UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

PhD thesis

Robert Schenck

Two Things I Did

Parallel Differentiation and Rank Polymorphism

Advisors: Fritz Henglein, Cosmin E. Oancea, Troels Henriksen.

Handed in: December 30, 2024.

This thesis has been submitted to the PhD School of The Faculty of Science, University of Copenhagen.

Abstract

In this thesis, I describe two things I did: (1) a compiler transformation that implements performant automatic differentiation (AD) for a functional, data-parallel language and (2) an approach for rank polymorphism in a statically typed language with parametric polymorphism and type inference.

On the AD side of things, a method for efficient reverse mode AD on nested parallel programs is presented. The approach uses a recomputation-based approach that eliminates storing program variables for the reverse sweep; instead, variables are recomputed as needed in each new scope. Under this technique, perfectly nested scopes do not introduce recomputation. This is exploited by applying a repertoire of compiler transformations to transform code into perfect nests. The language uses a lexicon of high-level parallel combinators—such as **`map`**, **`reduce`**, and **`scan`**—to build parallel-by-construction programs. Rewrite rules to differentiate each combinator are derived, yielding nested-parallel code which itself consists of parallel combinators. The resulting parallel code is aggressively optimized using a suite of general and AD-specific optimizations. An implementation in the Futhark programming language is reported on and evaluated against existing other modern AD implementations on a suite of benchmarks, demonstrating competitive performance.

On the rank polymorphism side of things, a mechanism for automatically lifting functions and replicating function arguments in a static context is presented. The aim is to capture the programming experience in dynamically typed array languages like NumPy and APL, which permit rank-polymorphic applications, while also preserving static typing guarantees. The type system—which supports parametric polymorphism, higher-order functions, and top-level let-generalization—determines the minimum number of lifting and replication operations by generating (and solving) integer linear programs from constraints generated at function application sites. Key theoretical properties of the mechanism are given. An implementation of the mechanism within the Futhark compiler is described and demonstrates the system's practicality.

Resumé

Foreliggende afhandling redegør for mine to bedrifter: (1) udførsel af en automatisk programbearbejdning, der kan udføre automatisk differentiering (AD) af programmer skrevet i et funktionsorienteret og dataparallelt programmeringssprog, samt (2) en tilgang til rangpolymorfi i et programmeringssprog med statiske typer, parametrisk polymorfi, og typeinferens.

For så vidt vedrører AD, da præsenterer jeg en effektiv fremgangsmåde for den såkaldte bagvendte tilgang. Denne fremgangsmåde undgår den sædvanlige lagring af de mellemresultater der skal bruges til returgennemløbet, og genberegner i stedet de nødvendige resultater for hvert indlejringstrin i programmet. Som et særtilfælde kræver perfekt indlejrede løkker ikke genberegning; en egenskab der udnyttes ved at foretage automatisk omskrivning af programtekst til slig perfekt indlejrede løkker. Det sprog, hvorpå jeg udfører automatisk differentiering, udtrykker deterministisk parallelisme via kombinatorer såsom `map`, `reduce`, og `scan`. Differentieringsreglerne for hver af disse kombinatorer gives, som ligeledes er udtrykt via tilsvarende parallelle kombinatorer. Den herfra dannede parallelle programtekst optimeres derefter gennem en række almene og specialiserede programoptimeringsteknikker. En konkret udførsel af fremgangsmåden er foretaget på oversætteren for programmeringssproget Futhark, og via en eksperimentel sammenligning af køretidsresultater med andre værktøjer til AD, demonstreres en konkurrencedygtig præstation.

For så vidt vedrører rangpolymorfi, da redegør jeg for en fremgangsmåde til automatisk tilpasning af funktioner og funktionsargumenter således at de kan anvendes på data af udvidet rang. Hensigten med denne fremgangsmåde er at imitere programmeringsoplevelsen i dynamisk typede geledsprog såsom NumPy og APL, der tillader rankpolymorfe funktionsanvendelser, men også samtidigt at bevare ønskværdige typesystemsgarantier. Typesystemet understøtter parametrisk polymorfi, højereordensfunktioner, samt let-generalisering, og er derudover i stand til at afgøre den minimale udvidelse af funktioner til argumenter af højere rang, ved at løse lineære heltalsproblemer dannet ud fra de typeligninger der opstår ved funktionsanvendelsespunkter. Centrale teoretiske egenskaber for systemet præsenteres. En implementering af fremgangsmåden, udført ved en udvidelse af oversætteren for programmeringssproget Futhark, er blevet udført, hvilket tjener til at tydeliggøre typesystemets praktiske anvendelighed.

Contents

Preface

Publications

During my studies, I worked on the following publications:

1. Martin Elsman, Fritz Henglein, Robin Kaarsgaard, Mikkel Kragh Mathiesen, and Robert Schenck. "Combinatory Adjoints and Differentiation". In: *Electronic Proceedings in Theoretical Computer Science* 360 (June 2022), pp. 1–26. ISSN: 2075-2180. DOI: 10.4204/eptcs.360.1. URL: http://dx.doi.org/10.4204/EPTCS.360.1

2. Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E. Oancea. "AD for an Array Language with Nested Parallelism". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022. ISBN: 9784665454445

3. Robert Schenck, Nikolaj Hey Hinnerskov, Troels Henriksen, Magnus Madsen, and Martin Elsman. "AUTOMAP: Inferring Rank-Polymorphic Function Applications with Integer Linear Programming". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA2 (2024). DOI: 10.1145/3689774. URL: https://doi.org/10.1145/3689774

This thesis is based off of publications 2 and 3.

Acknowledgments

A lot of people have been involved with and enabled the work in this thesis—often directly as collaborators and colleagues, but also in other supportive capacities. First and foremost, warm thanks to my advisors: Fritz Henglein, Cosmin E. Oancea, and Troels Henriksen. Fritz, for his deep and unique insights, active encouragement to explore my interests, and support throughout my PhD; Cosmin, for being the king of no bullshit, as well as a great collaborator and friend; and Troels, for his infectious capacity for (principled) irreverence and his sagely teachings (the most important of which is that IRC is indeed the best channel for research). I'd also like to thank Thomas Hamelryck for his support and rousing of many stimulating discussions.

I've had the pleasure (hopefully mutual) of sharing an office with many colleagues during my PhD. I shared my first office with Lys Sanz Moreta and Ola Rønning—both of whom added (much needed) levity to my early PhD days (albeit in very different ways). Mikkel Kragh Mathiesen soon joined (or, rather, we joined him) and was always willing to discuss ideas (and laments!) and impress us with his insights. After some moving around and graduations, I spent my last year in the company of Zhan Su,

Nikolaj Hey Hinnerskov, and Frans Zdyb, all of whom were great company. Special thanks go to Nikolaj for many fruitful discussions as a collaborator and for putting up with all my mischief and idiosyncrasies. Another thanks goes to Duarte David, for tolerating my messages of "*AAAAAAAAAAAAAAAAAAAAAAAA*" sent in despair as we both chipped away at our respective degrees. Many other colleagues at DIKU have supported me and made my PhD a rewarding and enjoyable (well, sometimes) experience—thank you!

I'm also grateful to my dad, brother, and sister for all their support, the dynamic of which remains unchanged from this photo:

My dad, sister, me, and my brother in the 90s.

My PhD work has been supported by the Independent Research Fund Denmark (DFF) under the grant *Deep Probabilistic Programming for Protein Structure Prediction*. Thank you!

Finally, thanks to Troels Henriksen and Niels G. W. Serup for diligently translating the English abstract of this thesis into Danish.

Chapter 1

Introduction

This thesis describes two things I did during my PhD studies—namely, the development of techniques for parallel automatic differentiation and a pragmatic approach for rank polymorphism in static type systems with parametric polymorphism. To the initiated, these two projects might seem rather disparate—they are! Although it wouldn't be remiss to classify them under the umbrella of scientific programming language research. The languages and tools used in scientific computing often have an ad-hoc flair to them. This thesis demonstrates that a more disciplined and rigorous approach is not only possible, but desirable, conferring benefits like static guarantees, greater efficiency (in terms of the programmer, the compiler engineer, and the computation) and predictability.

The Futhark programming language[1] [35, 38] (and its compiler) serves as the primary context for the work in this thesis; both of the projects described were realized as extensions to Futhark. Futhark is a statically typed, parallel, functional array language designed to support arbitrary nested parallelism. It is a pure language with a type system in the Hindley-Milner tradition that supports parametric polymorphism and top-level let-generalization. A lexicon of higher-order parallel combinators—including **`map`**, **`reduce`**, and **`scan`**—form the vocabulary by which programs in Futhark are built and enable a high-level, declarative approach to writing parallel-by-construction programs. Optimizing transformations over these combinators are what enable the Futhark compiler to generate high-performance code. Chapter 2 provides a more detailed introduction to Futhark.

The remainder of this chapter provides a brief, high-level introduction to automatic differentiation and rank polymorphism and the research that I did in these areas during my PhD studies. The image above is of a soundly sleeping hedgehog, dreaming of GOING FAST!! (Hedgehogs are the mascot of the Futhark programming language.)

1.1 Automatic Differentiation

Automatic differentiation (AD) (sometimes also called algorithmic differentiation) is a technique for efficiently computing the derivatives of computer programs. AD exploits the fact that all differentiable computer programs are ultimately compositions of simple functions. Simple functions are easy to programmatically differentiate—just hard code in the differentiation rules (e.g., the differentiation rule for $\sin(x)$ returns $\cos(x)$). The chain rule is the first tool any calculus practitioner pulls from their tool bag when

[1] `https://futhark-lang.org`

differentiating function compositions—it's the tool for the job in AD as well. In a real sense, AD is simply the application of the chain rule and basic differentiation rules. An important distinction from symbolic methods is that AD is only concerned with the numerical evaluation of a derivative and not a symbolic representation of the derivative. (Although I'd argue that a declarative program that computes a derivative at point x is itself a symbolic representation of the derivative.)

