v = \langle t_0, .., t_n \rangle$ and v_{-1} extracts last element t_n. $|v| = n+1$ gives number of elements in v. Unquantified variables are implicitly all quantified and $id \triangleq \lambda x \to x$.

Instead of placing barriers in the queues of all actors at the end of functions, we store messages scheduled on an actor in a totally ordered queue and enforce that all messages scheduled before a barrier must be placed in the ordered queue before the barrier is reached.

In order to guarantee that all required messages are placed in the queue before the barrier, the task graph constructed according to Chap. 2 must be extended with edges from all nodes that schedule messages to the function result node, which represents the barrier.

Since these additional task graph edges guarantee that messages which precede the barrier are placed in the ordered queue before the barrier and messages that depend on the function result are placed in the queue strictly afterwards, the partial order demanded by barriers is automatically maintained by the order of the messages inside the queue, so that the barrier from Rule (beta) is effectively a no-op.

Note that this implementation is stronger than the language semantics from Fig. 6.12, because it guarantees that messages are executed in the order in which they were scheduled, which is one of the many valid orders supported by our language semantics. Nevertheless, we cannot restrict our language semantics to this specific order, because this would prevent semantic invariant reordering of messages, which is required for automatic parallelization.

In the following example we show the task graph generated from a function which illustrates messages and barriers. Fig. 6.13 shows function f that schedules updates $1-6$ on actor a. Each lambda places a barrier into the message queue of a so that messages $1-3$ are scheduled strictly before messages $4-6$ because the lambda in line 3 depends on the result of the other one.

```
f : actor<int> -> ()
f a =
 (\a b -> let _ = update a \x -> 4 in let _ = update a \x -> 5 in let
  _ = update a \x -> 6) a
 ((\a update a \x -> 1 in let _ = update a \x -> 2 in let _ = update
 a \x -> 3) a)
```

Figure 6.13: Concurrent and barrier protected messages on actor a. The barriers at the end of the lambdas avoid mixing of messages.

Fig. 6.14 shows the correspondence of the task graph for the program from Fig. 6.13 to its maximally deferred trace $\langle [a \to \langle 0, \lambda x \to e, \mathsf{false}, \mathsf{false}\rangle], [a \to Q(a)], [a \to \langle a\rangle], \langle (), \bot\rangle\rangle$ where $Q(a) = \langle \{\mathbf{put}\ a\ 1, \mathbf{put}\ a\ 2, \mathbf{put}\ a\ 3\}, \{\mathbf{put}\ a\ 4, \mathbf{put}\ a\ 5, \mathbf{put}\ a\ 6\}\rangle$. The maximally deferred trace is obtained by applying the semantics on the code without applying any (exec-*) rule until no more rules can be applied so that all messages are enqueued but not executed. This trace exposes the partial order imposed by the barriers on all messages of an actor, as no messages are prematurely consumed. This partial order must be maintained by the corresponding task graph. Due to the additional task graph edges introduced in this section, the task graph semantics defined in Chap. 2 and the totally ordered queue used to store the messages, the partial order on messages imposed by the task graph is equivalent to the partial order required by the maximally deferred trace of $Q(a)$.

Note that the scheduling algorithm from Chap. 3 must be modified to not merge nodes scheduling messages with nodes containing blocking operations. A function must

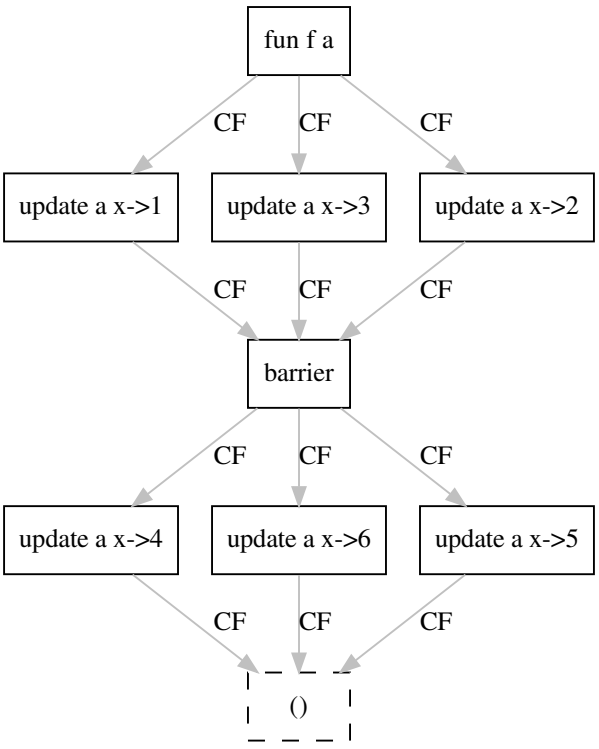

Figure 6.14: Task graph for the program from Fig. 6.13. Edges labelled a from root to all nodes containing a were omitted for readability and the lambdas were in-lined.

not return before all messages have been scheduled but blocking operations may not terminate so that merging such tasks may defer function termination indefinitely.

6.8.2 Finally Expressions

e **finally** e' expressions must wait for all non-nested **cancel** or **resume** expressions in e in order to reliably determine if any result was stored in the actor r' or not. In order to respect these implicit dependencies of e **finally** e' on **cancel** and **resume**, the task graph constructed according to Chap. 2 must be extended with edges from all nodes that contain non-nested **cancel** or **resume** expressions from e to the node containing the corresponding e **finally** e'. Furthermore, the control flow dependency from e' on e **finally** is added to the graph by replacing **finally** e' by an if-expression which reads the state of r'. Furthermore, we make updates on r' instantaneous using **put** rather than **update**, which is compatible with our semantics which randomly chooses a winner when multiple **update** are racing on r'. In combination, these implementation choices guarantee that all updates on r' have finished before **finally** is evaluated.

An example of these transformations is shown in Fig. 6.15, which depicts the task graph for the parallel array search from Sec. 6.4.1. Note that the computations in the *array* expression are independent so that every computation generates its own task graph which is control flow dependent on the *array* expression.

6.8.3 Actor Scheduling

In the following, we highlight the difference between our actor implementation and to the usual notion of mailbox addressed micro-process per actor.

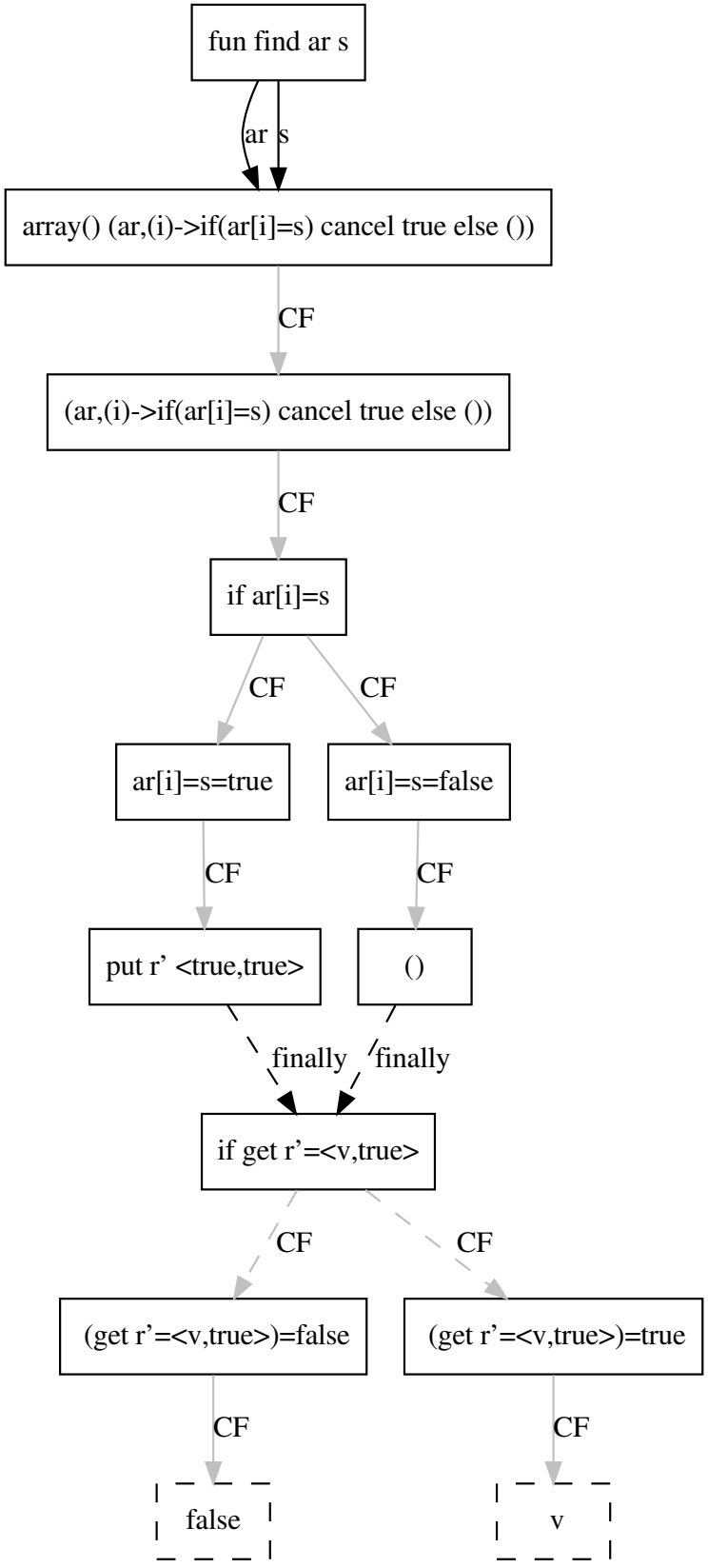

Figure 6.15: Task graph obtained from the function `find ar s =(`***`array`***`()(ar,\(i)->`***`if`***`(ar[i]=s)`***`cancel`*** `true` ***`else`*** `()))`***`finally`*** `false`*, discussed in Sec. 6.4.1.*

Instead of having a virtual process per actor, we use a fair reader-writer lock to protect the actor's memory. When a message on an actor is supposed to be scheduled and the actor's queue is empty, the message tries to lock the actor and to perform the update synchronously. Only if the lock fails because the actor is busy with another read or write, then the request is en-queued and processed later. Note that a scheduling operation cannot block without violating the actor semantics so that scheduling operations cannot wait for lock to become free. Most updates on actors are small, so using a lock to directly modify the actor instead of communicating with another thread is a lot more efficient as we have discussed in Chap. 3. The lock based default behavior shows the same performance characteristics as locked shared memory in C++.

Furthermore, the reader-writer lock allows different read messages to be processed in parallel. In order to implement parallel reads on mailbox based actor implementations, the actor would need to look ahead for messages which are known to be read-only, which is non-trivial.

6.8.4 Sync

The **sync** message uses a unique id (e.g. the actor's memory address) to determine a total order on all actors and locks actors in ascending order to avoid dead-locks.

6.9 Actor Synchronization

In addition to reader-writer locks, the mutual exclusion of message processing in actors can be implemented using other synchronization primitives which exhibit different performance characteristics. For example, an actor implementation based on transactions can be used to perform speculative destructive updates by combining actors with uniqueness types, as follows.

6.9.1 Transactions

Unique objects can be placed inside actors, as we have discussed in Sec. 6.3.2. This allows us to use messages to schedule updates of unique objects. If the actor uses transactions to synchronize messages and if we manage to send the messages concurrently to the actor, then the updates can be performed in parallel for those transactions that contain no collisions. We will use this method to build a parallel binary tree construction as shown in Fig. 6.16.

```
main : () -> ()
main = let tree:actor<<unique<BinTree>,int>> = new <(unique<BinTree>
 null 0 null),0> (\x -> printf "tree␣finished") in
array () (d1{1000},\(i)->update tree \<x,c> -> if(c+1==1000) raise
tree <x.add(rand c),c+1> else <x.add(rand c),c+1>)
```

Figure 6.16: Speculative destructive construction of a binary tree. Here rand c returns the n-th entry of a sequence of pseudo random numbers.

Note that non-deterministic parallel algorithms (like the one shown in Fig. 6.16) usually accumulate non-deterministically arriving inputs and results from sub-computations

and may spawn new computations based on the specific history of results already available. As these results must be communicated between different tasks, non-deterministic parallel algorithms may be seen as a special case of inter task communication. Nevertheless, these algorithms often have the additional complexity that termination of the algorithm is not easily determined.

Quiescence (absence of messages) in a network of actors cannot be efficiently determined. Often sound approximations of the network's state must be computed to determine termination. In the example in Fig. 6.16, we use a counter to compute a lower bound on the number of elements that have already been added to the tree. The algorithm is finished as soon as this number reaches 1000. The algorithm fails to terminate if less than 1000 elements are eventually added. If more elements are added then the exact state of the tree is unknown when the continuation is invoked. This fragile dependence on approximations underlines the complexity inherent to programming networks of actors. Furthermore, care must be taken that the counter c does not destroy most parallelism. Parallel counter implementations are discussed in more detail in [101].

Table 6.17 compares the performance of destructively building a binary tree asynchronously (code as shown in Fig. 6.16) using a reader-writer lock, hardware transactions and sequentially without actors (code as shown in Fig. 5.7).

Code	time [ms]	$\sigma/time$ [%]
Reader Writer Lock (all)	208.7.1	123
Reader Writer Lock (first 10%)	451.6	102
Reader Writer Lock (last 10%)	21.2	85
Lock Elision (all)	440.2	239
Lock Elision (first 10%)	67.1	102
Lock Elision (last 10%)	55.7	220
Sequential	13.1	30

Figure 6.17: All times were measured on an Intel Xeon CPU E3-1270 v3 at 3.50GHz with lock elision enabled. The version marked with (all) executed all updates in parallel, the version marked with (first/last 10%) executed only the first/last 10% of the updates in parallel before/after switching to a sequential implementation. Collisions should be least likely for the last items added to the tree.

To avoid interference from the counter c, we removed it from the program and tested for completion of the tree by waiting for a quiescent state of the worker threads after submitting all tree items. This mechanism cannot be used if anything else except the tree is computed so that in a real world implementation a counter is required. A detailed discussion of parallel counters can be found in [101].