Since the derivative of any expression depends only on its free variables and their derivatives, implementing AD involves interlacing a program's expressions with corresponding derivative computations. For example, you can augment the following program:

$$
\begin{aligned}
&\textbf{def}\ P\ x_0\ x_1 = \\
&\quad \textbf{let}\ v = x_0 \cdot x_1 \\
&\quad \textbf{let}\ y = \sin(v) \\
&\quad \textbf{in}\ y,
\end{aligned}
$$

to also compute its derivative as follows:

$$
\begin{aligned}
&\textbf{def}\ P_{\text{fwd}}\ x_0\ x_1\ \dot{x}_0\ \dot{x}_1 = \\
&\quad \textbf{let}\ v = x_0 \cdot x_1 \\
&\quad \textbf{let}\ \dot{v} = x_1 \cdot \dot{x}_0 + x_0 \cdot \dot{x}_1 \\
&\quad \textbf{let}\ y = \sin(v) \\
&\quad \textbf{let}\ \dot{y} = \cos v \cdot \dot{v} \\
&\quad \textbf{in}\ (y, \dot{y}),
\end{aligned}
$$

where $\dot{y}$ is the derivative of variable y with respect to the inputs x_0 and x_1 (and analogously for $\dot{v}$). The augmented inputs $\dot{x}_0$ and $\dot{x}_1$ specify the direction to take the derivative in (setting, for example, $\dot{x}_0 = 1$ and $\dot{x}_1 = 0$ would yield the derivative with respect to $\dot{x}_0$). This basic approach, called *forward mode AD*, is an efficient/effective way to differentiate programs [30]: don't let the simplicity of the approach mislead you!

Where forward mode AD comes up short, though, is differentiating programs that have (far) more inputs than outputs. Finding the complete gradient (or Jacobian for a program with more outputs), requires executing P_{fwd} once for each input of P. This means that differentiating, say, a loss function $l : \mathbb{R}^{10^6} \mapsto \mathbb{R}$ would require a million invocations of P_{fwd}.

Fortunately, there's a better way—as the name forward mode might suggest, there is a corresponding *reverse mode* which computes the derivative of a program in reverse program order. With reverse mode AD, the complete gradient (or Jacobian) requires executing the reverse mode program P_{rev} once for each output. This means that differentiating l would only require a single execution of P_{rev}—assuming P_{rev} isn't a million times slower than P_{fwd}[2]: that's a lot better!

However, the "reverse" in reverse mode also makes it much more challenging because the derivative computations have reversed dependencies from the original program. This mandates either executing the original program to first save all variables on a so-called "tape" (since the derivative computations depend on these variables) or recalculating the variables during the derivative computation (or using a strategic mix of the two). Designing effective data structures and access patterns to efficiently

[2]It isn't and, optimally, P_{rev}'s runtime is bound by a small constant factor times the runtime of P. [30]

read and write from the tape and also deciding when to store vs. recompute values is challenging [57, 6, 64, 30, 89]. This is made only more difficult in parallel computing contexts, which usually target accelerators with constrained execution models, like graphics processing units (GPUs).

In Chapter 3, I introduce a technique for efficient reverse mode AD within a nested-parallel framework that's based on a calculus of parallel combinators. Since handling a tape efficiently in a nested-parallel context is challenging, this approach utilizes recomputation in lieu of a tape. The transformation is articulated through parallel-combinator-specific rewrite rules. These rules are defined at a high-level—a benefit inherent to the combinator calculus—which facilitates straightforward arguments for their efficiency and correctness.

I also present an implementation of this technique in the Futhark compiler. The implementation is evaluated on a comprehensive suite of benchmarks, demonstrating competitive performance when compared against other state-of-the-art solutions.

1.2 Rank Polymorphism

In mathematics, notation is often overloaded, abused, or under-specified (and everything in between). This isn't necessarily problematic as the context in which the notation appears typically provides enough information to disambiguate it. A common example is using the same operator for arguments of different ranks. For instance, the + operator is used for standard scalar addition:

$$1 + 2,$$

but also for matrix addition

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

This flexibility (at the cost of potential ambiguity) is also enjoyed in some programming languages. These languages support mathematical syntax that is overloaded or implicitly coerced to suit the application and can bridge the gap between mathematical concepts and their computational implementation by allowing similar notational conventions. This alignment makes programs easier to write, read, and understand.

There are two approaches to enable a similar notational flavor as the example above. The first is *ad-hoc polymorphism* (also called *overloading*), where distinct (but related) functions are grouped under the same name or symbol. In this scheme, scalar addition and matrix addition are defined separately but are both associated with the + symbol. The compiler uses static information at the application site to determine which function to actually use.

Another approach, dubbed *rank polymorphism*, elides multiple definitions of a function entirely. Here, there is only a single definition of the + operator, typically scalar addition.[3] When + is applied on non-scalar arguments, it isn't an error. Instead, the compiler (or runtime system) appropriately coerces the function and its argument to produce a rank-correct application. For example, applying + to two matrices would require lifting the + function to operate on matrices via a **`map`**-like operator; in rank

[3] Another view is that actually + is itself rank polymorphic. In this view, a reasonable type signature is

$$+ \; : \; \forall S.\, \forall T.\, S\texttt{int} \to T\texttt{int} \to \max(S,T)\texttt{int}$$

where S and T are *shape variables*. The compiler figures out the specific S and T for each application and generates the appropriate code accordingly (but how to do so remains an unsolved problem [90]).

polymorphic languages, this lifting happens automatically and transparently to the programmer.

While ad-hoc polymorphism, or some form of overloading, is prevalent in modern programming languages, rank polymorphism isn't as common. This is particularly true in statically typed languages that support parametric polymorphism due to the inherent challenges involved in statically inferring how to coerce functions when type variables are involved. Type variables, by definition, can represent any type—how can a compiler determine how to coerce a function when its only information about an argument is that it has some generic type α? This gets at a core challenge in any language feature that introduces ambiguity: how should the compiler choose? Despite this, rank polymorphism (in a statically typed context) has a unique value proposition: it requires simpler machinery compared to ad-hoc polymorphism (which has a whole class of inference and ambiguity challenges of its own [77]) and doesn't require defining separate functions for each possible argument type. It's also a natural fit for parallel programming models since it can automatically extend operations to work over higher-dimensional structures and can abstract away the noise of iteration (and replication) entirely.

In Chapter 4, I develop a method to augment a statically typed, polymorphic language with rank polymorphic function applications. The key idea is to relax the type constraints generated at function application sites during type checking into so-called "rank constraints": the set of rank constraints of a program characterize the legal set of program coercions—namely function lifting and argument replication—that can be applied to make the program "rank-correct". A core difficulty is that any solver for the rank constraint set must consider all constraints simultaneously. Rank constraints cannot in general be solved in isolation: how one application is coerced may change how other applications are coerced. Another challenge is that the constraint set may have an infinite number of solutions, which is a lot of ambiguity to overcome! Fortunately, there's an elegant solution to both problems: the rank constraints can be used to build an integer linear program whose solution corresponds with the minimum number of coercions required for a "rank-correct" program. In practice, we found that coercions generated by this modality were unsurprising and corresponded well with programmer intent. Chapter 4 also demonstrates a number of important consistency properties of the system and also reports on an implementation of the method in the Futhark compiler.

Chapter 2

Background

The work described in this thesis might be best categorized as "applied theory"—practical work with robust and general theoretical underpinnings. The practical work takes the shape of extensions and modifications to the Futhark programming language, so a brief introduction is in order.

The image above is a (curious) interpretation of a particularly fast hedgehog, imagined and drawn by Nikolaj Hey Hinnerskov.

2.1 Futhark

Futhark is a statically-typed, high-level, parallel, functional array language designed for high-performance computing [35] (hence speedy hedgehog mascot—Gotta Go Fast!). In Futhark, programs are built using a calculus of parallel *second-order array combinators* (SOACs). These include parallel versions of common functional language constructs like **`map`**, **`reduce`**, and **`scan`** as well as more specialized combinators, such as **`hist`** [37], which computes (in parallel) generalized histograms. Programs in Futhark are parallel by construction (as long as they use SOACs), but maintain entirely sequential semantics that are guaranteed race-free. This results in a programming experience similar to any other ML-like language—parallel execution is abstracted away from the programmer as an implementation detail that's handled by the compiler.

To get a taste of the language, the following code implements matrix multiplication in Futhark (where xss and yss are the matrices to be multiplied):

$$
\begin{aligned}
&\textbf{map}\ (\lambda xs \rightarrow \\
&\quad \textbf{map}\ (\lambda ys \rightarrow \\
&\qquad \textbf{reduce}\ (+)\ 0\ xs\ (\textbf{map}\ (\cdot)\ xs\ ys) \\
&\quad)\ (\textbf{transpose}\ yss))\ xss.
\end{aligned}
$$

The above code is highly parallel: the two outer maps express parallel loops and the inner reduce expresses a parallel reduction. While Futhark is hardware-agnostic, parallel hardware typically offers only a few levels of parallelism and a finite number of threads for computation; programs often exhibit both more parallelism and deeper nested parallelism than the targeted hardware can efficiently support. To address this, Futhark employs a combination of optimization techniques. The first is *flattening* [9, 39], which transforms nested parallelism into flat parallelism. Flattening all available parallelism is inefficient in practice as it can incur significant overhead and negatively impact data locality. Instead, Futhark applies flattening judiciously, often preserving

some hierarchical parallelism. The second is *fusion*, in which nested SOACs are fused to prevent the allocation of intermediate data structures and improve locality. For example, **map** $g \circ$ **map** f can be fused into **map** $(g \circ f)$ (which, operationally, flattens the expression into a single parallel loop). The third is *sequentialization*, where excess parallelism that cannot be efficiently mapped to hardware is identified and sequentialized. Perhaps counterintuitively, Futhark can be understood as a sequentializing compiler: Futhark programs often contain excess parallelism—its success in efficiently compiling programs for parallel hardware is closely tied to its careful selection of what to execute in parallel and what to sequentialize.

The data-parallel Futhark programming model—built around SOACs—enables an easy-to-reason-about programming style at a high level of abstraction. This richness percolates into Futhark's internal representation, allowing for a compiler writing discipline in which transformation and optimizations are expressed with a high level of abstraction.

Futhark belongs to a small niche of strongly- and statically-typed high-performance languages. It features top-level parametric polymorphism, type inference, and even a light form of dependent types for tracking sizes [5]. While these features add complexity compared to languages with more permissive typing disciplines, they also enable a high-level programming experience in Futhark with more sophisticated static checks. Additionally, features commonly associated with weaker type systems aren't necessarily off-limits: for example, via its *uniqueness types* [39], Futhark's type system also enables in-place updates by tracking array reads/writes at the type level. Rank polymorphism (see Chapter 4) is another example and is illustrative of Futhark's type system design ethos: it should be pragmatically useful to the programmer.

2.2 Language

In this section, we introduce a simple data-parallel language that models the core of the full Futhark programming language. This language will be used throughout the rest of the thesis. As with Futhark, the language is an ML-like language based on the lambda calculus and expresses all parallelism via SOACs. It's strictly typed and features parametric polymorphism, higher-order functions, and type inference.

2.2.1 Basics

All functions take a single argument; functions with multiple arguments are curried, resulting in a sequence of multiple applications. Function application is denoted by spaces and is left-associative. For example, the imperative $f(x, y, z)$ is written as $f\ x\ y\ z$ and involves three separate function applications; it's equivalent to $((f\ x)\ y)\ z$.

Top-level functions are declared via **def** and may be polymorphic. For example,

$$\textbf{def}\ id\ (x : \alpha)\ :\ \alpha\ =\ x,$$

defines the polymorphic identity function.[1] Functions (and constants) inside top-level bodies are declared using **let** $x = e$ **in** b and are monomorphic.[2] **let**-expressions

[1] Similarly to Haskell's type signatures, type annotations for the arguments and return type aren't required and are placed by convention/as documentation.

[2] Forgoing generalization of let-bindings in top-level bodies turns out to not affect expressivity and considerably simplifies type inference. [100]

consist of a series of bindings—which we also call statements—followed by a sequence of one or more returns which follow the **in** keyword. For example,

$$\begin{array}{l}\textbf{let}\ a = 5\\ \textbf{let}\ b = a\ \cdot\ a\\ \textbf{in}\ b,\end{array}$$

evaluates to 25 (and is equivalent to **let** $a = 5$ **in** (**let** $b = a \cdot a$ **in** b)).

The language uses a list representation for arrays. Multidimensional arrays are represented as lists-of-lists. For example, the matrix

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix},$$

may be represented as the list $[[1, 2], [3, 4]]$.

The language does not support recursion. However, it features a sequential **loop**-construct that models the same semantics of a tail-recursive function:

$$\textbf{loop}\ x = x_0\ \textbf{for}\ i < n\ \textbf{do}\ e.$$

Semantically, the loop is initialized by binding x_0 to x and i to 0. Each iteration of the loop increments i and executes e, binding the result of the expression to x, which is used on the subsequent iteration. The loop terminates after n iterations and returns the final evaluation of e as its result. The language also supports while-loops:

$$\textbf{loop}\ x = x_0\ \textbf{while}\ c\ \textbf{do}\ e,$$

which work as you'd expect—initially setting $x = x_0$ and then evaluating e and binding x to the result until the condition c evaluates to **false**.

The language supports a functional flavor of in-place updates based on uniqueness types [38]. Values are written in-place using the **with** syntax:

$$\textbf{let}\ xs' = xs\ \textbf{with}\ [i]\ \leftarrow\ x.$$

The above has the semantics that xs' is a copy of xs in which the element at index i is updated to x, but also provides the operational guarantee that the update will be realized in place. The statement **let** $xs'[i] = x$ is syntactic sugar for the above; sometimes it will be written **let** $xs[i] = x$ (i.e., using the same name xs for the updated array); in this case xs should be understood to be a new variable that shadows the old xs. In a similar vein, statements of the form **let** $x\ +\!=\ a$ and **let** $xs[i]\ +\!=\ a$ are also allowed and are sugar for **let** $x\ =\ x + a$ and **let** $xs\ =\ xs$ **with** $[i]\ \leftarrow\ xs[i] + a$, respectively. Note that in the first statement x on the left-hand side shadows the x on the right-hand side (and analogously for xs in the second statement).

2.2.2 SOACs

In this section, we describe each SOAC in the language.

2.2.2 (a) `map`

A **map** applies a function to each element of an array, producing an array of the same length. Its signature is

$$\textbf{map}\ :\ (f\ :\ \alpha \to \beta) \to (as\ :\ []\alpha) \to []\beta,$$

and is semantically defined as

$$\textbf{map}\ f\ [a_0,\ a_1,\ \ldots,\ a_{n-1}]\ \equiv\ [f\ a_0,\ f\ a_1,\ \ldots,\ f\ a_{n-1}].$$

which is equivalent to the imperative parallel loop

$$\begin{array}{l}\textbf{forall}\ i = 0 \ldots n-1\\ \quad as[i] = f(as[i])\end{array}\quad ,$$

where $as = [a_0,\ a_1,\ \ldots,\ a_{n-1}]$.

For simplicity, we allow SOACs to be called with k-ary functions wherein the SOAC is applied to k equal-length arrays. For example,

$$\textbf{map}\ g\ as\ bs\ \equiv\ [g\ a_0\ b_0,\ g\ a_1\ b_1,\ \ldots,\ g\ a_{n-1}\ b_{n-1}],$$

where $as = [a_0,\ a_1,\ \ldots,\ a_{n-1}]$ and $bs = [b_0,\ b_1,\ \ldots,\ b_{n-1}]$.

2.2.2 (b) `reduce`

The **`reduce`** combinator combines all elements of an array with a binary associative operator $\odot$. Its type signature is

$$\textbf{reduce}\ :\ (\odot\ :\ \alpha \to \alpha \to \alpha) \to (e_\odot\ :\ \alpha) \to (as\ :\ [\,]\alpha) \to \alpha,$$

and is semantically defined as

$$\textbf{reduce}\ \odot\ e_\odot\ [a_0,\ a_1,\ \ldots,\ a_{n-1}]\ \equiv\ e_\odot \odot a_0 \odot a_1 \odot \cdots \odot a_{n-1}.$$

The $e_\odot$ argument is called the *neutral element* and is—as the name suggests—the identity of the $\odot$ operator. That is, for all $a\ :\ \alpha$, $a \odot e_\odot = e_\odot \odot a = a$.

2.2.2 (c) `scan`

The **`scan`** combinator returns a **`reduce`** (with respect to a given binary associative operator $\odot$) of every non-empty prefix of a list. Its type signature is

$$\textbf{scan}\ :\ (\odot\ :\ \alpha \to \alpha \to \alpha) \to (e_\odot\ :\ \alpha) \to (as\ :\ [\,]\alpha) \to [\,]\alpha,$$

and is semantically defined as

$$\textbf{scan}\ \odot\ e_\odot\ [a_0,\ a_1,\ \ldots,\ a_{n-1}]\ \equiv\ [e_\odot \odot a_0,\ e_\odot \odot a_0 \odot a_1,\ \ldots,\ e_\odot \odot a_0 \odot a_1 \odot \cdots \odot a_{n-1}],$$

where just like for **`reduce`**, $e_\odot$ is the neutral element of $\odot$.

2.2.2 (d) `hist`

The **`hist`** combinator is a generalized histogram computation [37]. Its type signature is

$$\textbf{hist}\ :\ (\odot :\ \alpha \to \alpha \to \alpha) \to (e_\odot :\ \alpha) \to (is :\ [n]\texttt{int}) \to (as :\ [n]\alpha) \to [m]\alpha,$$

where is is an array of indices (bins) that specifies where each value of as should be placed; is and as must be the same length and together constitute an index-value pairing. The $\odot$ operator must be associative and commutative; $e_\odot$ is its identity element. The number of bins is typically (much) smaller than the number of index-value pairs. Bins are initialized with $e_\odot$. The expression **`hist`** $\odot\ e_\odot\ is\ as$ has the same semantics as

$$\begin{array}{l}\textbf{loop}\ xs = \textbf{replicate}\ m\ e_\odot\ \textbf{for}\ i\ =\ 0 \ldots n-1\ \textbf{do}\\ \quad xs\ \textbf{with}\ [is[i]]\ =\ xs[is[i]] \odot as[i]\end{array}\quad ,$$

where m is the number of bins and n is the length of is and as.