The performance of the lock elision versions is very sensitive to several dynamic parameters. The ratio of writes to reads should be low and most of the updates should be performed sequentially to achieve a reasonable task (and transaction) size for speedup (see Chap. 3). Nevertheless, larger transaction sizes increase the risk of collisions and the size of transactions is limited by the hardware implementation which uses the L1 cache to check for collisions. It is not obvious how to find the right balance. In our experiments, it turned out that even in the absence of writes in combination with the largest transaction size the hardware would support, the sequential tree construction still outperformed the transactional construction. It is possible that these performance problems are related to a hardware bug in Intel's transactional memory implementation [68].

6.9.2 Atomic Instructions

In addition to transactional locks, modern CPUs support atomic messages like increment, decrement, compare and swap, etc. A simple analysis can be used to determine whether all messages performed on an actor conform to these supported operations. If that is the case, the actor messages can be efficiently implemented using atomic instructions.

6.10 Data-Race Freedom

In this section, we formally define data-races and show that adding actors to our language does not introduce data-races and keeps the language semantics invariant under parallelization according to Lemma 1.

The definition of trace sets from Sec. 6.7.3 allows us to define data-races. Given a semantics, mapping a program to a trace set which contains all traces the program may produce, we define a data-race as a difference in the trace set of the fully sequential program and the trace set of any parallel version of the same program. This means that the parts of our language defined in the previous chapters are data-race free, since the semantics was shown to be invariant under parallelization. This is not surprising, since our language without actors had no visible side-effects. In the following we present a proof sketch to show that the semantics of our language, including actors, is data-race free.

Lemma 10. *The semantics of the actor language is invariant under parallelization according to Lemma 1 and thus data-race free.*

Proof. The semantic invariance under parallelization and thus data-race freedom of our language without actors was discussed in the previous chapters. Hence, we only need to consider the part of our language dealing with actors and state.

The semantics in Fig. 6.12 states that all **update**, **query** and **raise** messages on actors are deferred and stored in sets so that any order on these messages is lost except for those messages that are separated by a barrier in all possible traces. The barriers introduced at the end of every functions in rule (beta) are effective only for those messages that are generated inside the function.

Hence, inside a function, **update**, **query** and **raise** messages can be reordered without changing the semantics of the function as long as the flow dependencies of the expressions generating the messages are respected.

Our task graph analysis from Chap. 2 respects all flow dependencies and parallelizes only inside a single function so that any reordering of **update**, **query** and **raise** messages produced by the function's task graph has the same semantics as the original function. Furthermore, the additional task graph edges for actor messages, as introduced in Sec. 6.8.1, guarantee that all **update**, **query** and **raise** are executed strictly inside the function they were defined in.

Thus, the semantics of the language including actors is maintained by the task graph analysis.

The argument for reordering **cancel** and **resume** messages inside the expression e of a e **finally** e' is analogous. □

6.11 Conclusion

Actors, as presented in this chapter, are a powerful tool to express concurrent and parallel destructive operations on graph shaped data structures and dynamic parallelism. By careful extension of the actor semantics from unordered messages to barrier induced partial orders, message ordering constraints can be concisely formulated and efficiently implemented while automatic, semantic invariant parallelization remains possible. We have shown that our actors can be efficiently implemented and enable exploitation of various synchronization methods like transactional memory and atomic instructions. Nevertheless, the complexity inherent to the non-deterministic actor model is hard to reason about and complex to verify. Parallel decomposition into a possibly dynamic task graph must be specified manually by the programmer and cannot be automatically adapted to different target systems. Therefore, we argue that not everything should be an actor [62], instead the deterministic parallel programming constructs introduced in the previous chapters should be used instead unless the programming goal cannot be achieved deterministically.

Furthermore, scheduling of actor networks on distributed systems remains an open problem. Generating correct code to distribute actors on a network of machines is trivial. The internal state of actors is not directly accessible from the outside, instead messages must be sent to communicate. Given a mapping from actor addresses to network locations, it is trivial to send a message to an actor on a remote machine. Nevertheless, static scheduling of actors, i.e. deciding which network location the actor should be placed on, is complex. Actors can be used to express dynamic graphs of communication making it non-trivial to statically estimate the amount of communication and the amount of computation performed by an actor network. Dynamic scheduling provides load balancing capabilities and should be used to correct miss-allocations from a static schedule. Nevertheless, a fully dynamic schedule suffers from the lack of lookahead. Finding good approximations for actor computation and communication is actively being researched, for example [55] uses stochastic modeling to deal with dynamic data dependencies.

Chapter 7

Implementation of funkyImp

In this chapter we will discuss the **funkyImp** compiler implementation in Sec. 7.2 and the run-time system in Sec. 7.3. Beforehand, we discuss additional language features and language implementation details in Sec. 7.1.

7.1 Language Implementation

Several functional language features, like higher order functions, partial function application and pattern matching cannot easily be mapped onto the machine code executed by today's processors. Since these features are not natively supported by the processors, they must be emulated incurring some performance overhead. Static analyses are used to remove some unnecessary boxing and closures while graph rewriting techniques can be used to execute such programs efficiently.

Nevertheless, the sequential performance of programs containing such higher level, functional features is often not competitive with the performance of sequential C programs [95] which can be translated into hardware friendly machine code more easily by the compilers available today [106]. Furthermore, most parallel programs are developed using imperative languages (like C++) so that programmers are familiar with neither the performance implications of these functional features nor how these features are properly used from a software engineering perspective. In other words, most programmers will not miss these features.

In order to avoid the performance problems of these features and to make our language easy to learn (for C++ programmers) and easy to integrate into existing tools, our implementation abstains from these features.

In order to further improve the usability and performance of our language, we have implemented the following additional language features, which we have not discussed so far.

Firstly, inside array types and for class fields, the programmer can specify a type attribute to control whether a value is an option or not. If the value is an option, then it is either null or a reference to a value of the corresponding type. Values that are not options are unboxed instances of the type. Thus, arrays of pointers to objects are clearly separated from arrays of values, which is an important prerequisite for efficient data-parallel code.

Secondly, the programmer can annotate functions with the attributes *group* or *thread* to indicate that functions are not re-entrant or must run on a specific hardware thread. The *group* attribute takes as single argument a group name and all functions in the same

group are guaranteed to not execute concurrently. The *thread* attribute takes a thread name as single argument and all functions belonging to the same thread are guaranteed to run on the same hardware thread. These annotation are especially useful in conjunction with the foreign function interface, which allows to import C-libraries into our language. Many C-libraries provide functions which are mutually not re-entrant and must thus be placed in the same group. Some libraries, for example OpenGL [145], store hidden state in thread local memory so that all calls to the library must originate from the same thread.

Due to the lack of higher order functions, we have implemented the actors described in Chap. 6 as classes annotated with the keyword *actor*. In *actor* classes, all fields are non-public and all functions are either read-only (*query*) or they read and return a modified version (*update*). This implementation of actors has the additional advantage that all methods acting on an actor's state must be declared in the actor's class. Hence, the scope of analyses which are applied on the behavior of an actor is limited to the single source file that defines the actor rather than the full program.

7.2 Compiler Implementation

The current prototype implementation of **funkyImp** can be found at [59]. We have based our compiler on the javac compiler from the OpenJDK [113]. One specific advantage of the OpenJDK compiler is that it is one of the very few compilers, for a realistic language, which is based on a grammar generated parser rather than a hand written one so that changes to the language syntax could be integrated with little effort. The grammar was extended to handle all the language features we have introduced in this thesis, including data-parallel array operations from Chap. 4, unique types from Chap. 5 and actors from Chap. 6.

The original Java compiler performs virtually no optimizations since the optimizations are performed by the virtual machines running the generated byte code. We have implemented the required type system changes and the required analyses from the previous chapters into the compiler, in order to produce code for performance evaluation. Fig. 7.1 shows the compilation tool chain for **funkyImp** programs.

The pre-processor is based on the anarres-cpp pre-procesor [97] and the parser is generated using Antlr [114]. The type checker was extended by template instantiation, uniqueness types and polytope restricted sub arrays utilizing the barvinok library [11] to perform subset tests of parametric polytopes.

We have extended the existing data flow analysis to statically verify that every variable is assigned exactly once, so our implementation is a single assignment language executing blocks of statements rather than evaluating a huge, functional expression. Imperative programmers are comfortable with this concept, these kind of programs map well onto today's hardware and step-by-step debugging becomes meaningful. Instead of parallelizing let-expressions, as shown in Chap. 2, we parallelize statements. After having introduced side-effecting results using *cancel*, *resume* and *finally* in Chap. 6, the difference between our core language from Chap. 2 and the single assignment language implemented by our compiler reduces to syntactic sugar. Therefore, in the absence of implementation bugs, all properties defined for **funkyImp** in this thesis also hold for the implementation.

The task graphs generated by the flow analysis according to Chap. 2 are stored and processed using the jgrapht library [105]. The work and memory analysis for task graph nodes is based on data from [45]. Then, static scheduling for the task graphs is performed according to Chap. 3.

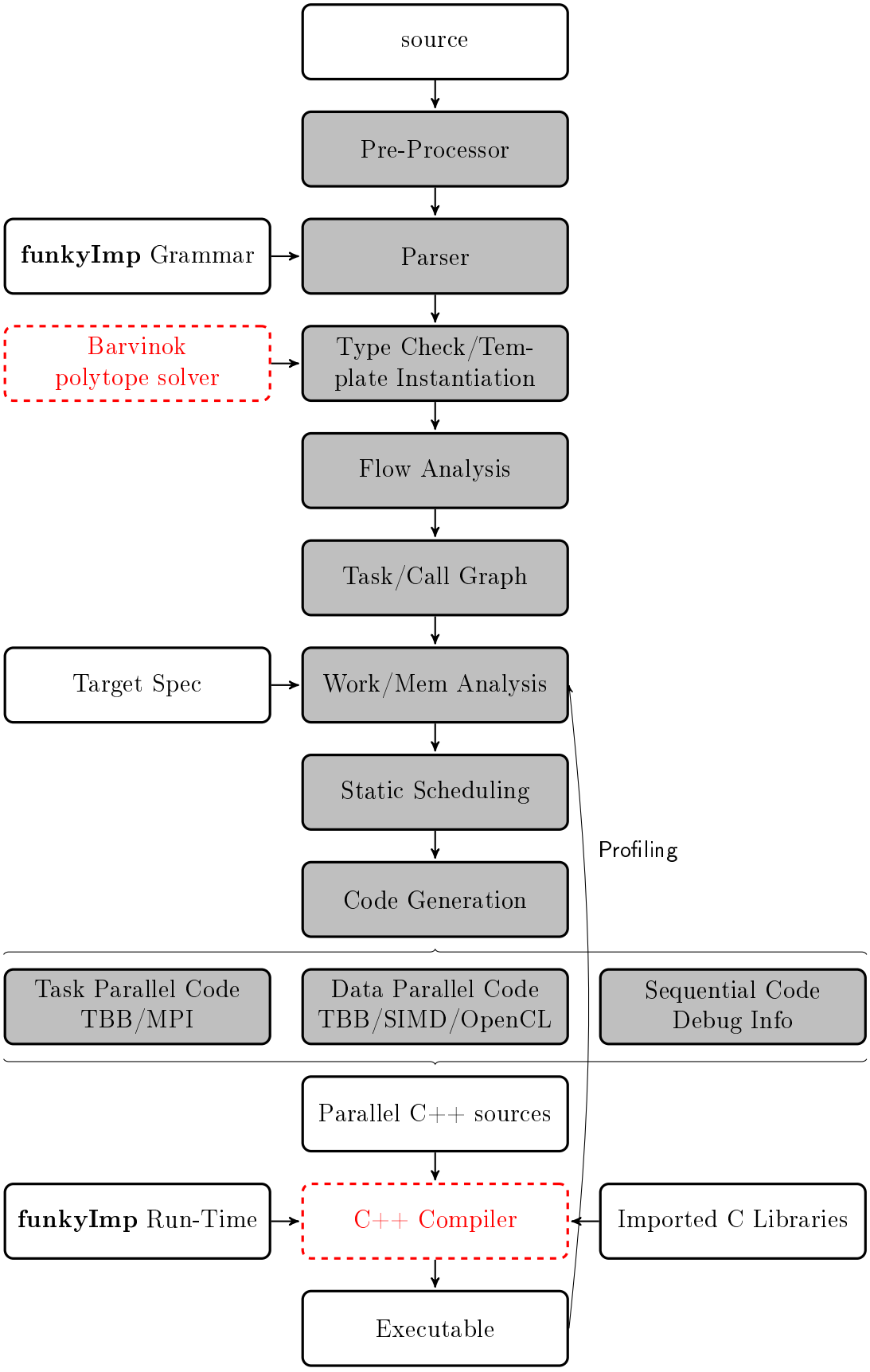

Figure 7.1: ***funkyImp*** *compilation components and work flow. The actual compiler consists of the gray components, dashed components refer to external tools. The remaining components denote input and output files.*

Finally, the compiler back-end emitting Java byte code was replaced by a back-end that is divided in three parts, emitting sequential, data- and task-parallel C++ code. By using C++ code as target for our compiler, we benefit in multiple ways:

- no virtual machine layer obscuring performance results exists
- the sequential pieces of code inside the tasks benefit from the existing C++ compiler optimizations
- unlike Java byte code, data-parallelism can be readily expressed and exploited
- the whole IDE and development tool chain, including source level debugging and low level profiling, are available for our language
- similarity of the source and target language simplify the translation
- we can leverage the existing library system and can interface with existing C code

When generating C++ code, the foreign function interface to C reduces to declaring the function to be imported inside **funkyImp** and including the proper header as well as library into the generated C++ code. Generally, we feel that the proliferation of functional languages, like Haskell or Ocaml, is hampered by the lack of strong tool support which enables debugging and profiling with the same level of sophistication as the tools available for imperative languages. By using a simple, single assignment language and targeting C++, we evade most performance problems of functional languages and inherit all available C++ tools for free.