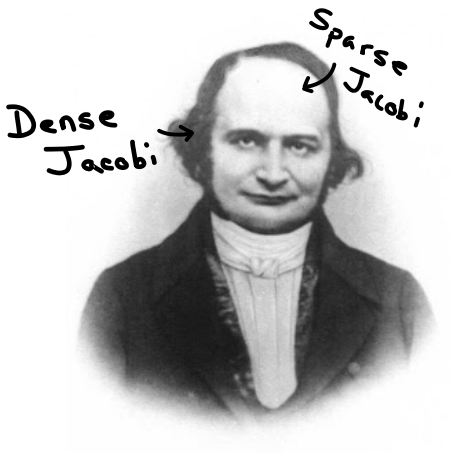
2.2.2 (e) `scatter`

The **`scatter`** combinator updates an array in-place according to its arguments. Its type signature is

$$\textbf{scatter} : (xs\ :\ [\,]\alpha) \to (is\ :\ [\,]\texttt{int}) \to (vs\ :\ [\,]\alpha) \to [\,]\alpha,$$

and it produces an array by updating in-place the array xs at the m indices in the duplicate-free is array with corresponding values of vs. It is semantically equivalent to

$$\begin{array}{l}\textbf{loop}\ xs' = xs\ \textbf{for}\ i\ =\ 0 \ldots m-1\ \textbf{do}\\ \quad xs'\ \textbf{with}\ [is[i]]\ =\ vs[i]\end{array}\ .$$



Chapter 3

Parallel Automatic Differentiation

This chapter is an adaptation of the following publication:

> Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E. Oancea. "AD for an Array Language with Nested Parallelism". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022. ISBN: 9784665454445

The image above is of Carl Gustav Jacob Jacobi, the namesake of the Jacobian matrix.

3.1 Introduction

Automatic differentiation (AD) is a practical way to compute derivatives of functions that are expressed as programs. AD of sequential code is implemented in tools such as Tapenade [4], ADOL-C [29], and Stalingrad [74]. Modern deep learning is built on array programming frameworks such as Tensorflow [1], PyTorch [73], and JAX [12] which provide implicitly parallel bulk operations that support AD.

A largely unsolved challenge is supporting AD for high-level parallel languages [38, 71, 98] that permit arbitrary nesting of sequential and parallel constructs. Such solutions may in principle act as a catalyst for prototyping and training of more advanced machine learning (ML) models.

ML often involves minimizing a cost function, a procedure which generally involves computing its derivative. Cost functions in ML workloads typically have far more inputs than outputs; the *reverse mode* of AD is the most efficient in such cases [6] but is challenging to implement because intermediate program values are required by the differentiated code. The program must first run a *forward sweep* (also known as a primal trace) that stores intermediate program states on the *tape*. The tape is read from in the *return sweep*, which executes the program in reverse to compute the derivative.

A significant amount of work has studied how to elegantly model reverse mode AD as a compiler transformation and how to hide the tape under powerful programming abstractions such as (dynamic) closures [74] and delimited continuations [102]. These abstractions are not suited for efficient parallel execution on many core hardware such as GPUs.

This work is, to our knowledge, the first to demonstrate an efficient GPU implementation of reverse mode AD as a compiler transformation on a data-parallel language

that supports nested parallelism by means of higher-order array combinators such as **`map`**, **`reduce`**, **`scan`**, **`hist`**, and **`scatter`**.

Most reverse mode AD systems save all variables on the tape by default and support checkpointing annotations as a memory footprint optimization. However, in a nested-parallel context, the tape may give raise to complex, irregular data structures which are passed across deep nests and are challenging to implement efficiently in regard to optimizing spatial and temporal locality. Our approach exploits the fact that applying reverse mode AD to a straight line of side effect-free code does not require any tape because all intermediate values remain available [6]. We expand this idea to drive the code transformation across lexical scopes by requiring that whenever the return sweep enters a new scope s, it first redundantly re-executes the forward sweep of s in order to bring all the needed variables into scope.

Our technique preserves work-span asymptotics because the recomputation overhead is at worst proportional to the depth of the deepest nest of scopes, which is constant for a nonrecursive program. Moreover, perfectly nested scopes (other than loops) are guaranteed to not introduce re-execution, hence the overhead can be minimized by classic compiler transformations such as flattening nested parallelism [40] and polyhedral-like optimizations [10]. Since we forgo passing a tape across scopes, scalars are efficiently accessed from registers rather than global memory, and the code resulting from the AD transformation fully benefits from the existent compiler optimization repertoire.[1]

This recomputation technique is the glue that binds scopes together; the other components of our technique are high-level rewrite rules for differentiating parallel combinators (SOACs). These rules are derived using the main rewrite rule of the reverse mode transformation (see Figure 3.3) along with reasoning that combines imperative (e.g., dependence analysis and loop distribution) and functional thinking (e.g., rewrite rules and recurrences as scans). In particular, the simplest parallel operator, **`map`**, is the most difficult one to differentiate because its purely functional semantics allow free variables to be freely read inside it; reverse mode AD replaces reads with accumulations (to the corresponding adjoint variable), which are not representable as a combination of classical data-parallel constructs. We report safe support for accumulations inside **`map`**s by introducing *accumulators*, which may can be thought of as a read-only view with an array. We also introduce optimizations to transform accumulators into more specialized constructs (such as reductions), which are further optimized for locality and may yield speedups close to one order of magnitude.

Our overall contribution is an end-to-end AD technique that supports nested parallel combinators as well as nesting of forward and reverse mode. We also report on an implementation of the technique as a compiler pass for the Futhark programming language. Our specific contributions are:

- **Redundant Execution:** A redundant execution technique for reverse mode AD that eliminates the need for tape and does not introduce re-execution for perfectly nested scopes other than loops.

- **Rewrite Rules:** A set of rewrite rules for differentiating higher-order parallel combinators, including in the presence of free variables.

[1] Our approach is not "AD-efficient" because there is no constant bound for the scope depth that programs may have. We view this more as a tradeoff rather than a weakness because tape-based systems may incur order-of-magnitude overheads (due to inefficient utilization of locality) which are higher than the depth of most programs.

- **Optimizations:** A collection of optimizations that rewrite common cases of accumulators to reductions, which benefit from specialized code generation.
- **Implementation:** We report on an implementation of our reverse mode AD technique in the Futhark compiler.
- **Evaluation:** An experimental evaluation of the implementation that demonstrates sequential and GPU performance competitive with Tapenade [4], Enzyme [63], PyTorch [73], and JAX [12].

3.2 Preliminaries

3.2.1 Forward mode

(a)

$$
\begin{aligned}
&\texttt{def } P\ x_0\ x_1 = \\
&\quad \texttt{let } v_0 = \sin x_0 \\
&\quad \texttt{let } v_1 = x_1 \cdot v_0 \\
&\quad \texttt{let } v_2 = v_0 \cdot v_1 \\
&\quad \texttt{let } y_0 = v_1 + v_2 \\
&\quad \texttt{let } y_1 = \cos v_2 \\
&\quad \texttt{in } [y_0, y_1]
\end{aligned}
$$

(b)

$$
\begin{aligned}
&\texttt{def } P_{\text{fwd}}\ x_0\ x_1\ \dot{x}_0\ \dot{x}_1 = \\
&\quad \texttt{let } v_0 = \sin x_0 \\
&\quad \texttt{let } \dot{v}_0 = \cos x_0 \cdot \dot{x}_0 \\
&\quad \texttt{let } v_1 = x_1 \cdot v_0 \\
&\quad \texttt{let } \dot{v}_1 = v_0 \cdot \dot{x}_1 + x_1 \cdot \dot{v}_0 \\
&\quad \texttt{let } v_2 = v_0 \cdot v_1 \\
&\quad \texttt{let } \dot{v}_2 = v_1 \cdot \dot{v}_0 + v_0 \cdot \dot{v}_1 \\
&\quad \texttt{let } y_0 = v_1 + v_2 \\
&\quad \texttt{let } \dot{y}_0 = \dot{v}_1 + \dot{v}_2 \\
&\quad \texttt{let } y_1 = \cos v_2 \\
&\quad \texttt{let } \dot{y}_1 = -\sin v_2 \cdot \dot{v}_2 \\
&\quad \texttt{in } ([y_0, y_1], [\dot{y}_0, \dot{y}_1])
\end{aligned}
$$

(c)

$$
\begin{aligned}
&\texttt{def } P_{\text{rev}}\ x_0\ x_1\ \overline{y}_0\ \overline{y}_1 = \\
&\quad \texttt{let } v_0 = \sin x_0 \\
&\quad \texttt{let } v_1 = x_1 \cdot v_0 \\
&\quad \texttt{let } v_2 = v_0 \cdot v_1 \\
&\quad \texttt{let } y_0 = v_1 + v_2 \\
&\quad \texttt{let } y_1 = \cos v_2 \\
&\quad \texttt{let } \overline{v_2}\ \mathrel{+}= -\sin v_2 \cdot \overline{y}_1 \\
&\quad \texttt{let } \overline{v}_1\ \mathrel{+}= \overline{y}_0 \\
&\quad \texttt{let } \overline{v}_2\ \mathrel{+}= \overline{y}_0 \\
&\quad \texttt{let } \overline{v}_0\ \mathrel{+}= v_1 \cdot \overline{v_2} \\
&\quad \texttt{let } \overline{v}_1\ \mathrel{+}= v_0 \cdot \overline{v}_2 \\
&\quad \texttt{let } \overline{x}_1\ \mathrel{+}= v_0 \cdot \overline{v}_1 \\
&\quad \texttt{let } \overline{v}_0\ \mathrel{+}= x_1 \cdot \overline{v}_1 \\
&\quad \texttt{let } \overline{x}_0\ \mathrel{+}= \cos x_0 \cdot \overline{v}_0 \\
&\quad \texttt{in } ([y_0, y_1], [\overline{x}_0, \overline{x}_1])
\end{aligned}
$$

Figure 3.1: (a) a program P, (b) the forward mode AD transformation of P with augmentations to P highlighted in blue, (c) the reverse mode AD transformation of P with augmentations highlighted in red.

In AD, we seek to answer a basic question: how do changes to the inputs of a program affect its outputs? In *forward mode* AD this question is answered by computing the *tangents* of programs variables. The tangent of a program variable v_i measures how v_i changes as the program's input changes and is defined as

$$\dot{v}_i = \sum_{j=0}^{n-1} \frac{\partial v_i}{\partial x_j}, \tag{3.1}$$

where $x_0, \ldots, x_{n-1}$ are the n input variables of the program.

As an example, consider variable v_2 from program P in Figure 3.1 (a): $\dot{v}_2$ measures how v_2 changes as the inputs x_0 and x_1 change. Since $v_2 = v_0 \cdot v_1$, changes to x_0 and x_1 affect v_2 indirectly via v_0 and v_1. The chain rule of calculus says that we can express $\dot{v}_2$

in terms of $\dot{v}_0$ and $\dot{v}_1$, scaled by the sensitivity of v_2 to v_0 and v_2 to v_1, respectively:

$$\dot{v}_2 = \frac{\partial v_2}{\partial v_0} \cdot \dot{v}_0 + \frac{\partial v_2}{\partial v_1} \cdot \dot{v}_1 = v_1 \cdot \dot{v}_0 + v_0 \cdot \dot{v}_1.$$

Computing the tangent of v_2 is now reduced to computing the tangents of v_0 and v_1. To compute $\dot{y}_0$ and $\dot{y}_1$—the answer to our original question of how the outputs of a program change with its inputs—we follow the same procedure. Because every non-output program variable must have at least one output which (indirectly or directly) depends on it (otherwise it's dead code and can be eliminated), this amounts to computing the tangent of all intermediate variables in program order. The *forward mode rewrite rule* for **`let`**-statements in Figure 3.2 encapsulates this program transformation:

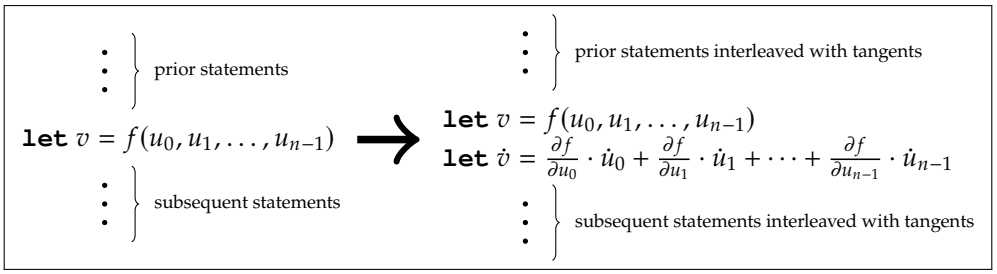


Figure 3.2: The forward mode rewrite rule for **`let`**-statements.

The forward mode rule preserves the original statement **`let`** $v = f(u_0, u_1, \ldots, u_{n-1})$ because the derivatives appearing in tangent expressions may depend on variables in the original program.

Application of the forward mode rewrite rule to program P in Figure 3.1 (a) yields the body of the differentiated program P_{fwd}, shown in Figure 3.1 (b). For example, applying the forward mode rewrite rule to the statement **`let`** $v_2 = v_0 \cdot v_1$ in P yields the statements

$$\begin{aligned} &\textbf{let } v_2 = v_0 \cdot v_1 \\ &\textbf{let } \dot{v}_2 = v_1 \cdot \dot{v}_0 + v_0 \cdot \dot{v}_1, \end{aligned}$$

in P_{fwd}. Notice that the inputs of P_{fwd} are also augmented with the tangents of the input variables; each invocation of P_{fwd} computes the derivative of P with respect to a particular direction as defined by these tangents. For example, to find the derivative with respect to the i-th input, we set $\dot{x}_i = 1$ and all other input tangents to 0 (which defines the directional of the derivative to be along x_i). This has the consequence that, if a program Q has n inputs, n executions of Q's forward mode AD transformation are required to yield the derivative with respect to each input, i.e., the full Jacobian matrix $\mathbf{J}$. For program P, which has two inputs, the Jacobian is computed via two invocations of P_{fwd}:

$$\textbf{def } \mathbf{J}\ x_0\ x_1 = \texttt{transpose}\ [(P_{\text{fwd}}\ x_0\ x_1\ 1\ 0).1, (P_{\text{fwd}}\ x_0\ x_1\ 0\ 1).1],$$

where $e.n$ projects out the n-th component of tuple e. Since each invocation of P_{fwd} yields one column of the Jacobian matrix, forward mode AD actually computes a so-called *Jacobian-vector product* or *jvp*, i.e.:

$$P_{\text{fwd}}\ x_0\ x_1\ \dot{x}_0\ \dot{x}_1\ \equiv\ \textbf{matvec}\ (\mathbf{J}\ x_0\ x_1)\ (\texttt{transpose}\ [\dot{x}_0, \dot{x}_1]),$$

where **matvec** is matrix-vector multiplication. Additionally, notice that the output of Figure 3.1 (b) includes both the original output of the program (a.k.a. the *primal* outputs) and the tangents. Often, the programmer needs both values (e.g., in gradient descent) and it's more efficient to compute them in conjunction rather than computing the primal value redundantly in a separate evaluation.

3.2.2 Reverse mode

Forward mode AD computes how a program's outputs are affected by its inputs from the top down: the output tangents are computed via intermediate program tangents (which themselves are computed via tangents of variables appearing even earlier in the original program) as dictated by Figure 3.2. In contrast, *reverse mode* AD computes how a program's outputs are affected by its inputs from the bottom-up. Whereas the quantity of interest in forward mode AD computes are the program variables' tangents, in reverse mode AD the corresponding quantity is each variable's *adjoint*, which quantifies its sensitivity to the program's output. The adjoint of a program variable v_i is written $\overline{v_i}$ and is defined as

$$\overline{v_i} = \sum_{j=0}^{m-1} \frac{\partial y_j}{\partial v_i}, \tag{3.2}$$

where $y_0, \ldots, y_{m-1}$ are the m outputs of the program.

In Figure 3.1 (a), notice that y_0 depends on v_0 indirectly through its dependence on v_1 and v_2; the chain rule says that $\overline{v_0}$ is simply the addition of $\overline{v_1}$ and $\overline{v_2}$, each scaled by the sensitivity of v_1 to v_0 and v_2 to v_0, respectively:

$$\overline{v_0} = \sum_{j=0}^{m-1} \frac{\partial y_j}{\partial v_0} = \sum_{j=0}^{m-1} \frac{\partial y_j}{\partial v_1} \cdot \frac{\partial v_1}{\partial v_0} + \frac{\partial y_j}{\partial v_2} \cdot \frac{\partial v_2}{\partial v_0} = \frac{\partial v_1}{\partial v_0} \cdot \overline{v_1} + \frac{\partial v_2}{\partial v_0} \cdot \overline{v_2}.$$

Notice that the adjoints of variables that appear earlier in the program depend on the adjoints that appear later in the program; computing adjoints is inherently bottom-up, and in reverse mode AD the adjoint of each variable is determined in reverse program order. Since adjoint variables have the reverse dependencies of the original program variables, this necessitates that the original program (a.k.a., the primal program) must first be executed (to bring all variables into scope as the adjoints may depend on them) before any adjoints can be computed. A single variable may be used on the right-hand side of multiple statements, meaning that their adjoints must in general accumulate contributions throughout the program since each usage of a variable contributes to its adjoint. All of this is captured in the reverse mode rewrite rule in Figure 3.3.

Application of the reverse mode rewrite rule on P in Figure 3.1 (a) yields the reverse mode differentiated program P_{rev}, shown in Figure 3.1 (c). As an example, applying the reverse mode rewrite rule to the statement **let** $v_2 = v_0 \cdot v_1$ in P yields the lines

$$\begin{aligned}
&\textbf{let } v_2 = v_0 \cdot v_1 \\
&\qquad\vdots \\
&\textbf{let } \overline{v_0} \mathrel{+}= v_1 \cdot \overline{v_2} \\
&\textbf{let } \overline{v_1} \mathrel{+}= v_0 \cdot \overline{v_2},
\end{aligned}$$

in P_{rev}. Notice the inputs of P_{rev} are augmented with the adjoints of the outputs; in analogy to forward mode AD—where the input tangents determine the direction of the

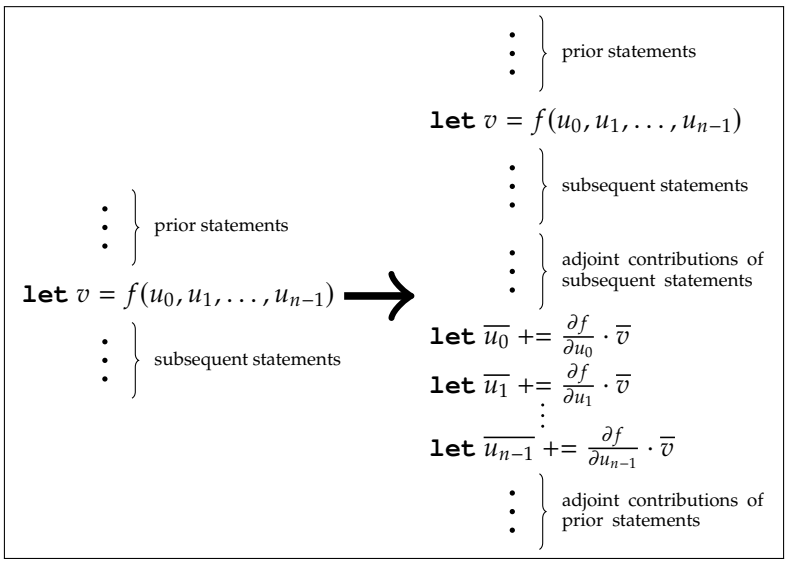


Figure 3.3: The reverse mode rewrite rule for **let**-statements. The assignment u_i += $\partial f/\partial u_i \cdot \overline{v}$ is syntactic sugar for $\overline{u_i} = \overline{u_i} + \partial f/\partial u_i \cdot \overline{v}$ where the $\overline{u_i}$ on the right-hand side is shadowed by the syntactically identical but distinct $\overline{u_i}$ on the left-hand side. All adjoints that have not been assigned to yet in the program are defined to be zero: if $\overline{u_i}$ += $\partial f/\partial u_i \cdot \overline{v}$ is the first contribution to $\overline{u_i}$, then the statement is equivalent to $\overline{u_i} = \partial f/\partial u_i \cdot \overline{v}$.

derivative—these determine the direction that the outputs change in affecting changes on the inputs. To find the derivative with respect to the i-th output, we set $\overline{y}_i = 1$ and all other output adjoints to 0; the Jacobian matrix of P is computed by the program

$$\textbf{def J}\ x_0\ x_1 = [(P_{\text{rev}}\ x_0\ x_1\ 1\ 0).1, (P_{\text{rev}}\ x_0\ x_1\ 0\ 1).1],$$

requiring two invocations of P_{rev} because P has two outputs.