We have implemented several back-ends to generate code for the different parts of the run-time system.

- Thread Building Blocks [67] based code generation to exploit shared memory task-parallelism and to distribute large data-parallel computations onto tasks
- Vectorization pragmas [107] to exploit data-parallelism on the CPU (SIMD)
- OpenCL [54] kernel and client code generation to run data-parallel computations on GPUs [118]
- MPI [108] code generation to exploit task-parallelism with explicit communication

7.3 Run-Time System Implementation

The generated parallel C++ code is linked by a C++ compiler to our run-time system and other, imported libraries to produce the final executable. Our run-time system consists of a library wrapping TBB, OpenCL, MPI and the Boehm garbage collector [19] for automatic memory management.

7.3.1 Thread Pools

The TBB library uses a thread pool to limit the number of threads which are created on a system. In contrast, when executing task graphs, as generated by the algorithm in Chap. 3, the number of concurrently executing tasks is not strictly limited. Running more threads than the hardware can execute in parallel may reduce the overall throughput of the systems because the OS context switches between threads contesting for processors and context switches are expensive [37, 87, 103]. Also, the cost of thread creation is high.

So instead of starting a thread for every task in the task graph, thread pools can be used. Thread pools preallocate a set of worker threads (the number of threads is usually in the order of the number of available cores) and the threads execute tasks which are ready. Ling et al. [89] have investigated the relation between the thread pool size (number of available worker threads) and execution performance. Since the number of threads is bound and the number of tasks may exceed this bound, some ready tasks may have to wait in case all worker threads are busy.

Thus, on the one hand, thread pools can improve performance by reducing context switches and placing a bound on the number of thread creations. On the other hand, the bound on concurrently executing threads may hinder progress if a task graph contains *blocking* tasks which run indefinitely or tasks that wait for some signal from other tasks without a data or control flow dependency between these tasks.

7.3.2 Blocking Tasks

Usually, only the data flow and control flow dependencies between tasks are guaranteed when executing a task graph. So tasks waiting for other signals may start execution before the task that produces the signal starts. This is not a problem in an environment where the number of concurrently executing tasks is unbounded because the task producing the signal will eventually run. In an environment with bounded concurrency, the system will dead-lock if all available threads are busy executing tasks which are waiting for signals from tasks that cannot execute because all worker threads are busy.

To avoid this problem, *blocking* tasks must be scheduled on additional threads so that they do not interfere with the worker threads. Our language provides only a limited set of methods to create *blocking* tasks:

- infinite (tail) recursion
- foreign function interface

Recursive functions are detected by cycles in the call graph. Recursion that does not terminate (or where termination depends on an external signal) is useful to create asynchronous background computation. For example, a loop that blocks for the next key press and communicates the result (via an actor) can be used to decouple blocking key reads from the rest of the program. So only the part of the program reading the keys actually blocks, the rest can continue in parallel. Foreign functions can do anything and thus may also block execution.

Foreign functions must be annotated by the programmer with a qualifier which determines whether they are blocking or not. Recursive functions are automatically annotated as blocking. Since blocking tasks are executed on extra threads (rather than on the default worker threads), some overhead may be incurred because the extra threads may need to be created on demand.

Assuming that recursive functions are always blocking is an over-approximation. We cannot automatically prove termination, so we assume non-termination which is sound. For recursive functions, the programmer can overwrite this behavior by specifying that the function is non-blocking. This removes the overhead associated with blocking tasks. Since we cannot verify the programmers claim that the function terminates, progress of the program can be impaired by incorrectly annotated blocking functions as non-blocking. The same holds for incorrectly annotated foreign functions.

7.3.3 Garbage Collection

Sometimes, garbage collectors are criticized to hamper parallel performance. When not configured properly, the allocation routine of most garbage collectors uses a global lock to safeguard from concurrent allocation and stop the world for the complete garbage collection process. Nevertheless, modern garbage collectors, including the Boehm collector, can be configured to run in a parallel environment so that each thread allocates from his private memory pool and collection runs in parallel to the mutator as much as possible.

We have run a set of experiments comparing manual reference counting schemes and garbage collection. We have not encountered a single instance for our experiments in [23] nor for the experiments presented in this thesis, where the garbage collector could be considered the bottleneck or manual reference counting would show a viable benefit. Bigger examples that are intended to stress the garbage collector are necessary to investigate if and under which conditions garbage collections becomes problematic in a parallel environment.

Chapter 8

Conclusion

We have presented **funkyImp**, a data-race and dead-lock free implicit parallel language that offers performance competitive with hand written C++ code on shared memory systems and automatically adapts the parallel decomposition to specific target systems. The language solves many of the problems inherent to manual parallelization, which we have elaborated in Sec. 1.1.

Obviously, the correctness of our compiler hinges on the fact that the compiler transformations must be correctly implemented. This can be done using formal proof assistants like Coq [15] which was used to verify a large subset of the C-standard in CompCert [86].

Syntactically and in terms of tool support, our language is closely related to well established high performance languages like C++. Const-correct type discipline and controlled aliasing using smart pointers is common in today's C++ code. Therefore, the barrier to switching from C++ to our language where aliasing is controlled with unique types should be low and is not obscured by a complex type system, complex evaluation rules and a lack of performance as introduced by some of the available functional languages. The simple and high performance interfacing with existing C libraries allows to slowly migrate existing projects to our language. Finally, a large incentive to switch to **funkyImp** is given by its strong security guarantees, the compose-ability of program parts and automation in terms of static task and data-parallel programming without sacrificing the performance offered by C++. In this sense, our language combines two of the main benefits of imperative and functional programming, namely safety by construction and performance.

Instead of pressing all programs into a single paradigm (e.g. everything is an actor), we have introduced a set of language features that allow to address the different kinds of parallelism exposed by today's hardware explicitly. Hence, the programmer need not rely on a set of increasingly complex compiler optimizations to achieve good performance. All relevant forms of parallelism may be addressed by combining the presented language features. In the future, new forms of hardware parallelism and synchronization features may emerge and may require to add more language features in order to support these efficiently.

Furthermore, there are a couple of loose ends in the current incarnation of **funkyImp**. In principle, we would like to be able to express any correct algorithm that may be expressed in C++ without performance penalty. Nevertheless, some limitations in regards to expressibility remain in order to not violate our safety guarantees.

In future work with regards to expressibility we would like to investigate ways to enable:

- unique graphs (currently not allowed by plain unique types)

- error handling using exceptions or a new mechanism using actors (exception scopes may hinder parallelization)
- efficient persistent partial array updates to allow partial updates of non-unique arrays

Next to limits in expressibility, there are limits to the automatic parallelization in **funkyImp** we would like to overcome in the future. Our language allows a precise extraction of task-parallelism from a given implementation and we have presented a static scheduling algorithm to achieve good results on homogeneous shared memory machines. In the future, we would like to extend our scheduling mechanism to automatically target inhomogeneous computer clusters and GPUs. This requires a good cost model for data transfer and execution on the respective hardware and a corresponding scheduling algorithm that utilizes the cost model.

In this thesis, we have placed a strong emphasis on static methods to extract and optimize parallelism. While precise extraction and good static optimization of parallelism is a prerequisite to achieve good parallel performance automatically, not all behavior can be predicted statically. Especially imprecision in the estimation of task sizes may hinder an optimal solution. Profiling can be used to find the average task size with high precision but this method is elaborate and oblivious to dynamically changing processes. In cases where the dynamic program behavior must be taken into account, our language offers high quality input to dynamic task graph optimization [130].

Furthermore, we would like to investigate methods to simplify the debugging and verification of non-deterministic actor computations. While **funkyImp** programs including actors are data-race and dead-lock free by construction, actor computations are prone to unintended non-determinism and non-termination. Actor networks may even expose concurrency issues if concurrent computations of an actor network are performed without proper data separation between the different computations.

Finally, we would like to implement bigger test applications like a data base or a ray tracer in order to stress our implementation and explore further compile-time and run-time optimizations to deal with potential scalability issues.

We are currently in the process of comparing the effort and performance results of students developing a particle simulation in both C++ and **funkyImp**.

Part II

Prediction of Parallelization Hazards

Chapter 9

Model based Parallelization

In the software industry, there exist large legacy code bases of sequential imperative code that must be turned into parallel code in order to benefit from future hardware development. As discussed in the first part of this thesis, manual parallelization of imperative code is prone to many subtle errors and may require experimentation with many different parallelization strategies to yield a benefit. This makes turning sequential legacy code into parallel code a highly expensive and risky endeavor. There is a lot of interest from the industry to support research which allows to reduce the associated risk and costs.

In this chapter we will present a framework that can be used to predict parallelization hazards in existing sequential software. The framework can be used to evaluate different parallelization strategies before implementing the strategies, thus drastically reducing risk and costs of failed parallelization attempts. This chapter is structured as follows. First, we present a model that allows to define parallelization strategies for sequential code in Sec. 9.1. Then, in Sec. 9.2, we present an example problem from an industry partner before we derive a class-modular points-to analysis from the example. In Sec. 9.3, the impact of class-modular points-to analyses is evaluated before we present a working example for the analysis in Section 9.4. After describing the abstract semantics of our analysis in Sec. 9.5, we define the language our analysis operates on in Sec. 9.6 before we prove the soundness of the analysis in Sec. 9.7. Finally, we present our implementation of the analysis and benchmarks in Section 9.8. In Section 9.9 we summarize our findings.

9.1 Modeling Parallelization

Before implementing a parallelization strategy, the responsible programmers must build a mental model describing which parts of the software are supposed to run in parallel and which shared resources must be protected from concurrent execution in order to achieve a correct and efficient parallelization. Due to the large size of the code base, some shared resources that would need to be protected might be overlooked. The additional protection required for the missed resources may make the whole model infeasible. Often, the insufficiencies of the model become apparent only after the model was implemented. Usually, the process of refining the model (which takes the newly discovered unprotected resources into account) and implementing it is iterated until a sufficient solution is found.

Instead of implementing models to determine their fitness, we propose to formalize the model and use a static analysis to find missed resources and thus refine the model without a costly implementation.

The model needs to define memory regions M_i where consistency (non-con-current

access) should be maintained after parallelization. For example, a memory region might be defined by all data reachable from a pointer created on a specific program point. Furthermore, a set of code regions which may not execute concurrently and which may depend on the memory regions $E(M_i) = \{C_1, .., C_n\}$ must be defined. Implicitly, any code region C_i contained in $E(M_i)$ is non-concurrent to itself.

Together, memory regions and mutually exclusive code regions define a parallelization strategy. The memory regions M_i and code regions $E(M_i)$ define what memory and which functions should be protected from concurrent access in an implementation of the strategy, e.g. by using locks or any other suitable protection method. We assume that the rest of the legacy code, which is not in E, is executed sequentially but asynchronously to the parallel code regions in E.

Now, for every memory region M_i, a may-points-to analysis can be used to find accesses to M_i that are outside mutually exclusive code and thus constitute a data-race which destroys the semantics of the parallel program. Often, it is sufficient to find pointers to M_i that escape the protected code regions and will constitute a data-race when accessed later to vastly reduce the amount of code that must be analyzed. In case data-races are found, the model must be refined, by either extending $E(M_i)$ to include the offending code region or reducing M_i to exclude the offending memory region or joining different memory regions into a single, larger region.

After refining the model to the point that the analyses finds no more errors, the strategy guarantees race freedom and consistent (non-concurrent) access to each memory regions M_i. Therefore, the risk of implementing a model that has insufficient protection of consistent access for all M_is is effectively removed. The performance modeling introduced in Part I, Chap. 3 may be used to predict the performance of a model implementation.

After having selected a provably correct model using our framework, the model can be implemented and afterwards static analysis may be used to verify that the implementation adheres to the model: Any accesses to memory that may-alias with an M_i must be protected by locks which must alias.

Any structure available in M_i and $E(M_i) = \{C_1, .., C_n\}$ may be used to refine generic may- and must-points-to analyses for better performance and precision. In the next section we will discuss an application of our framework on an industrial application in which we could drastically improve the efficiency of the may-alias analysis by taking into account the structure of M_i and $E(M_i)$.

9.2 Example Problem

As an example we consider the embedded, sequential C++ code for a mobile phone relay station from an industry partner. The relay coordinates one large state machine (which consists of several smaller, internal state machines) per user which implements various communication protocols. Our partner assumed that all internal state machines (per user and over users) share no state and thus messages being processed by different state machines may be executed in parallel without changing the program semantics. Due to the large and interdependent code base, manual verification of this assumption proved futile, instead our framework was applied to show that the assumption was indeed wrong for lower level state machines. A refinement of the parallelization model to parallelize at the level of whole state machine of a single user was shown correct and prototype implementations of the pattern proved promising so that the model was subsequently implemented. We have not verified the implementation.

9.2.1 Example Model

The initial model defines M_i as any instance of a class derived from the state machine base class and $E(M_i)$ contains all member functions in the class hierarchy of M_i. Hence, all function calls on different instances of a state machines would run in parallel whereas functions calls on the same instance are mutually exclusive.

The points-to analysis required to verify the correctness of the model can by simplified from a whole-program analysis to an intra-class hierarchy escape analysis by taking into account the structure of M_i and $E(M_i)$. Since each M_i consists of the (private) memory transitively reachable from the pointer on a class instance and all member functions are in $E(M_i)$, so that they are protected from running concurrently, any pointers to the class's memory (M_i) escaping from the class's functions ($E(M_i)$), will produce a date race if they are read from or written to anywhere outside the class's code. Therefore, it is sufficient to analyze each state machine's class hierarchy instead of the whole program to find all possible data-races.

In the following we will discuss the class-modular points-to and escape analysis we have developed for this purpose in detail. The analysis was published in [60].

9.3 Class-modular Points-to and Escape Analysis

Accessibility of an object from different program parts is important information for optimizing compilers and verification tools. This information can be inferred using may points-to analyses. Often, not all code that uses a class declaration is available to analysis because program modules are compiled independently and linked dynamically at run-time. In order to apply optimizations or verification in this scenario, points-to information for a class must be inferred in isolation from the code that uses the class. Such class-modular points-to information cannot be obtained from existing whole-program points-to analyses as these expect the complete program as input. In contrast to whole-program analyses, modular analyses can abstract different program parts independently. Commonly, individual methods are abstracted without calling-context. Later, these method-summaries are instantiated with calling-context information, so that eventually the context information and the summaries of the whole program are combined. Instantiating method-summaries with unknown context information (e.g. for class methods without the code that calls the method) yields imprecise results for the *this*-pointer and all other method parameters. Therefore, it is not enough to use method-summary based analyses.

To solve this problem, we present a framework that transforms a common whole-program or summary-based points-to analysis into a class-modular points-to analysis. Given a sound plug-in analysis, the transformed analysis is sound and may be useful even if the whole program is available. The analysis time can be reduced by analyzing class declarations independently or in parallel, mostly without loosing much precision.

As a side-effect, the transformed analysis produces class-escape information. Escape analysis as presented by e.g. Blanchet [16] determines which local objects escape from a method as only local objects that do not escape can be considered truly local to that method and may be stack allocated. In contrast, our framework extends the scope from methods to classes. Local variables, *private* fields and locally used heap objects are considered class-local if and only if they can never become accessible from outside the class. If a local variable, a *private* field or a locally used heap object can possibly become accessible from outside the class (e.g. a pointer to it escapes through one of the *public*

class methods) then it is considered *class-escaped*. This class-escape information can be used to improve other analysis (e.g. object in-lining), which depend on object accessibility information but commonly rely on a whole-program or summary-based analysis.

We have implemented an instance of the analysis in the Goblint [140] framework showing that it can handle large classes from industrial and open source C++ code in seconds. Our contributions in this chapter are

- we present the combined class-modular, class-escape and points-to analysis based on encapsulation that is fully independent from the code that uses the analyzed class,
- we present a framework that allows to transform common points-to analyses into a class-modular, class-escape and points-to analysis,
- we prove the soundness of the transformed analysis,
- we present an implementation and a set of benchmarks applying an instance of the analysis to large, real world code in seconds.

Related Work.

Many of the points-to analyses are based on work from Steensgaard [128] and Andersen [6] which are neither class-modular nor deal with class-escape information. Abstract interpretation based modular analyses in general is described by Cousot and Cousot [34], where program modules can be analyzed independently but a completely unknown (worst case) context is assumed and access modifiers are not taken into account.

Rountev [121], Cheng and Hwu [28], Horwitz and Shapiro [64] present pointer analyses that are modular on the function level but require additional information from the function's calling context. Whaley and Rinard [143] present a compositional pointer and (method-)escape analysis. They need information from the analyzed methods' calling context as their analysis is not on the class-level. The precision of non-escaped objects cannot be as good if the class methods are analyzed separately, neglecting the object state information available through the access modifiers. Rountev and Ryder [122] present a similar approach, assuming worst case information on the function level.

The partitioning of fields into escaping and non-escaping fields performed by our analysis relates our analysis to region analysis where mutually unreachable heap regions are identified. The non-escaping field set represents a heap region that is unreachable from outside the class declaration and the resulting points-to information directly represents which of these fields can reach outside the region. Region analysis for C programs was recently investigated by Seidl and Vojdani [126]. These region analyses are not class-modular and often the pointer information is undirected and less precise.

Boyapati, Liskov, and Shrira [21] have applied ownership type-systems to verify encapsulation and alias protection properties of object-oriented programs. These type-systems heavily rely on annotations and restrict the programming language they can be applied to, as e.g. iterators are not easily incorporated. The ownership property verified in these systems is more restrictive than our class-escape property.

To the best of our knowledge, this is the first presentation of a class-modular, points-to and class-escape analysis. Class-level modular static analysis of classes and class methods that automatically infer class invariants have been proposed by Logozzo [94]. For the analysis to be sound it is required that accessibility of object internal state (to code that

is outside the analyzed class) is detected by another static analysis. The suggested whole-program escape analysis from Blanchet [16] uses the different notion of method-escaping rather than class-escaping and cannot be applied when only the class declaration is given. As such, it does not provide the accessibility information required for Logozzo's analysis. Instead, the class-escape information provided by our analysis can be used.

Porat et al. [117] present a mutability analysis for Java that can handle missing class definitions and utilizes access modifiers. They give no details on their state accessibility analysis. It is not clear whether their state accessibility analysis is sound nor how it works.

Based on our analysis class-modular object in-lining [42] optimizations for garbage collected languages can be implemented. The life-time of fields which do not escape a class is limited to the life-time of the enclosing object. Since non-escaping fields are guaranteed to be non-accessible from outside a class it is sufficient to modify the code of the class itself to in-line an object, so method cloning is not required and the optimization is modular. JIT compilers could benefit from similar improvements [144].

9.4 Example Program

In this Section a C++ example is shown where pointer assignments in one method of a class have a non-local effect which is visible in another method and how the analysis handles this information.

The *private* fields of a class can become accessible from external code if a pointer to such a field escapes the class, for example as a return value as shown in line 27 in the example in Fig.9.1.

In the constructor of *Rect*, an instance of *Point* which we denote as Pt_b for convenience, is assigned to *lr*. Then the address of *lr* is assigned to *e* and *e*'s address is assigned to *p* in turn. This is denoted as edges leading from *p* to *e* to *lr* and finally to Pt_b in Fig.9.2, where an edge from node a to node b denotes that b is in the points-to set of a.

Within DoEscape from line 22 to line 26 Pt_b is escaped through various routes as noted in the comments of Fig.9.1. Especially interesting is line 23, here the pointer *pr* may be equal to the current instance *this* or another instance of the class. So Pt_b may be assigned to a field from *this* if *pr* equals *this*. Otherwise, Pt_b escapes because it is assigned to an external variable. In line 27 *e* is returned, so the content of *e* and everything reachable thereof may escape. Generally, an object escapes when its address may be stored in an externally accessible object.

Analysis Overview.

In order to collect all escaped pointers in the points-to set of the variable *a_ext*, our analysis proceeds in three steps. First an instance *a_this* of the class is created and all public fields of the class are considered escaped. The created instance serves as representative object for this class. Then the effect of calling *any* constructor is over-approximated on the representative object *a_this*. Finally, all possible combinations of *public* method calls on *a_this* are simulated.

When applied to our example class *Rect*, the first step produces points-to information telling us that *a_ext* may point to itself, *a_this*, or *pub* — these are considered class-escaped.

```
1  class Point{ public: int x,y; };
2  extern void unknown(Rect* pr);
3  class Rect
4  {
5  private:  Point *ul,*lr;
6         Point **e,**l;
7         Point ***p;
8         Point *priv;
9  public:  Point *pub;//escapes
10
11 Rect(int x1,int y1, int x2, int y2)
12 {
13   ul=new Point();//Pt_a
14   lr=new Point();//Pt_b
15   p=&e;e=&lr;l=&ul;
16   ul->x=x1;ul->y=y1;
17   lr->x=x2;lr->y=y2;
18 }
19
20 Point** DoEscape(Point**v,Rect* pr)
21 { //pt_b escapes in the following:
22   pub=lr;//copied into public var
23   pr->priv=lr;//maybe copied into other
       instance
24   unknown(lr);//passed to unknown fun
25   *v=*e;//copied into external var
26   **p=lr;//copied into lr, becomes external
       in next line
27   return e; //lr, Pt_b escape (returned from
       public fun)
28 }
29 };
```

Figure 9.1: C++ Example Code. Sound points-to information for class Rect in the absence of the code that uses class Rect is generated. The field e is assigned the address of lr in line 15, so when the content of e is escaped in line 27 then lr and the instance of Point which lr is pointing to are escaped. An object escapes when its address is stored in an externally accessible object. The points-to information of fields from Rect is a global property, points-to relations set up in one method are retained when another method is called.

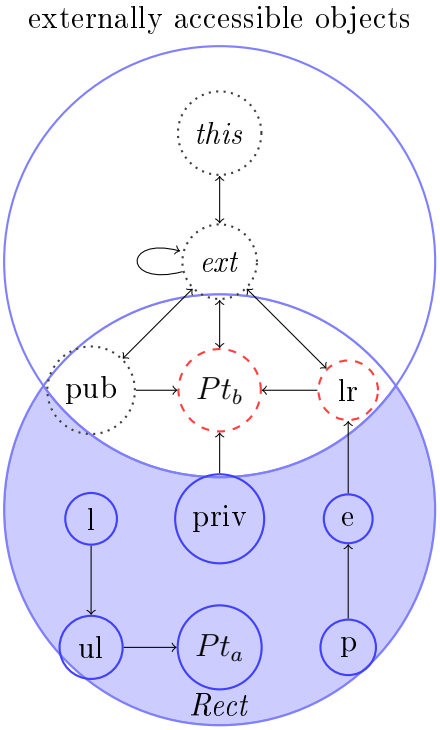

Figure 9.2: Graph based representation of the final domain state for class Rect for the program on the left after the analysis has finished, dashed nodes represent escaped objects. Dotted nodes are externally accessible before any method from Rect is called. An edge from symbol a to symbol b means that b is in the points-to set of a. All nodes reachable from ext are also connected with each other, this is not shown for clarity. All addresses except this that are neither declared nor allocated inside Rect are abstracted to ext. this is an external instance of Rect for which the class-invariant is generated.

In the next step we need to apply the effect of any constructor to our abstract state. As our example only has one constructor, only the effect of that constructor is applied.

Finally, we simulate all possible *public* method calls on *Rect*. Class *Rect* only has one (*public*) method DoEscape, but this method could be applied several times on the same object (while changing the escaped objects between each call). Therefore, the effect of $a\_ext = a\_this \rightarrow \text{DoEscape}(a\_ext, a\_ext)$ is computed until the smallest fix-point for a_ext and the fields of a_this is reached. In the first iteration, Pt_b class-escapes on line 22, 23 and 25, because a pointer to Pt_b is assigned to a potentially externally accessible variable. In line 24, Pt_b class-escapes because it is given as an argument to an unknown

function. Eventually, lr and Pt_b class-escape in line 27, because they are returned. In the second iteration, Pt_b also escapes in line 26 because it is assigned to lr, which has escaped in the previous iteration. This last step reaches the fix-point and the result is shown in Fig. 9.2. At the end of each iteration step, all escaped pointers are modified so that any escaped object may point to any other escaped object.

9.5 Abstract Semantics

Our framework transforms a given plug-in points-to analysis from whole-program or summary-based analysis to a class-modular class-escape analysis that can analyze a given class without any context information on how the class may be used. To achieve this, the domain of the plug-in analysis is extended by our own global domain $\mathbb{G}'$. The semantics of our analysis is defined by lifting the plug-in semantics to be able to handle the extended domain and the unknown context in which the class may be used in. A special address *ext* that abstracts all addresses which may exist outside the analyzed class definition is introduced. Furthermore, an address *this* is created that together with our global domain abstracts all possible instances of the analyzed class.

The concrete language our analysis operates on and its semantics is given in Appendix 9.6. Any points-to analyses which adhere to the set of requirements given in this section can be used as plug-in analysis for our framework. Eventually, our analysis inherits the properties from the plug-in analysis while making the analysis class-modular and calculating sets of maybe class-escaping objects which are collected in the points-to set of *ext*.

This section is structured as follows. First, we list the requirements for the plug-in points-to analysis to be suitable for this framework. Afterwards, we define the necessary functions to lift the plug-in semantics $[\![s]\!]^\sharp$ to the abstract semantics $[\![s]\!]^{\sharp'}$ of our analysis. Finally, we give a set of initialization steps and a set of constraints that must be solved in order to perform the analysis.

Let $val^\sharp$ be the abstract values and $addr^\sharp \subseteq val^\sharp$ the abstract addresses used by the plug-in analysis.

Let $\mathcal{A} : \mathbb{D} \to lval \to \mathcal{P}(addr^\sharp)$ be a plug-in provided function that calculates the set of possible abstract addresses of a l-value given an abstract domain state $\rho^\sharp$.

Let $[\![s]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ be the abstract semantics of the plug-in points-to analysis for the statement s. The complete lattice $\mathbb{D}$ is the abstract domain used by the points-to analysis. From that we construct $\mathbb{D}' = \mathbb{D} \times \mathbb{G}'$ — the domain of our analysis, where $\mathbb{G}' : addr^\sharp \to \mathcal{P}(val^\sharp)$ extends the global (flow-insensitive) domain of the plug-in analysis.

Furthermore, let $q : \mathbb{D} \to addr^\sharp \to \mathcal{P}(val^\sharp)$ be a plug-in provided function that calculates the set of possible abstract values that may be contained by the memory at the given address when provided an abstract domain state and $\forall \rho^\sharp \in \mathbb{D} : q_{\rho^\sharp}\ \mathbf{null} = \emptyset$.

Intuitively, the plug-in analysis maintains some kind of mapping from abstract addresses to sets of abstract values for each program point abstracting the stack and the heap. How exactly this information is encoded inside the plug-in domain $\mathbb{D}$ is not relevant for our analysis. All addresses are initialized to **null**, so if an address a has not yet been written to in $\rho^\sharp$ then $q_{\rho^\sharp}\ a = \{\mathbf{null}\}$.

Finally, $\rho_1^\sharp = \rho_0^\sharp[x \to Y]$ denotes the weak update of $\rho_0^\sharp \in \mathbb{D}$ such that:

$$\forall z \in addr^\sharp : q_{\rho_1^\sharp}\ z \sqsupseteq \begin{cases} q_{\rho_0^\sharp}\ z \cup Y & : z = x \\ q_{\rho_0^\sharp}\ z & : \text{else} \end{cases}$$

Using this notation we can perform weak updates on the plug-in domain without knowing the details of $\mathbb{D}$.

For example the summary-based points-to and escape analysis from Whaley and Rinard [143], like virtually all other sound points-to analyses, fulfills all our requirements and can be plugged into our framework and thus become a class-modular, points-to and class-escape analysis.