Each invocation of P_{rev} yields one row of the Jacobian matrix **J** and corresponds to the *vector-Jacobian product* or *vjp*. That is,

$$P_{\text{rev}}\ x_0\ x_1\ \overline{y}_0\ \overline{y}_1\ \equiv\ \textbf{vecmat}\ [\overline{y}_0, \overline{y}_1]\ (\mathbf{J}\ x_0\ x_1),$$

where **vecmat** is vector-matrix multiplication.

A program Q with n inputs and m outputs requires m evaluations of the reverse mode differentiated program, Q_{rev}, to yield the adjoint of all n inputs. This means that if $n < m$, fewer evaluations of Q_{rev} are required to yield the full Jacobian than of the forward mode differentiated program, Q_{fwd}. In many applications, $n \ll m$ (e.g., scalar-valued functions have $m = 1$); in these applications the reverse mode of AD is vastly more efficient than the forward mode as yielding the full Jacobian will only require a single execution of Q_{rev} (but n executions of Q_{fwd}).

3.2.3 AD Interface

The main interface for AD is via two higher-order functions, **jvp** and **vjp**, which correspond to forward and reverse mode AD, respectively. The types of **jvp** and **vjp**

are

$$\texttt{jvp} \ : \ (f : \alpha \to \beta) \to (x : \alpha) \to (dx : \alpha) \to \beta,$$
$$\texttt{vjp} \ : \ (f : \alpha \to \beta) \to (x : \alpha) \to (dy : \beta) \to \alpha,$$

respectively. Each transforms a function $f : \alpha \to \beta$ into its derivative (if it exists) at $x : \alpha$. The additional arguments $dx : \alpha$ and $dy : \beta$ for **jvp** and **vjp** correspond to the input tangents and output adjoints, respectively. That is, if the input x to f is perturbed by dx, **jvp** reports how much the output changes, which is why it returns something of type β. On the other hand, **vjp** answers how much x must change to observe an output difference of dy, returning something of type α.

We also expose the functions **jvp2** and **vjp2**, with types

$$\texttt{jvp2} \ : \ (f : \alpha \to \beta) \to (x : \alpha) \to (dx : \alpha) \to (\beta, \beta),$$
$$\texttt{vjp2} \ : \ (f : \alpha \to \beta) \to (x : \alpha) \to (dy : \beta) \to (\beta, \alpha),$$

which are semantically defined as **jvp2** $f\ x\ dx \equiv (f\ x, \texttt{jvp}\ f\ x\ dx)$ and **vjp2** $f\ x\ dy \equiv (f\ x, \texttt{vjp}\ f\ x\ dy)$, respectively. The utility of **jvp2** and **vjp2** lies in the fact that the primal computation (i.e., $f\ x$) has many computations in common with the derivative computations (since derivative computations in general depend on primal values). In general, it's more efficient to compute the derivative of f in concert with $f\ x$—see Section 3.2.5 for an example.

Forward mode (**jvp**) is implemented in close analog to the dual number formulation described in [6] and won't be discussed further. Reverse mode (**vjp**) is more complex, requiring computation of adjoints in reverse program order and accumulations into adjoint variables (which requires special consideration to do efficiently), and thus, reverse mode is the focus of the remainder of this chapter.

3.2.4 Source Language

We perform our transformation on the data-parallel language introduced in Section 2.2. While the source language supports high-level features like higher-order functions and polymorphism, in this chapter we consider only a simplified monomorphic subset of the language with a number of restrictions—this models the approach in the actual Futhark compiler, where its high-level features are compiled away using a variety of techniques [24, 42] before we perform AD. Further, a significant battery of standard optimizations (including common subexpression elimination, constant folding, and aggressive inlining) is also applied prior to AD.

The only remaining second-order functions are the SOACs. Lambdas (i.e., unnamed functions) can only appear syntactically in SOACs and **vjp**/**jvp**, and are not values. As such we do not suffer from "perturbation confusion" [55]. The language is written in A-normal form [80] (ANF): all subexpressions are variable names or constants except for the *body* expression of **loops**, **if-then-else**-expressions and **let**-expressions. We re-emphasize from Section 2.2 that the language is purely functional: re-definitions of the same variable should be understood as a notational convenience for variable shadowing: **let** x += y is syntactic sugar for **let** $x = x + y$, where the x on the right-hand side is shadowed by the one on the left.

```
def cost points centers =
  sum (map (λp → min (map (dist2 p) centers))
           points)

def kmeans (k: i64) (n: i64) (d: i64) (b: i64)
           (points: [n][d]f32) =
  loop (centers: [k][d]f32 = ...) for i < b do
    let (cost', cost'') =
      jvp2 (λx → vjp (cost points) x 1) centers
           (replicate k (replicate d 1))
    let dx = map (map (/)) cost' cost''
    in map (map (-)) centers dx
```

Figure 3.4: Applying reverse (**vjp**) and forward (**jvp2**) mode AD to solve k-means. Recall that **jvp2** $f\ x\ dx$ returns both the primal value and the derivative (**jvp** only returns the derivative); this is useful because Newton's method requires both the gradient (`cost'`) and the diagonal of the Hessian (`cost''`). `dist2` computes the Euclidean distance.

3.2.5 Example: k-means

k-means clustering is an optimization problem that partitions n points P in d-dimensional space into k clusters centroids C that minimizes the cost function

$$f(C) = \sum_{p \in P} \min \left\{ ||p - c||^2, c \in C \right\}. \tag{3.3}$$

f can be minimized using Newton's method [11], which iteratively finds the minimizing cluster locations via the recurrence

$$C_{i+1} = C_i - \mathbf{H}(f(C_i))^{-1} \nabla f(C_i),$$

until convergence, where $\mathbf{H}(f(C_i))$ is the Hessian of f and $\nabla f(C_i)$ is the gradient of f. Since the centroids are independent of each other, $\mathbf{H}(f(C_i))$ is a diagonal matrix and the above computation can instead be written as

$$C_{i+1} = C_i - \nabla f(C_i) \oslash \text{diag}(\mathbf{H}(f(C_i)),$$

where $\oslash$ is element-wise division and $\text{diag}(\mathbf{H}(f(C_i))$ is the vector containing the diagonal elements of $\mathbf{H}(f(C_i))$.[2] This avoids computing the full Hessian as well as its inverse—an expensive operation if done naively!

Figure 3.4 above shows Futhark code with a function `cost` which implements the cost function (Equation (3.3)) along with a function `kmeans` which minimizes it and exploits sparsity of the Hessian via the recurrence above (realized by a **loop**); this shows how the **jvp**/**vjp** interface allows the programmer to exploit sparsity. Note the nesting of **vjp** inside **jvp2**, which allows the gradient to be differentiated to produce the Hessian.[3]

[2] Since the inverse of a diagonal matrix is obtained by replacing each element on the diagonal by its reciprocal.

[3] $\nabla f(C_i)$ can be directly computed by **vjp**; since $\mathbf{H}(f(C_i)) = \mathbf{J}(\nabla f(C_i))$, $\text{diag}(\mathbf{H}(f(C_i))) = \mathbf{H}(f(C_i)) \cdot \mathbf{1}$ is just a Jacobian-vector product and may be directly computed with a single iteration of **jvp** on $\nabla f(C_i)$ (computed by **vjp**), where $\mathbf{1}$ is a vector of all 1s.

3.3 Reverse Mode AD by Redundant Execution

This section discusses how the transformation operates across scopes without requiring an explicit tape (as in traditional reverse mode AD techniques): Section 3.3.1 gives an example and sketches the overall structure of the analysis, Section 3.3.2 demonstrates the analysis of loops, and Section 3.3.3 discusses the trade-offs related to redundant execution.

3.3.1 Transformation Rules Across Scopes

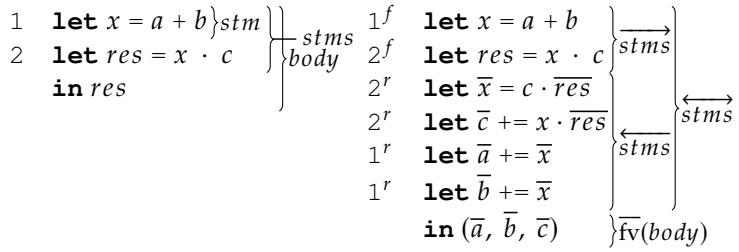


Figure 3.5: An example of applying the vjp_{body} rule (see Figure 3.6) to a body: the code on the left shows the original body and the code on the right shows the resulting differentiated body. The line numbers 1^f and 2^f indicate that the line corresponds to the forward sweep of lines 1 and 2 (of the original program), respectively. 1^r and 2^r indicate that the line is part of the return sweep corresponding to lines 1 and 2, respectively.

Figure 3.5 illustrates the reverse mode AD transformation on a simple example. The transformation acts on a body (labeled $body$), which consists of a list of statements (labeled $stms$) followed by a result (**in** res), as depicted in the code on the left of Figure 3.5. The right side of the figure shows the result of the transformation: it consists of a forward sweep (labeled $\overrightarrow{stms}$) which is a re-execution of the statements from the original program to bring into scope any variables which may be needed in the return sweep, and the reverse sweep itself (labeled $\overleftarrow{stms}$) which computes the new adjoint contributions to each variable, in reverse program order. The forward and return sweeps together constitute the statements of the differentiated body, which we label $\overleftrightarrow{stms}$. Finally, the result of the differentiated body consists of the adjoints of the free variables in the body, which are returned by $\overline{\text{fv}}(body)$, and consist of $\overline{a}$, $\overline{b}$, and $\overline{c}$. Only these adjoints can contribute to the adjoints of other program variables: all bound variables within the body will be out of scope once the body returns.

In the following sections, the reverse mode transformation VJP is implemented as a syntax-directed translation [60] where translation rules are defined for each syntactic category of the language. Figure 3.6 shows the primary translation rules for the VJP transformation and illustrates the treatment of new scopes.

We elide symbol table bookkeeping and assume that the adjoint $\overline{x}$ of a variable x is always available.[4] The VJP_{body} rule refers to a body of statements (coupled with a return), which **always begins a new scope**. The rule first extends the environment by binding the body result res to its adjoint $\overline{res}$ (not shown); this is safe since the transformation works backwards, hence the adjoint $\overline{res}$ is already available from the

[4] In practice, either the adjoint is available or it hasn't had any contributions yet, in which case a statement can be inserted which initializes the adjoint to a zero element of the appropriate type and shape.

$\text{V}_{\text{JP}_{\text{body}}}$ **applies to bodies of the form** $stms$ `in` res**:**

$$\text{V}_{\text{JP}_{\text{body}}}(\overline{res},\ stms\ \texttt{in}\ res) \Rightarrow \overleftrightarrow{stms}\ \texttt{in}\ (res,\ \overline{\text{fv}}(stms))$$
$$\texttt{where}\ \overleftrightarrow{stms} \leftarrow \text{V}_{\text{JP}_{\text{stms}}}(stms)$$

$\text{V}_{\text{JP}_{\text{stms}}}$ **folds over each statement:**

$$\text{V}_{\text{JP}_{\text{stms}}}(stm, stms) \Rightarrow (\overrightarrow{stms},\ \text{V}_{\text{JP}_{\text{stms}}}(stms),\ \overleftarrow{stms})$$
$$\texttt{where}\ (\overrightarrow{stms},\ \overleftarrow{stms}) \leftarrow \text{V}_{\text{JP}_{\text{stm}}}(stm)$$

$\text{V}_{\text{JP}_{\text{stm}}}$ **for a scalar multiplication statement:**

$$\text{V}_{\text{JP}_{\text{stm}}}(\texttt{let}\ x = a \cdot b) \Rightarrow (\overrightarrow{stms},\ \overleftarrow{stms})$$
$$\begin{aligned}\texttt{where}\ \overrightarrow{stms} &\leftarrow \texttt{let}\ x = a \cdot b\\ \overleftarrow{stms} &\leftarrow \texttt{let}\ \overline{a}\ \texttt{+=}\ b \cdot \overline{x}\\ &\quad\ \ \texttt{let}\ \overline{b}\ \texttt{+=}\ a \cdot \overline{x}\end{aligned}$$

$\text{V}_{\text{JP}_{\text{stm}}}$ **for an array-indexing statement:**

$$\text{V}_{\text{JP}_{\text{stm}}}(\texttt{let}\ y = a[i]) \Rightarrow (\texttt{let}\ y = a[i],\ \texttt{let}\ \overline{y} = \texttt{upd}\ i\ \overline{y}\ \overline{a})$$

$\text{V}_{\text{JP}_{\lambda}}$ **for a lambda function (**$\lambda x_1 \ldots x_n \rightarrow stms$ `in` res**):**

$$\text{V}_{\text{JP}_{\lambda}}(\overline{res},\ \lambda x_1 \ldots x_n \rightarrow stms\ \texttt{in}\ res) \Rightarrow \lambda x_1 \ldots x_n \rightarrow \overleftrightarrow{stms\ \texttt{in}\ res}$$
$$\begin{aligned}\texttt{where}\ \overleftrightarrow{stms}\ \texttt{in}\ (\_,\ \overline{\text{fv}}(stms)) &\leftarrow \text{V}_{\text{JP}_{\text{body}}}(\overline{res},\ stms\ \texttt{in}\ res)\\ \overleftrightarrow{body} &\leftarrow \overleftrightarrow{stms}\ \texttt{in}\ \overline{\text{fv}}(stms)\end{aligned}$$

Figure 3.6: The VJP code transformation for several syntactic categories. $\overline{x}$ denotes the adjoint of x, $\overline{\text{fv}}(body)$ returns the free variables of $body$, $\overrightarrow{stms}$ and $\overleftarrow{stms}$ denote the forward and return sweeps, respectively, generated for $stms$.

outer scope. The statements of the transformed body ($\overleftrightarrow{stms}$) are those generated by $\text{V}_{\text{JP}_{\text{stms}}}$, which demonstrates the **redundant execution mechanism** that eliminates the need for a separate tape abstraction. Each statement is processed individually by $\text{V}_{\text{JP}_{\text{stm}}}$, which produces a sequence of statements on the forward sweep ($\overrightarrow{stms}$) that brings into scope whatever information is necessary to execute the return sweep ($\overleftarrow{stms}$), which is always structured in the reverse order of the original statements.

The remainder of the rules in Figure 3.6 show how VJP works on a selection of individual statement types and functions. Observe that the operation of VJP on scalar multiplications is a direct application of the reverse mode rewrite rule specified in Figure 3.3.

On array-indexing statements of the form `let` $y = a[i]$, the return sweep must update $\overline{a}[i]$ with the contribution of $\overline{y}$, which is accomplished in the rule by `let` $\overline{y}$ = `upd` $i\ \overline{y}\ \overline{a}$. Semantically, `upd` $i\ v\ a$ returns a new value equal to a but with the value at index i changed to be $v + a[i]$. Operationally, the array a is directly modified in-place. To preserve purely functional semantics, we require that the "old" a and its aliases are never accessed again, similar to (but distinct from—the syntax involves *accumulators* and will be explained in-depth in Section 3.4.5) the in-place updates described in Chapter 2 and section 4.3.1.

Finally, the rule $\text{V}_{\text{JP}_{\lambda}}$ transforms an anonymous function; its result is obtained by calling $\text{V}_{\text{JP}_{\text{body}}}$ on the function body. Note that $x_1, \ldots, x_n$ are free variables in $stms$, so

their adjoints—if the variables were read in $stms$—are among $\overline{\text{fv}}(stms)$, which returns the adjoints of the free variables in $stms$.

3.3.2 Reverse Mode Transformation for Loops

(a) Original loop

$\vdots\}\ stms_{\text{before}}$
let $y'' =$
loop $y = y_0$
for $i = 0 \ldots m^k - 1$ **do**
$\vdots\}\ stms_{\text{loop}}$
in y'
$\vdots\}\ stms_{\text{after}}$

(c) Strip-mined loop

$\vdots\}\ stms_{\text{before}}$
let $y'' =$
loop $y_1 = y_0$
for $i_1 = 0 \ldots m - 1$ **do**
$\ldots$
loop $y_k = y_{k-1}$
for $i_k = 0 \ldots m - 1$ **do**
let $y = y_k$
let $i =$
$i_1 \cdot m^{k-1} + \ldots + i_k$
$\vdots\}\ stms_{\text{loop}}$
in y'
$\vdots\}\ stms_{\text{after}}$

(b) Reverse AD of loop

1 $\vdots\}\ \overrightarrow{stms}_{\text{before}}$
2 **let** ys_0 = **scratch** m^k (`sizeOf` y_0)
3 **let** $(y'',\ ys) =$
4 **loop** $(y,\ ys) = (y_0,\ ys_0)$
5 **for** $i = 0 \ldots m^k - 1$ **do**
6 **let** $ys[i] = y$
7 $\vdots\}\ stms_{\text{loop}}$
8 **in** $(y',\ ys)$
9 $\vdots\}\ \overrightarrow{stms}_{\text{after}}$
10 $\vdots\}\ \overleftarrow{stms}_{\text{after}}$
11 **let** $(\overline{y'''},\ \overline{fvs_{\text{loop}}}) =$
12 **loop** $(\overline{y},\ \overline{fvs_{\text{loop}}}) = (\overline{y''},\ \overline{\text{fv}}(stms_{\text{loop}}))$
13 **for** $i = m^k - 1 \ldots 0$ **do**
14 **let** $y = ys[i]$
15 $\vdots\}\ \overrightarrow{stms}_{\text{loop}}$
16 $\vdots\}\ \overleftarrow{stms}_{\text{loop}}$
17 **in** $(\overline{y'},\ \overline{fvs'_{\text{loop}}})$
18 **let** $\overline{y_0}\ \overline{+}{=}\ \overline{y'''}$
19 $\vdots\}\ \overleftarrow{stms}_{\text{before}}$

Figure 3.7: (a) A loop, (b) the result of applying the VJP transformation, and (c) the result of strip-mining it into a depth-k loop nest. In (b), $\text{VJP}_{\text{body}}(\overline{y},\ stms_{\text{loop}}$ **in** $y')$ is called to generate $\overrightarrow{stms}_{\text{loop}}$, $\overleftarrow{stms}_{\text{loop}}$, and $\overline{fvs'_{\text{loop}}}$, as defined in Figure 3.6.