To shorten the notation we also define a function $Q : \mathbb{D} \to \mathcal{P}(addr^\sharp) \to \mathcal{P}(val^\sharp)$ for sets of addresses.

$$Q_{\rho^\sharp}\ S \triangleq \bigcup_{x \in S} q_{\rho^\sharp}\ x$$

The function $Q^*_{\rho^\sharp} : \mathbb{D} \to \mathcal{P}(addr^\sharp) \to \mathcal{P}(val^\sharp)$ defines the abstract reachability using $Q_{\rho^\sharp}$:

$$Q^*_{\rho^\sharp}\ S \triangleq F \cup Q^*_{\rho^\sharp}(F)$$
$$\textbf{where } F = Q_{\rho^\sharp}\ S \cup \bigcup_{\substack{c \in Q_{\rho^\sharp} S \\ f_i \in public\ fields\ of\ c}} \mathcal{A}_{\rho^\sharp}(c \to f_i)$$

The analysis is performed on a given class which we will call *Class*. Before starting the analysis an instance of *Class* is allocated and stored in the global variable *a_this* which we assume is not used in the analyzed code. Also, a global variable called *a_ext* of the most general pointer type (e.g. Object for Java or void* for C++) is created using the plug-in semantics (*a_ext* is also assumed not to be used in the analyzed code):

$$\rho_0^\sharp = (\llbracket a\_\ this := new\ Class \rrbracket^\sharp \circ \llbracket a\_\ ext := \textbf{null} \rrbracket^\sharp)\ d_0^\sharp$$

where $d_0^\sharp$ is the initial state of the plug-in domain, before any code has been analyzed. At this stage of the analysis *new* does not execute any constructors. As both *a_this* and *a_ext* are not used within the analyzed code, they do not change the semantics of the analyzed code and our lifted semantics can use these variables to communicate with the plug-in analysis and store special information as explained in the following.

The set *this* which contains all possible addresses of the allocated *Class* is defined as:

$$this = Q_{\rho_0^\sharp}(\mathcal{A}_{\rho_0^\sharp}(a\_\ this))\,.$$

The value *this* is meant to abstract all instances of *Class* that can exist (for the plug-in, *this* is an instance of *Class* that cannot be accessed from the program unless our analysis provides its address). The plug-in analysis should be field sensitive at least for the *Class* instance addressed by *this* in order to exceed the precision of other points-to analyses when an unknown context is used.

The set *fields* contains all addresses of the *public* fields from the *Class* instance *a_this*:

$$fields = \bigcup_{f_i \in public\ fields\ of\ Class} \mathcal{A}_{\rho_0^\sharp}(a\_\ this \to f_i)\,.$$

Since the analysis is class-modular, only class declarations are analyzed. Hence, most of the program code is hidden from the analysis. We differentiate program segments which are visible to the analysis and the rest by defining external code:

Definition 1 (External Code)**.** *External code with respect to a class C denotes all code that is not part of the class declaration of C. If no class C is stated explicitly, then class Class is assumed.*

The points-to set of *ext* abstracts all addresses accessible from external code.

$$ext = \mathcal{A}_{\rho_0^\sharp}(a\_\ ext)$$

Initially, only *ext* itself, *this* and the *public* fields from *this* are reachable from external code, so *ext* must point to itself, *this* and the *public* fields, as an instance of *Class* may be allocated in code external to *Class*.

$$\rho_1^\sharp = \rho_0^\sharp[a \to ext \cup this \cup fields \mid a \in ext] \tag{9.1}$$

As the points-to set of *ext* contains multiple distinct objects, only weak updates can be performed on *ext* by the plug-in analysis.

Fields from $a\_\ this$ and their content become member of $Q_{\rho^\sharp}\ ext$ during the analysis if they may escape the *Class*. So after the analysis has finished, all possibly escaped memory locations are contained in $Q_{\rho^\sharp}\ ext$, all other memory locations do not escape the *Class* and are inaccessible from external code.

In the following we describe how the plug-in semantics $[\![s]\!]^\sharp$ is lifted to produce the abstract semantics $[\![s]\!]^{\sharp'}$ of our class-modular class-escape analysis:

The global addresses are constituted by *ext* and the fields of *this* since modifications of these addresses' values are observable inside different member methods of *this*, even if these methods do not call each other. For example, a method from *Class* may return an address to external code which was not previously accessible by external code. Later, external code may invoke a method from *Class* passing the newly accessible address (or something reachable thereof) as parameter to the method.

$$global : \mathcal{P}(val^\sharp)$$
$$global \triangleq\ ext \cup fields$$

Given the state of the plug-in domain, $globals : \mathbb{G}' \to \mathbb{D} \to \mathbb{G}'$ calculates the new state of the global domain $\mathbb{G}'$:

$$globals\ g^\sharp\ \rho^\sharp\ x \triangleq \begin{cases} q_{\rho^\sharp}\ x \cup g^\sharp\ x & : x \in global \\ \emptyset & : \text{else} \end{cases}$$

The global domain state tracks modifications to fields of $a\_\ this$ between different invocations of *Class*-methods from external code.

modify over-approximates the effects of code external to *Class*. In a single-threaded setting, these effects cannot occur inside code of *Class* so *modify* is applied when leaving code from *Class*. This happens either when returning from a *public* method to external code or when calling an unknown method. We assume that all methods from *Class* are executed sequentially. If other threads (that do not call methods from *Class*) exist, then *modify* must be applied after every atomic step a statement is composed of, as the other threads my perform modifications on escaped objects at any time. If no additional threads exit, then modifications of escaped objects can happen only before a method from *Class*

is entered, when an external function is called and after a method from *Class* is exited. As external code may modify all values from addresses it can access to all values it can access, *modify* ensures all possible modifications are performed.

$$\begin{aligned} &\mathit{modify} \quad : \mathbb{D} \to \mathbb{D} \\ &\mathit{modify}\ \rho^\sharp \triangleq\ \rho^\sharp[x \to Q^*_{\rho^\sharp}\ \mathit{ext} \mid x \in Q^*_{\rho^\sharp}\ \mathit{ext}] \end{aligned}$$

The following semantic equation is inserted into the semantics $[\![s]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ of the plug-in analysis (or it replaces the existing version).

$$\begin{aligned} [\![l := e_0 \to m^{\text{extern}}(e_1, \ldots, e_n)]\!]^\sharp_{\rho^\sharp} \triangleq ([\![l := a\_\ \mathit{ext}]\!]^\sharp \circ \mathit{modify} \circ \\ [\![\mathbf{deref}(a\_\ \mathit{ext}) := e_0]\!]^\sharp \circ \ldots \circ [\![\mathbf{deref}(a\_\ \mathit{ext}) := e_n]\!]^\sharp)\ \rho^\sharp \end{aligned}$$

Methods m^{extern} are called from within *Class* but are not analyzed (e.g. because the code is not available). This makes the analysis modular with respect to missing methods in addition to its class-modularity. The procedure *unknown*, which is called in line 24 from our example in Fig. 9.1, represents such an external method where the above rule applies.

As shown in Theorem 15 and 16 in Appendix 9.7, reading from $a\_\ \mathit{ext}$ and writing to $\mathbf{deref}(a\_\ \mathit{ext})$ correctly over-approximates reads from non-class-local r-values and writes to non-class-local l-values.

Finally, we give the abstract semantics $[\![s]\!]^{\sharp'} : \mathbb{D} \times \mathbb{G}' \to \mathbb{D} \times \mathbb{G}'$ of our analysis for a statement s.

$$\begin{aligned} [\![e_0 \to m(e_1, \ldots, e_n)]\!]^{\sharp'}_{(\rho^\sharp, g^\sharp)} \triangleq\ & (\rho_2^\sharp, \mathit{globals}\ g^\sharp\ \rho_2^\sharp) \\ & \mathbf{where}\ \rho_2^\sharp = \mathit{modify}([\![\mathbf{deref}(a\_\ \mathit{ext}) := e_0 \to m(e_1, \ldots, e_n)]\!]^\sharp_{\rho_1^\sharp}) \\ & \quad \mathbf{and}\ \rho_1^\sharp = \rho^\sharp[x \to g^\sharp\ x \mid x \in \mathit{global}] \end{aligned}$$

Our transfer function is invoked only for top-level methods when solving the constraint systems for the analysis (see Eq. 9.3,9.4). First, the current state of the flow-insensitive fields and *ext* is joined into the plug-in domain. Then the plug-in semantics (which now contains our patched rule for m^{extern}) is applied and stores the return value of the method in $a\_\ \mathit{ext}$. Afterwards, *modify* is applied to over-approximate the effects of external code that may execute after the top-level method is finished. Finally, the new global domain state is calculated using *globals*.

Before starting the actual analysis, the effects of external code that might have executed before a constructor from *Class* is called are simulated by applying *modify*:

$$(\rho_i^\sharp, g_i^\sharp) = \mathit{modify}(\rho_1^\sharp, \mathit{globals}\ g_0^\sharp\ \rho_1^\sharp) \tag{9.2}$$

Here, $g_0^\sharp$ is the bottom state of the global domain. Then, the constructors are analyzed.

Since we know that only one constructor is executed when a new object is created, it is sufficient to calculate the least upper bound of the effects of all available constructors:

$$(\rho_c^\sharp, g_c^\sharp) = \bigsqcup_{m \in public\ constructor\ of\ Class} [\![a\_\ this \to m(a\_\ ext, ..., a\_\ ext)]\!]^{\sharp'} (\rho_i^\sharp, g_i^\sharp) \tag{9.3}$$

Afterwards, the *public* methods from *Class* with all possible arguments and in all possible orders of execution are analyzed by calculating the solution [125] of the following constraint system,

$$(\rho_f^\sharp, g_f^\sharp) \sqsupseteq (\rho_c^\sharp, g_c^\sharp) \tag{9.4}$$

$$\forall m \in public\ method\ of\ Class:$$
$$(\rho_f^\sharp, g_f^\sharp) \sqsupseteq [\![a\_\ this \to m(a\_\ ext, ..., a\_\ ext)]\!]^{\sharp'} (\rho_f^\sharp, g_f^\sharp)$$

in order to collect all local and non-local effects on the *Class* until the global solution is reached. $a\_\ ext$ is passed for all parameters of the method as it contains all values that might be passed into the method. For non pointer-type arguments the plug-in's top value $\top_{type}$ for the respective argument type must be passed as argument to the top-level methods. If the target language supports function-pointers then all *private* methods for which function-pointers exist must be analyzed like *public* methods, if the corresponding function-pointer may escape the class.

When inheritance and *protected* fields are of interest, the complete class hierarchy must be analyzed. If the language allows to break encapsulation then additional measures must be taken to detect this. For example, C++ allows friends and *reinterpret_ cast* to bypass access modifiers [131]. Friend declarations are part of the class declaration and as such easily detected. Usage of *reinterpret_ casts* on the analyzed *Class* can be performed outside the class declaration, so additional code must be checked. Still, finding such casts is cheaper than doing a whole-program pointer analysis. In other languages, e.g. Java, such operations are not allowed and no additional verification is required.

9.6 C++ Subset Language

In this Section the grammar for the language our analysis operates on is given. Furthermore, the program state, the concrete semantics of the language and the semantics of access modifiers is defined.

First we define an object oriented language (Sec. 9.6.1) which is expressive enough to model the peculiarities of common OO-languages like Java or C++. A program in our language is a sequence of class declarations. Statements in the language are similar to the C++ language [131]. The biggest difference being, that we only concern ourselves with pointers as we are interested in points-to and escape information. To simplify the discussion we neglect operators.

9.6.1 Grammar

$$
\begin{aligned}
\mathit{lval} &::= \mathit{var} \mid \textbf{deref}(\mathit{exp}) \mid \mathit{lval} \to \mathit{field} \\
\mathit{exp} &::= \mathit{lval} \mid \textbf{addrof}(\mathit{lval}) \mid \textbf{null} \mid \textbf{new}\ \mathit{class\_name} \\
\mathit{stmt} &::= \mathit{lval} := \mathit{exp} \mid \mathit{lval} := \mathit{exp} \to \mathit{method}(\mathit{exp}^*) \mid \textbf{return}\ \mathit{exp} \\
\mathit{block} &::= \mathit{stmt}; \mid \{\mathit{block}^*\} \mid \textbf{if}\ \mathit{exp}\ \textbf{then}\ \mathit{block}\ \textbf{else}\ \mathit{block} \mid \\
&\quad\ \textbf{while}\ (\mathit{exp})\ \mathit{block}
\end{aligned}
$$

$$
\begin{aligned}
A &::= \textbf{private} \mid \textbf{public} \\
\mathit{method} &::= A\ \mathit{class\_name}\ \mathit{method\_name}(\mathit{parameter}^*)\ \{\mathit{block}^*\} \\
\mathit{class} &::= \textbf{class}\ \mathit{class\_name}\ \{(A\ \ \mathit{field\_name} \mid \mathit{method})^*\} \\
\mathit{parameter} &::= \mathit{class\_name}\ \mathit{var}
\end{aligned}
$$

9.6.2 Program State

As usual, the program state is a partial function ($\text{Addr} \hookrightarrow \text{Addr}$) from addresses to values. Values, in this case, are also addresses, as the language does not allow to express any other kind of values. An address can refer to a variable, a ninstance of a class, or to a field inside an instance. We assume variables and fields can be distinguished by their name, and an instance of a class is identified by a unique natural number. Also, there are two distinct special address: **null** and Ret.

$$\text{Addr} = \text{Var} \cup (\mathbb{N} \times \text{Field}) \cup \mathbb{N} \cup \{\textbf{null}, \text{Ret}\}$$

The semantics of the update operator of a program state $\rho : \text{Addr} \hookrightarrow \text{Addr}$ where a value v is written to the address $x \neq \textbf{null}$ is defined as an overwrite, if the address was defined in ρ, or as an extension to the partial map when it was not defined previously. Note, that an update on the address **null** is undefined.

$$\rho[x \to v] = \rho' : \rho'(x) = v, \forall y \in \mathsf{dom}(\rho) : y \neq x \Rightarrow \rho(y) = \rho'(y)$$

9.6.3 Expression Semantics

Next we are going to co-recursively define l-value and r-value semantics, where l-value semantics is defined on *lval* expressions and r-value semantics is defined on *exp* expressions. The semantics of a l-value expression l and a r-value expression e are the respective partial functions $A[\![l]\!] : (\text{Addr} \hookrightarrow \text{Addr}) \hookrightarrow \text{Addr}$, and $V[\![e]\!] : (\text{Addr} \hookrightarrow \text{Addr}) \hookrightarrow \text{Addr}$.

$$
\begin{aligned}
A[\![x]\!]_\rho &\triangleq x && x \in \text{Var} \\
A[\![l \to f]\!]_\rho &\triangleq (v, f) && v = V[\![l]\!]_\rho \in \mathbb{N} \\
A[\![\textbf{deref}(e)]\!]_\rho &\triangleq V[\![e]\!]_\rho
\end{aligned}
$$

$$
\begin{aligned}
V[\![\textbf{new}\, c]\!]_\rho &\triangleq v && \text{where } v \in \mathbb{N} \text{ is an new instance of } c \\
V[\![\textbf{addrof}(l)]\!]_\rho &\triangleq A[\![l]\!]_\rho \\
V[\![l]\!]_\rho &\triangleq \rho\,(A[\![l]\!]_\rho)
\end{aligned}
$$

9.6.4 Concrete Semantics

The helper function call performs parameter assignments and applies the semantics of the called method's body. The $\mathcal{H}$ function transforms the effect of the function to the effect of a function call by reverting the values of local variables back to what they were before the call.

$$\begin{aligned}\mathsf{call}(m, c, (e_i)_{0\le i\le n}) \triangleq\ & [\![b]\!] \circ [\![p_n := e_n]\!] \circ \ldots \circ [\![p_1 := e_1]\!] \circ [\![\text{this} := e_0]\!] \\ & \text{where } p_i \text{ are the formal parameters} \\ & \quad \text{and } b \text{ is the body of } m.\end{aligned}$$

$$\mathcal{H}\, f_\rho\, v \triangleq \begin{cases} \rho\, v, & \text{if } v \in \text{Var} \\ f_\rho\, v & \text{otherwise} \end{cases}$$

The semantics of a statement or a control flow block b is a partial function $[\![b]\!] : (\text{Addr} \hookrightarrow \text{Addr}) \hookrightarrow (\text{Addr} \hookrightarrow \text{Addr})$ that preforms, when it is defined, a state transformation. The reason why for some block b and state ρ the semantics $[\![b]\!]_\rho$ is undefined lies in the fact that some subexpression of b cannot be evaluated in the state ρ, or during the evaluation an assignment to the **null** address is attempted.

$$\begin{aligned}[\![l := e]\!]_\rho &\triangleq \rho[A[\![l]\!]_\rho \to V[\![e]\!]_\rho] \\ [\![\textbf{return } e]\!]_\rho &\triangleq \rho[\text{Ret} \to V[\![e]\!]_\rho] \\ [\![l := e_0 \to m(e_1, \ldots, e_n)]\!] &\triangleq [\![l := \text{Ret}]\!] \circ \mathcal{H}(\mathsf{call}(l, m, (e_i)_{0\le i\le n})) \\ [\![\{b_1 b_2 \cdots b_n\}]\!] &\triangleq [\![b_n]\!] \circ \ldots \circ [\![b_2]\!] \circ [\![b_1]\!] \\ [\![\textbf{if } e \textbf{ then } b_{\mathsf{true}} \textbf{ else } b_{\mathsf{false}}]\!]_\rho &\triangleq [\![b_{v\neq\textbf{null}}]\!]_\rho \quad \text{where } v = V[\![e]\!]_\rho \\ [\![\textbf{while } (e)\ b]\!] &\triangleq [\![\{\textbf{if } e \textbf{ then } \{b \textbf{ while } (e)\ b\} \textbf{ else } \{\}\}]\!]\end{aligned}$$

9.6.5 Access Modifiers

We define the semantics of the *private* modifier of a field f defined in class C as the restriction that an expression that is an alias for f and that is external from C must not contain a direct reference to *private* field in the form of $lval \to f$ where $lval$ has the same type as C:

Definition 2 (Private Field)**.** *If $A_c(e) = (class, field) \wedge attrib(class, field) = private$ where e is an expression from a method that is external from class then $lval \to field$ is not contained in e if $type(lval) = class$.*

For public fields no such restrictions apply, so they can be referenced directly from any code.

9.7 Correctness & Completeness

In this Section a structural induction based proof showing that the analysis invariant always holds for the abstract semantics defined in Section 9.5 is given.

The binary relation $\Delta \subseteq (\text{Addr} \hookrightarrow \text{Addr}) \times \mathbb{D}'$ defines the condition that must hold so that the given abstract domain state correctly over-approximates the concrete program state:

$$\forall x \in Addr : y = \rho(x) \Rightarrow \alpha_\rho(y) \subseteq \rho^\sharp(\alpha_\rho(x)) \tag{9.5}$$

where $\alpha : (\text{Addr} \hookrightarrow \text{Addr}) \to Addr \to \mathcal{P}(val^\sharp)$ abstracts a concrete address to a set of abstract values using a concrete program state:

$$\alpha_\rho\ addr \triangleq \begin{cases} \mathcal{A}_{\rho_1^\sharp}(a\_ext) & : addr \text{ is declared or allocated in external code} \\ \{\textbf{null}\} & : addr = \textbf{null} \\ \alpha_\rho^\sharp(addr) & : \text{else} \end{cases}$$

Here, $\alpha^\sharp : (\text{Addr} \hookrightarrow \text{Addr}) \to Addr \to \mathcal{P}(val^\sharp)$ is the abstraction from the plug-in analysis.

The following invariant must hold for all abstract program states $\rho^\sharp$ at all program points that are *inside external code*:

$$\forall \rho^\sharp\ external\ to\ Class\ \forall x \in Q_{\rho^\sharp}\ ext, x \neq \textbf{null} : Q_{\rho^\sharp}\ x = Q^*_{\rho^\sharp}\ ext \tag{9.6}$$

Given a sound plug-in points-to analysis, (9.5) holds for the abstract semantics of the plug-in analysis. So we have to show that both, the lifting performed by our analysis does not violate the invariant and that it is strong enough to over-approximate the effects of external code, which is not visible to the analysis. Specifically, this includes calls to external code and returns to external code.

First, we show that inside code external to *Class* all expressions and left-values are abstracted to an element of $\rho^\sharp\ ext$ if (9.5) holds.

Lemma 11. *Given (9.5) and a concrete program state ρ,*

$$\forall e \in exp\ where\ e\ is\ external\ to\ Class\ : \alpha_\rho(V_\rho(e)) \subseteq \rho^\sharp\ ext \tag{9.7}$$

and

$$\forall l \in lval\ where\ l\ is\ external\ to\ Class\ : \alpha_\rho(A_\rho(l)) \subseteq \rho^\sharp\ ext \tag{9.8}$$

hold.

Proof. Inside external code, *private* fields from *Class* can be accessed only via a pointer, a direct reference is not possible due to Definition 2. So if a *private* field from *Class* (or something reachable thereof) can be accessed from external code then there must exist a pointer that can be accessed from external code and that points to the *private* data from *Class*:

Given a concrete program state ρ and $\rho \Delta \rho^\sharp$,
$\forall e \in exp$ where e is external to $Class, V_\rho(e)$ is declared or allocated inside $Class : \exists x \in Addr, \alpha_\rho(x) = \mathcal{A}_{\rho_1^\sharp}(a\_ext), y = \rho(x), \alpha_\rho(y) \not\subseteq \alpha_\rho(x), V_\rho(e) \in \rho^*\ y$.
$\alpha_\rho(y) \overset{(9.5)}{\subseteq} Q_{\rho^\sharp}(\alpha_\rho(x)) \overset{\alpha}{\subseteq} Q^*_{\rho^\sharp}\ ext \overset{(9.6)}{\subseteq} Q_{\rho^\sharp}\ ext$. (9.8) analog to (9.7). □

Applying *modify* to an abstract state $\rho^\sharp$ ensures that (9.6) holds:

Lemma 12. $\forall \rho^{\sharp\prime} = modify\ \rho^\sharp : \forall x \in Q_{\rho^\sharp}\ ext, x \neq \textbf{null} : Q_{\rho^\sharp}\ x = Q^*_{\rho^\sharp}\ ext$

Proof. *modify* joins the transitive closure $Q^*_{\rho^\sharp}\ ext$ with the content of all addresses of the transitive closure $\forall x \in Q^*_{\rho^\sharp}\ ext$. As $Q_{\rho^{\sharp\prime}}\ ext \subseteq Q^*_{\rho^{\sharp\prime}}\ ext$ (9.6) holds for $\rho^{\sharp\prime}$. □

9.7.1 Start of Induction

In order to show that $\rho\Delta\rho^\sharp$ holds when applying our analysis we prove (9.5) by structural induction over the analysis semantics. The start state for the induction is the state of the domain before any non-external code was called ($\rho_{init}\Delta\rho_i^\sharp$), but after any external code that might have occurred before non-external code is called. Initially,

$$Q_{\rho_i^\sharp}\ ext = \mathcal{A}_{\rho_1^\sharp}(a\_ext) \cup \{\textbf{null}\} \cup fields \cup this$$

, as the address of all *public* fields is assigned to a_ext before the actual analysis starts as shown in Eq. 9.1. Afterwards, *modify* is applied and (9.6) holds due to Theorem 12.

All *private* fields and variables from *Class* are initialized to **null** (in both the concrete and abstract semantics) and have not been modified due to Definition 2 since no non-external code was invoked, so we must distinguish two cases:

$x \in Addr$, **x is a *private* field or variable from *Class*:**
$\rho_{init}(x) = \textbf{null} \in Q_{\rho_i^\sharp}\ \alpha_{\rho_{init}}(x)$ due to the initialization

$x \in Addr$, **x is not a *private* field nor a variable from *Class*:**
$\exists l \in lval$ where l is external to *Class* :
$A_{\rho_{init}}(l) = x \overset{(9.8)}{\Rightarrow} \alpha_{\rho_{init}}(A_{\rho_{init}}(l)) \subseteq Q_{\rho_i^\sharp}\ ext$

, hence (9.5) holds.

As first part of the final proof, we show that external code is correctly over-approximated, as long as it does not invoke a method from *Class*. Here $\rho\Delta\rho^\sharp$ denotes the concrete and abstract state at any program point that is external to *Class*.

Lemma 13. *$\forall s \in stmt$ where s is external to Class : $\rho\Delta\rho^\sharp \Rightarrow (\llbracket s \rrbracket\rho)\Delta\rho^\sharp$*

Proof. Induction over the concrete semantics, induction start as in 9.7.1. Intuitively, as all values available to external code are in $Q_{\rho^\sharp}\ ext$, $Q_{\rho^\sharp}\ ext$ cannot grow without leaving external code by calling a (*public*) method. Only statements that may modify the concrete program state are investigated.

$s \equiv l := e$:
Due to (Theorem 11), (9.7) and (9.8) hold. Due to (9.8), $\alpha_\rho(A_\rho(l)) \subseteq Q_{\rho^\sharp}\ ext$. Due to (9.6) all possible addresses of l contain $Q_{\rho^\sharp}\ ext$: $\forall x \in \alpha_\rho(A_\rho(l)), x \neq \textbf{null} : q_{\rho^\sharp}\ x = Q_{\rho^\sharp}\ ext$
Since $\alpha_\rho(V_\rho(e)) \subseteq Q_{\rho^\sharp}\ ext$ due to (9.7), the concrete semantics $\rho[A_\rho(l) \rightarrow V_\rho(e)]$ is already over-approximated as only values from $Q_{\rho^\sharp}\ ext$ are assigned to the possible addresses of l which already contain these values.

$s \equiv$ **return** e:
The concrete semantics store the return value e of a method call in a helper variable *Ret* which is then assigned to the actual variable l that takes the return value. This is equivalent to storing the return value directly into the variable $\llbracket l := e \rrbracket$. This case has been proven above.

$s \equiv l := e_0 \rightarrow m(e_1, .., e_n)$: where $type(e_0) = Class$
The semantics of methods from *Class* are handled in the induction step in Section 9.7.2. □

The global domain state cannot shrink

Lemma 14. *$\forall x \in global, \forall s \in stmt, \forall \rho_0^\sharp \in \mathbb{D}, \forall g_0^\sharp \in \mathbb{G}' : (\rho_1^\sharp, g_1^\sharp) = \llbracket s \rrbracket^{\sharp'}_{(\rho_0^\sharp, g_0^\sharp)} \Rightarrow g_0^\sharp\ x \subseteq g_1^\sharp\ x$*

Proof. As *globals* joins any new state with the previous state, the global domain state cannot shrink. □

Next, we show that an assignment to an external l-value $l := e$ when e is non-external, is over-approximated by $[\![\mathbf{deref}(a\_ext) := e]\!]^\sharp$:

Lemma 15.
$\forall l \in lval$ *where* l *is external to Class*, $e \in exp$ *where* e *is non-external to Class* :
$[\![l := e]\!]^\sharp \rho^\sharp = [\![\textbf{\textit{deref}}(a\_ext) := e]\!]^\sharp \rho^\sharp$.

Proof. $\alpha_\rho(a\_ext) = \mathcal{A}_{\rho_1^\sharp}(a\_ext) = \alpha_\rho(l)$, so the set of abstract addresses of l is correctly approximated by the abstract addresses of a_ext. Writes to a_ext are performed via $\mathbf{deref}(a\_ext)$ in order to force the plug-in analysis to use a weak update on a_ext. Due to Eq.9.1 a_ext points at least to itself and *this* so it cannot be strong updated and $Q_{\rho^\sharp}(A_{\rho^\sharp}(\mathbf{deref}(a\_ext))) \supseteq Q_{\rho^\sharp}(A_{\rho^\sharp}(a\_ext))$. Therefore, $[\![\mathbf{deref}(a\_ext) := e]\!]^\sharp\ \rho^\sharp \sqsupseteq [\![a\_ext := e]\!]^\sharp\ \rho^\sharp$ and the content of a_ext cannot be destroyed by strong updatess of the plug-in analysis. □

Also, we show that an assignment to a non-external l-value $l := e$ when e is external, is over-approximated by $[\![e := a\_ext]\!]^\sharp$:

Lemma 16.
$\forall e \in exp$ *where* e *is external to Class*, $l \in exp$ *where* l *is non-external to Class* :
$[\![l := e]\!]^\sharp \rho^\sharp = [\![l := a\_ext]\!]^\sharp \rho^\sharp$

Proof. Analog to (15). □

9.7.2 Induction Step

Finally, we show that the over-approximation is sound also, when a method from *Class* is invoked. To do so, we perform the induction step over the abstract semantics of our analysis. The induction start was shown in Section 9.7.1.

$[\![e_0 \to m(e_1, \ldots, e_n)]\!]^{\sharp'}$**:**

Before executing the plug-in semantics we join the current globals into the plug-in domain, which does not break (9.5). As the plug-in semantics must be sound, it doesn't break (9.5). Due to Theorem 15, $[\![a\_ext := e]\!]^\sharp$ handles the assignment of the return value to an external variable correctly and (9.5) holds. Afterwards, *modify* is applied so that (9.6) holds for external code. Due to Theorem 13, the semantics of the external code that is executed after leaving the non-external method is correctly over-approximated. The globals are joined with their old values (see definition of *globals*), such that the global domain state cannot shrink (Theorem 14).

$[\![l := e_0 \to m^{extern}(e_1, .., e_n)]\!]^\sharp$**:**

Let $p_0, ..., p_n$ be the formal parameters of method m. Due to Theorem 15, the $[\![a\_ext := e_i]\!]^\sharp$ performed by the abstract semantics over-approximates the $[\![p_i := e_i]\!]$ performed by the concrete semantics and (9.5) holds. Afterwards, *modify* is applied so that (9.6) holds for external code. Due to Theorem 13, the semantics of the external method is correctly over-approximated. Again, due to Theorem 15, the assignment of the unknown, external return value is correctly handled by $[\![l := a\_ext]\!]^\sharp$.

$[\![other\ statements]\!]^\sharp$**:**

Soundness of plug-in semantics.

Table 9.1: Benchmark results.

Code	Classes	C++[loc]	C[loc]	Time[s]	Ext[%]	σ
Industrial	44	28566	1368112	282	23	24
Ogre	134	71886	1998910	42	62	36

9.8 Experimental Results

In this Section we present an implementation of the analysis and give a set of benchmark results that show how the analysis performs when it is applied to large sets of C++ code. As plug-in analysis a custom points-to analysis was implemented using the Goblint [140] framework.

The C++ code is transformed to semantically equivalent C using the LLVM [82] as the Goblint front-end is limited to C. Inheritance and access modifier information is also extracted and passed into the analysis. During this transformation, we verify that the access modifiers are not circumvented by casts or friends. No circumvention of access modifiers was found for the benchmarked code. Better analysis times can be expected from an analyzer that works directly with OO code as the LLVM introduces many temporary variables that have to be analyzed as well.

As additional input to the analysis a list of commonly used methods from the STL that were verified by hand was provided. Without this information more fields are flagged as escaping incorrectly by our implementation, because they are passed to an STL method which is considered external. This additional input is not required when using a different plug-in analysis that does not treat the STL methods as external.

The analysis is performed on two code-sets — the **Industrial** code is a collection of finite state machines that handle communication protocols in an embedded real-time setting whereas **Ogre** [109] is an open source 3d-engine. The results are given in Table 9.1.

The **C++** and **C** columns describe the size of the original code and its C code equivalent, respectively. More complex C++ code lines generate more C lines of code, so the ratio of code size is a measure for the complexity of the code that needs to be analyzed. The table shows that the industrial code is on average more complex than the Ogre code, requiring more time to analyze per line of code.

The time column represents the total time to analyze all the classes. The last two columns in the table show the mean and the standard deviation of the percentage of fields that the analysis identified as escaping. Fields that are identified as escaping limit the precision of subsequent analysis passes since they can be modified by external code. For the rest of the fields detailed information can be generated. A field f_i is escaping if and only if $\mathcal{A}_{\rho_f^\sharp}(a\_this{\rightarrow}f_i) \cap Q_{\rho_f^\sharp}\, ext \neq \emptyset$.

Only 23% of the fields are escaping for the industrial code. This is the case because most of the code processes the fields directly rather than passing the fields to methods outside the analyzed class, yielding good precision.

Inside the Ogre code most fields are classes themselves, so many operations on fields are not performed by code belonging to the class containing the field but by the class that corresponds to the field's type. Our plug-in analysis implementation handles the methods of these fields as unknown methods and assumes that the field escapes. By using a plug-in analysis that analyses into these methods from other classes (e.g. Whaley and

Rinard [143]) the precision of the analysis for the Ogre code can be improved to about 25% escaping fields, as indicated by preliminary results. Our analysis provides an initial context to analyze deeper into code outside of the class declaration. Especially for libraries it is necessary to generate an initial context if the library is analyzed in isolation.

So for the presented examples, for more than 75% of the fields detailed information can be extracted without analyzing the code that instantiates and uses the initial class. The benchmark times are obtained by analyzing all classes sequentially on a 2.8 Ghz Intel Core I7 with 8GB RAM. Since the results for each class are independent from the results for all other classes, all classes could be analyzed in parallel.

9.9 Conclusion

We have presented a sound class-modular, class-escape and points-to analysis based on the encapsulation mechanisms available in OO-languages. The analysis can be applied to a set of classes independently without analyzing the code that uses the class thus reducing the amount of code that needs to be analyzed compared to whole-program and summary-based analysis.

In addition, we have presented an easy to apply, yet powerful transformation of non-class-modular points-to analyses into class-modular, points-to and class-escape analyses. Since our framework has very weak requirements on potential plug-in points-to analyses, it can be applied to virtually all existing points-to analyses. We have shown, that the transformation will produce a sound analysis, given that the whole-program plug-in analysis was sound. Moreover, the resulting class-modular, class-escape and points-to analysis will inherit the properties of the plug-in and therefore benefits from previous and future work on points-to analyses.

The presented benchmarks show that the analysis can be applied to large, real world code yielding good precision. Due to the modularity of the analysis, flow sensitive pointer analysis becomes viable for compiler optimization passes. Class files can be analyzed and optimized independently before they are linked to form a complete program. Hence, various compiler optimizations and static verifiers can benefit from a fast class-modular class-escape and pointer analysis. Especially in large OO software projects that enforce common coding standards [132] the usage of non-private fields is rare, so good results can be expected.

Bibliography

[1] Adve, V. S., and Vernon, M. K. The influence of random delays on parallel execution times. *SIGMETRICS Perfom. Eval. Rev. 21*, 1 (1993).

[2] Adve, V. S., and Vernon, M. K. Parallel program performance prediction using deterministic task graph analysis. *ACM Transactions on Computer Systems 22*, 1 (2004), 94–136.

[3] Agha, G., and Thati, P. *An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 26–57.

[4] Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., and Torquati, M. An efficient unbounded lock-free queue for multi-core systems. In *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 662–673.

[5] Almasi, G. S., and Gottlieb, A. *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc., 1989.

[6] Andersen, L. Program analysis and specialization for the C programming language. Tech. rep., 94-19, University of Copenhagen, 1994.

[7] Arabnejad, H., and Barbosa, J. G. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Trans. Parallel Distrib. Syst. 25*, 3 (Mar 2014), 682–694.

[8] Armstrong, J., Virding, R., Wikström, C., and Williams, M. Concurrent programming in erlang.

[9] Baker, H. G. Use-once variables and linear objects: Storage management, reflection and multi-threading. *SIGPLAN Not. 30*, 1 (Jan 1995), 45–52.

[10] Barney, B., and Livermore, L. Posix threads programming. *Internet: https://computing.llnl.gov/tutorials/pthreads/>* (2009).

[11] Barvinok, A. I. A Polynomial Time Algorithm for Counting Integral Points in Polyhedra When the Dimension is Fixed. *Mathematics of Operations Research 19*, 4 (Nov. 1997), 769–779.

[12] Baskaran, M. M., Vydyanathan, N., Bondhugula, U. K. R., Ramanujam, J., Rountev, A., and Sadayappan, P. Compiler-assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors. In *Principles and Practice of Parallel Programming* (Feb. 2009), PPoPP, ACM, pp. 219–228.

[13] Bastoul, C. Code Generation in the Polyhedral Model Is Easier Than You Think. In *International Conference on Parallel Architecture and Compilation Techniques* (Juan-les-Pins, France, September 2004), IEEE, pp. 7–16.

[14] Bender, M. A., Farach-Colton, M., Pemmasani, G., Skiena, S., and Sumazin, P. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms 57*, 2 (2005), 75–94.

[15] Bertot, Y., and Castéran, P. *Interactive theorem proving and program development*. Springer Science & Business Media, 2004.

[16] Blanchet, B. Escape analysis for object-oriented languages: application to Java. *SIGPLAN Not. 34* (1999), 20–34.

[17] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (1999), 720–748.

[18] Boehm, H.-J., and Adve, S. V. Foundations of the c++ concurrency memory model. In *ACM SIGPLAN Notices* (2008), vol. 43, ACM, pp. 68–78.

[19] Boehm, H.-J., and Weiser, M. Garbage Collection in an Uncooperative Environment. *Software: Practice and Experience 18*, 9 (1988), 807–820.

[20] Bone, P. Automatic parallelisation for mercury.

[21] Boyapati, C., Liskov, B., and Shrira, L. Ownership types for object encapsulation. *SIGPLAN Not. 38* (2003), 213–223.

[22] Boyland, J. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience 31*, 6 (2001), 533–553.

[23] Brunner, J. Comparison of weighted reference counting and the boehm mark and sweep garbage collector for parallel programs on smp systems, 2011.

[24] Cascaval, C., DeRose, L., Padua, D. A., and Reed, D. A. *Compile-Time Based Performance Prediction*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 365–379.

[25] Chakravarty, M. M. T., Keller, G., Lechtchinsky, R., and Pfannenstiel, W. *Nepal – Nested Data Parallelism in Haskell*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan 2001, pp. 524–534.

[26] Chamberlain, B. L., Callahan, D., and Zima, H. P. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[27] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Object-oriented Programming, Systems, Languages, and Applications* (2005), ACM, pp. 519–538.

[28] Cheng, B.-C., and Hwu, W.-M. W. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (New York, NY, USA, 2000), PLDI '00, ACM, pp. 57–69.

[29] Clarke, D., and Wrigstad, T. *External Uniqueness Is Unique Enough*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 176–200.

[30] Clarke, E. M., Grumberg, O., and Peled, D. *Model checking*. MIT press, 1999.

[31] Clinger, W. D. Foundations of actor semantics.

[32] Coffman, Jr., E. G., Garey, M. R., and Johnson, D. S. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing 7*, 1 (1978), 1–17.

[33] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1977), ACM, pp. 238–252.

[34] Cousot, P., and Cousot, R. Modular Static Program Analysis. In *Compiler Construction*, R. Horspool, Ed., vol. 2304 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 263–283.

[35] Cruanes, T., Dageville, B., and Ghosh, B. Parallel sql execution in oracle 10g. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (2004), ACM, pp. 850–854.

[36] Darte, A., Robert, Y. P., and Vivien, F. *Scheduling and Automatic Parallelization*. Birkhäuser Boston, 2000.

[37] David, F. M., Carlyle, J. C., and Campbell, R. H. *Context Switch Overheads for Linux on ARM Platforms*. ExpCS '07. ACM, 2007.

[38] de Oliveira Castro, P., Louise, S., and Barthou, D. A multidimensional array slicing dsl for stream programming. In *CISIS* (2010), pp. 913–918.

[39] Debattista, K., Vella, K., and Cordina, J. Cache-affinity scheduling for fine grain multithreading. *Communicating Process Architectures 2002* (2002), 135–146.

[40] Dick, R. P., Rhodes, D. L., and Wolf, W. Tgff: Task graphs for free. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign* (1998), IEEE Computer Society, pp. 97–101.

[41] Dijkstra, E. W. Self-stabilization in spite of distributed control. In *Selected writings on computing: a personal perspective*. Springer, 1982, pp. 41–46.

[42] Dolby, J., and Chien, A. An automatic object inlining optimization and its evaluation. *SIGPLAN Not. 35*, 5 (2000), 345–357. ACM ID: 349344.

[43] Dwyer, M. B., and Clarke, L. A. *Data flow analysis for verifying properties of concurrent programs*, vol. 19. ACM, 1994.

[44] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS) 9*, 3 (1987), 319–349.

[45] Fog, A. Software optimization resources. *Internet: http://www.agner.org/optimize/* (2015).

[46] Garey, M. R., and Johnson, D. S. Computer and intractability. *A Guide to the Theory of NP-Completeness* (1979).

[47] Gerasoulis, A., and Yang, T. On the granularity and clustering of directed acyclic task graphs. *Parallel and Distributed Systems, IEEE Transactions on 4*, 6 (1993), 686–701.

[48] Girkar, M., and Polychronopoulos, C. D. Automatic extraction of functional parallelism from ordinary programs. *Parallel and Distributed Systems, IEEE Transactions on 3*, 2 (1992), 166–178.

[49] Girkar, M., and Polychronopoulos, C. D. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming 22*, 5 (1994), 519–551.

[50] Glass, R. L. Frequently forgotten fundamental facts about software engineering. *IEEE Software 18*, 3 (2001), 112,110–111.

[51] Graham, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics 17*, 2 (1969), 416–429.

[52] Grimm, R. *C++ 11: der Leitfaden für Programmierer zum neuen Standard.* Addison-Wesley, 2012.

[53] Grosser, T., Pop, S., Ramanujam, J., and Sadayappan, P. On recovering multi-dimensional arrays in polly.

[54] Group, K. O. W., et al. The opencl specification. *version 1*, 29 (2008), 8.

[55] Ha, S., and Lee, E. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers 46*, 7 (Jul 1997), 768–778.

[56] Hall, C. V. *Strictness Analysis Applied to Programs with Lazy List Constructors.* PhD thesis, Indiana University, 1987.

[57] Harper, R. Existential type: The point of laziness. *Internet: https://existentialtype.wordpress.com/2011/04/24/the-real-point-of-laziness/* (2015).

[58] Henretty, T., Veras, R., Franchetti, F., Pouchet, L.-N., Ramanujam, J., and Sadayappan, P. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing* (2013), ACM, pp. 13–24.

[59] Herz, A. funkyimp documentation. *Internet: http://www2.in.tum.de/funky/wiki* (2015).

[60] Herz, A., and Apinis, K. *Class-Modular, Class-Escape and Points-to Analysis for Object-Oriented Languages.* Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 106–119.

[61] Herz, A., and Pinkau, C. *Real-World Clustering for Task Graphs on Shared Memory Systems*. Lecture Notes in Computer Science. Springer International Publishing, May 2014, pp. 17–35.

[62] Hewitt, C. Actor model of computation: Scalable robust information systems.

[63] Horwitz, S. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst. 19*, 1 (Jan 1997), 1–6.

[64] Horwitz, S., and Shapiro, M. Modular Pointer Analysis. Tech. rep., 98-1378, University of Wisconsin–Madison, 1998.

[65] Hwu, W.-M., and Patt, Y. N. Checkpoint repair for high-performance out-of-order execution machines. *Computers, IEEE Transactions on 100*, 12 (1987), 1496–1514.

[66] Intel. Math kernel library v. 12.1.2 20111128. *Internet: http://software.intel.com/en-us/intel-mkl* (2013).

[67] Intel. Thread building blocks 4.1. *Internet: http://www.threadingbuildingblocks.org/* (2013).

[68] Intel. Desktop 4th generation intel core processor family, desktop intel pentium processor family, and desktop intel celeron processor family. *Internet: www.intel.com/content/dam/www/public/us/en/documents/specification-updates/4th-gen-core-family-desktop-specification-update.pdf* (2015).

[69] Intel. Intel® 64 and ia-32 architectures software developer manuals. *Internet: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html* (2015).

[70] Jeschke, E. R. *An Architecture for Parallel Symbolic Processing Based on Suspending Construction*. PhD thesis, Indiana University, 1995.

[71] Joerg, C. F. *The cilk system for parallel multithreaded computing*. PhD thesis, Citeseer, 1996.

[72] Jones, S. L. P. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[73] Jones, S. L. P., and Wadler, P. *Imperative functional programming*. ACM, 1993, pp. 71–84.

[74] Kane, G. *mips RISC Architecture*. Prentice-Hall, 1988.

[75] Kasahara, H., Honda, H., Mogi, A., Ogura, A., Fujiwara, K., and Narita, S. *A multi-grain parallelizing compilation scheme for OSCAR (optimally scheduled advanced multiprocessor)*, vol. 589 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1992, pp. 283–297.

[76] Keller, G., Chakravarty, M. M., Leshchinskiy, R., Peyton Jones, S., and Lippmeier, B. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *International Conference on Functional Programming* (2010), ACM, pp. 261–272.

[77] Khan, A. A., McCreary, C. L., and Gong, Y. A numerical comparative analysis of partitioning heuristics for scheduling tak graphs on multiprocessors. *Auburn University, Auburn, AL* (1993).

[78] Kogge, P. *Reading the tea-leaves: How architecture has evolved at the high end.* May 2014, pp. 515–515.

[79] Kong, M., Veras, R., Stock, K., Franchetti, F., Pouchet, L.-N., and Sadayappan, P. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013), pp. 127–138.

[80] Kwok, Y.-k., and Ahmad, I. *Benchmarking the Task Graph Scheduling Algorithms.* 1998, pp. 531–537.

[81] Lam, M. S., and Wilson, R. P. *Limits of Control Flow on Parallelism.* ISCA '92. ACM, 1992, pp. 46–57.

[82] Lattner, C., and Adve, V. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on* (2004), pp. 75 – 86.

[83] Launchbury, J. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1993), ACM, pp. 144–154.

[84] Leighton, F. T. *Introduction to parallel algorithms and architectures*, vol. 188. Morgan Kaufmann San Francisco, 1992.

[85] Leissa, R., Hack, S., and Wald, I. Extending a c-like language for portable simd programming. *ACM SIGPLAN Notices 47*, 8 (2012), 65–74.

[86] Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (Jul 2009), 107–115.

[87] Li, C., Ding, C., and Shen, K. *Quantifying the Cost of Context Switch.* ExpCS '07. ACM, 2007.

[88] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. Nvidia tesla: A unified graphics and computing architecture. *Ieee Micro 28*, 2 (2008), 39–55.

[89] Ling, Y., Mullen, T., and Lin, X. Analysis of optimal thread pool size. *SIGOPS Oper. Syst. Rev. 34*, 2 (Apr 2000), 42–55.

[90] Liou, J.-C., and Palis, M. A. An efficient task clustering heuristic for scheduling dags on multiprocessors. In *Workshop on Resource Management, Symposium on Parallel and Distributed Processing* (1996), Citeseer, pp. 152–156.

[91] Liou, J.-C., and Palis, M. A. *A Comparison of General Approaches to Multiprocessor Scheduling.* IEEE Computer Society, Washington, DC, USA, 1997, pp. 152–156.

[92] Lisper, B. Data Parallelism and Functional Programming. In *The Data Parallel Programming Model*, G.-R. Perrin and A. Darte, Eds., vol. 1132 of *LNCS*. Springer, Jan. 1996, pp. 220–251.

[93] Liu, Z. Worst-case analysis of scheduling heuristics of parallel systems. *Parallel Computing 24*, 5-6 (1998), 863–891.

[94] Logozzo, F. Automatic Inference of Class Invariants. In *Verification, Model Checking, and Abstract Interpretation* (2004), B. Steffen and G. Levi, Eds., vol. 2937, Springer Berlin / Heidelberg, pp. 211–222.

[95] Loidl, H.-W., Rubio, F., Scaife, N., Hammond, K., Horiguchi, S., Klusik, U., Loogen, R., Michaelson, G. J., Peña, R., Priebe, S., and et al. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput. 16*, 3 (Sep 2003), 203–251.

[96] Loogen, R., Ortega-Mallén, Y., and Peña-Marí, R. Parallel functional programming in eden. *Journal of Functional Programming 15*, 03 (2005), 431–475.

[97] Mankin, B. Jcpp - a java c preprocessor. *Internet: http://www.anarres.org/projects/jcpp/* (2015).

[98] Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. K., and Trinder, P. *Seq No More: Better Strategies for Parallel Haskell*. Haskell '10. ACM, 2010, pp. 91–102.

[99] McCreary, C., and Gill, H. Automatic determination of grain size for efficient parallel processing. *Commun. ACM 32*, 9 (1989), 1073–1078.

[100] McCreary, C. L., Khan, A., Thompson, J., and McArdle, M. A comparison of heuristics for scheduling dags on multiprocessors. In *Proceedings on the Eighth International Parallel Processing Symposium* (1994), IEEE Computer Society, pp. 446–451.

[101] McKenney, P. E. Is parallel programming hard, and, if so, what can you do about it? *Linux Technology Center, IBM Beaverton* (2011).

[102] Mellor-Crummey, J. M., and Scott, M. L. *Scalable Reader-writer Synchronization for Shared-memory Multiprocessors*. PPOPP '91. ACM, 1991, pp. 106–113.

[103] Mogul, J. C., and Borg, A. *The Effect of Context Switches on Cache Performance*. ASPLOS IV. ACM, 1991, pp. 75–84.

[104] Mycroft, A., and Voigt, J. *Notions of Aliasing and Ownership*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan 2013, pp. 59–83.

[105] Naveh, B., et al. Jgrapht. *Internet: http://jgrapht. sourceforge. net* (2008).

[106] Novillo, D. Openmp and automatic parallelization in gcc. In *GCC developers summit* (2006), Citeseer.

[107] Novillo, D. Parallel programming with gcc. *Red* (2006), 1.

[108] of Indiana, U. Open mpi v. 1.4.5. *Internet: http://www.open-mpi.org/* (2013).

[109] Open Source 3D Graphics Engine OGRE. *Internet: http://www.ogre3d.org/*.

[110] Oliveira, B. C., and Cook, W. R. *Functional Programming with Structured Graphs.* ICFP '12. ACM, 2012, pp. 77–88.

[111] OpenMP. Openmp. *Internet: http://openmp.org/wp/* (2013).

[112] Oracle. Java language specification. *Internet: http://docs.oracle.com/javase/specs/* (2015).

[113] Oracle. Openjdk 6 sources. *Internet: http://download.java.net/openjdk/jdk6/* (2015).

[114] Parr, T. J., and Quong, R. W. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience 25*, 7 (1995), 789–810.

[115] Plasmeijer, R., and Eekelen, M. V. *Functional Programming and Parallel Graph Rewriting*, 1st ed. 1993.

[116] Plasmeijer, R., and Eekelen, M. v. Keep it clean: A unique approach to functional programming. *SIGPLAN Not. 34*, 6 (Jun 1999), 23–31.

[117] Porat, S., Biberstein, M., Koved, L., and Mendelson, B. Automatic detection of immutable fields in Java. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research* (2000), CASCON '00, IBM Press, pp. 10–.

[118] Pöppl, A. Evaluation and prediction of execution times for opencl-based computations on gpgpu systems. Master's thesis, 2014.

[119] Quinton, P., Rajopadhye, S., and Wilde, D. On Deriving Data Parallel Code from a Functional Program. In *Parallel Processing Symposium* (Apr. 1995), IEEE, pp. 766–772.

[120] Rémy, D. Using, understanding, and unraveling the ocaml language from practice to theory and vice versa. In *Applied Semantics*. Springer, 2002, pp. 413–536.

[121] Rountev, A. Component-level dataflow analysis. In *Component-Based Software Engineering* (2005), Springer-Verlag Berlin, pp. 21–23.

[122] Rountev, A., and Ryder, B. Points-to and Side-Effect Analyses for Programs Built with Precompiled Libraries. In *Compiler Construction*, R. Wilhelm, Ed., vol. 2027 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001, pp. 20–36.

[123] Sarkar, V. *Determining Average Program Execution Times and Their Variance.* PLDI '89. ACM, 1989, pp. 298–312.

[124] Scholz, S.-B. Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting. *J. Funct. Program. 13*, 6 (Nov. 2003), 1005–1059.

[125] Seidl, H., Vene, V., and Müller-Olm, M. Global invariants for analyzing multithreaded applications. *Proc. of the Estonian Academy of Sciences: Phys., Math. 52*, 4 (2003), 413–436.

[126] Seidl, H., and Vojdani, V. Region Analysis for Race Detection. In *Static Analysis*, J. Palsberg and Z. Su, Eds., vol. 5673 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 171–187.

[127] Shin, D., and Kim, J. *Power-aware Scheduling of Conditional Task Graphs in Real-time Multiprocessor Systems*. ACM, New York, NY, USA, 2003, pp. 408–413.

[128] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1996), POPL '96, ACM, pp. 32–41.

[129] Stork, S., Marques, P., and Aldrich, J. *Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs*. OOPSLA '09. ACM, 2009, p. 933–940.

[130] Streit, K., Hammacher, C., Zeller, A., and Hack, S. Sambamba: A runtime system for online adaptive parallelization. In *Compiler Construction* (2012), Springer, pp. 240–243.

[131] Stroustrup, B. *The C++ programming language*, vol. 3. Addison-Wesley Reading, Massachusetts, 1997.

[132] Sutter, H., and Alexandrescu, A. *C++ coding standards: 101 rules, guidelines, and best practices*. Addison-Wesley Professional, 2005.

[133] Sutter, H., and Larus, J. Software and the concurrency revolution. *Queue 3*, 7 (Sep 2005), 54–62.

[134] Søndergaard, H., and Sestoft, P. Referential transparency, definiteness and unfoldability. *Acta Informatica 27*, 6 (May 1990), 505–517.

[135] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., , and Leiserson, C. E. The Pochoir Stencil Compiler. In *Parallelism in Algorithms and Architectures* (2011), ACM, pp. 117–128.

[136] Traub, K. R. *Implementation of Non-strict Functional Programming Languages*. MIT Press, 1991.

[137] Venet, A., and Brat, G. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In *Programming Language Design and Implementation* (Washington DC, USA, 2004), ACM, pp. 231–242.

[138] Verdoolaege, S. isl: An Integer Set Library for the Polyhedral Model. In *International Congress on Mathematical Software*, K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327 of *LNCS*. Springer, Kobe, Japan, Sept. 2010, pp. 299–302.

[139] Verdoolaege, S. Barvinok library v. 0.35. *Internet: http://barvinok.gforge.inria.fr/* (2013).

[140] Vojdani, V., and Vene, V. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp* (2009), vol. 30, pp. 141–155.

[141] Wadler, P. *Is There a Use for Linear Logic?* PEPM '91. ACM, 1991, pp. 255–273.

[142] Wadler, P. Comprehending monads. *Mathematical structures in computer science 2*, 04 (1992), 461–493.

[143] Whaley, J., and Rinard, M. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1999), OOPSLA '99, ACM, pp. 187–206.

[144] Wimmer, C., and Mössenböck, H. Automatic feedback-directed object inlining in the java hotspot(tm) virtual machine. In *Proceedings of the 3rd international conference on Virtual execution environments* (New York, NY, USA, 2007), VEE '07, ACM, pp. 12–21.

[145] Woo, M., Neider, J., Davis, T., and Shreiner, D. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[146] Xi, H., and Pfenning, F. Eliminating Array Bound Checking Through Dependent Types. In *Programming Language Design and Implementation* (1998), ACM, pp. 249–257.

[147] Yang, T., and Gerasoulis, A. Dsc: Scheduling parallel tasks on an unbounded number of processors. *Parallel and Distributed Systems, IEEE Transactions on 5*, 9 (1994), 951–967.

[148] Yu, Y., Rodeheffer, T., and Chen, W. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM SIGOPS Operating Systems Review* (2005), vol. 39, ACM, pp. 221–234.