This section illustrates the transformation rule for loops: Figure 3.7 (b) shows the result of VJP applied to the loop in Figure 3.7 (a). The forward sweep of the differentiated loop in Figure 3.7 (b) (lines 2–8) consists of the original loop except that its result and body are modified to checkpoint into array ys the value of y at the start of each iteration; ys is also declared as loop variant and initialized to ys_0, which is allocated (by **scratch**[5]) just before the loop statement. Importantly, **only the loops of the current scope are checkpointed**; a more deeply nested loop would be re-executed, not checkpointed.

The return sweep of the loop (lines 11 – 18) consists of a loop that iterates backward from $m^k - 1$ down to 0. The loop variant values consist of the adjoint of the primal loop

[5] **scratch** d s allocates an array of length d consisting of s-byte elements. `sizeOf` returns the size of its argument's type.

variant value(s) along with initial adjoints for the free variables in the loop, which are returned by $\overline{\text{fv}}(stms_{\text{loop}})$ (line 12). The first statement of the loop (line 14) re-installs the value of y of the current iteration from the checkpoint, so that the forward sweep of the loop body ($\overrightarrow{stms}_{\text{loop}}$) can be re-executed to bring into scope the variables needed by the return sweep of the loop body $\overleftarrow{stms}_{\text{loop}}$. The result of the reversed loop body (line 17) is the adjoint of the original result, $\overline{y'}$, together with the (updated) adjoints of the free variables used in the loop, $\overline{fvs'_{\text{loop}}}$. These are declared as variant through the loop (line 12) such that the updates of all iterations are recorded. Finally, the statement at line 18 semantically updates the adjoint of the loop initializer $\overline{y_0}$ with the result of the loop $\overline{y'''}$ ($\overline{+}$ denotes a vectorized addition that fits $\overline{y_0}$'s data type). This is because the first executed instruction of the source loop sets $y = y_0$ and the adjoint of y corresponding to the first iteration is the same as the adjoint of the loop result since the return sweep executes backwards.

With our strategy, the original loop is executed twice (lines 7 and 15). Strip-mining the loop (which has m^k iterations) into a depth-k loop nest, as shown in Figure 3.7 (c), suffers at worst a re-execution overhead factor of $k + 1$ when differentiated; however, it has a much smaller memory overhead than the original loop as each of the k m-iteration loops checkpoints its own variable, resulting in a memory footprint that is proportional to mk rather than m^k.[6] When m is constant, this results in the logarithmic space and time overhead case of the time-space tradeoff studied by Siskind and Pearlmutter [89]. We exploit the tradeoff in a simple and practical way by allowing the user to annotate the loops with a constant strip-mining depth, which is applied automatically before AD.

We conclude by noting that sequential loops are the only construct that require iteration checkpointing and that, importantly, **parallel constructs do not** because there are no dependencies between the inputs of parallel loops and subsequent iterations.

3.3.3 Perfect Nests Do Not Incur Redundant Execution

Figure 3.8 shows the code generated by applying VJP to a function whose body consist of perfectly nested scopes. (The differentiation of **`map`** is discussed in Section 3.4.5 and isn't necessary to understand this section.) It turns out that the code which constitutes the re-execution of the forward trace for each of the four scopes[7]—highlighted in red in the figure— is dead code. The reason is that perfectly nested scopes (other than loops) are guaranteed not to introduce recomputation because—by definition—their bodies consist of one (composed) statement: there are no intermediate variables whose adjoints will accumulate contributions in the return sweep (since there are no statements to reference them in the forward trace).[8]

It follows that overheads can be optimized by well-known transformations that generate perfect nests [10, 99], rooted in techniques like loop distribution and interchange. These transformations are also supported by the Futhark compiler [40]. With these transformations, we typically expect the forward sweep to be executed twice:

[6] The checkpoint of each of the k loops stores m versions of y.

[7] The outermost scope (scope 0) is the function's body, which consists of an outer **`map`** whose function's body (scope 1) consists of an **`if`**, whose **`else`**-body (scope 2) consists of an inner **`map`**, whose function body (scope 3) consists of a multiplication.

[8] Nested loops, however, must be flattened into a single loop because, even if perfectly nested, they may remain live due to checkpointing.

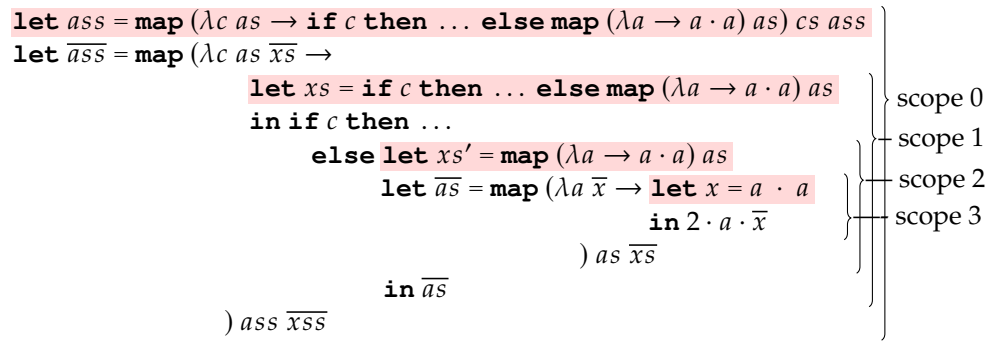


Figure 3.8: The body of the function generated by applying VJP to
$\lambda ass \to \texttt{map}(\lambda c\ as \to \texttt{if}\ c\ \texttt{then}\ \ldots\ \texttt{else}\ \texttt{map}\ (\lambda a \to a \cdot a)\ as)\ cs\ ass$
Red denotes the re-execution of the forward trace in each (new) scope. Note that all re-executions are dead code; this is guaranteed when the original code consists of perfectly nested scopes.

1. Once for the outermost scope because programs typically consist of multiple nests (the user may also require the result of the original program).
2. Once for the innermost scope that typically performs scalar computation, which is cheap to recompute. In comparison, vectorization or the use of tape would require such scalars to be retrieved from global memory, which has (an) order(s) of magnitude higher latency on a GPU.

Note that loops whose variant values are addition-based accumulations also do not introduce recomputation since the differentiation of addition does not depend upon primal values.

3.4 Rewrite Rules for Parallel Constructs

This section presents in detail the differentiation rules for **`reduce`**, **`hist`**, **`scan`**, **`scatter`**, and **`map`**.

3.4.1 Reduce

Recall from Section 2.2.2 that a **`reduce`** combines all elements of an array with a binary associative operator $\odot$: $\texttt{reduce}\ \odot\ e_\odot\ [a_0,\ a_1,\ \ldots,\ a_{n-1}] \equiv a_0 \odot a_1 \odot \cdots \odot a_{n-1}$, where $e_\odot$ is the neutral element of $\odot$.[9] Now, consider differentiating the statement

$$\texttt{let}\ y = \texttt{reduce}\ \odot\ e_\odot\ [a_0,\ a_1,\ \ldots,\ a_{n-1}]. \tag{3.4}$$

For each a_i, we can group the terms of the **`reduce`** as

$$\underbrace{a_0 \odot \cdots \odot a_{i-1}}_{l_i} \odot a_i \odot \underbrace{a_{i+1} \odot \cdots \odot a_{n-1}}_{r_i}.$$

[9] We assume that the array being reduced over is non-empty.

We then apply the main rule for reverse mode AD given in Figure 3.3, which results in the contributions

$$\overline{a_i} \mathrel{\overline{+}}= \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i}\,\overline{y},$$

where l_i and r_i are constants and can be computed by exclusive scans,[10] $\overline{+}$ denotes a (potentially) vectorized addition that matches $\overline{a_i}$'s datatype, and y is the variable bound to the result of the **reduce** (as in Equation (3.4)). The code for the right-hand side can be generically generated as a function f that is parameterized over a_i, l_i, and r_i (the VJP_λ rule is defined in Figure 3.6):

$$f \;\leftarrow\; \text{VJP}_\lambda\,(\overline{y},\; \lambda(l_i,\; a_i,\; r_i) \to l_i \odot a_i \odot r_i),$$

except f is modified to not return the adjoints of l_i and r_i since they aren't needed. Two (exclusive) **scan**s are used to compute the l_is and r_is for each a_i of the array $[a_0,\; a_1,\; \ldots,\; a_{n-1}]$; f can then be mapped over the result of the scans to compute the adjoint contribution to each element of the array being reduced over.

Summing things up, the forward sweep is simply the original **reduce** statement shown in Equation (3.4) above. Letting $as = [a_0,\; a_1,\; \ldots,\; a_{n-1}]$ and assuming for simplicity that $\odot$ has no free variables, the return sweep is

$$\begin{aligned}
&\textbf{let}\ ls = \textbf{scan}^{exc}\ \odot\ e_\odot\ as\\
&\textbf{let}\ rs = \textbf{reverse}\ as \triangleright \textbf{scan}^{exc}\ (\lambda x\ y \to y\ \odot\ x)\ e_\odot \triangleright \textbf{reverse}\\
&\textbf{let}\ \overline{as} \mathrel{\overline{+}}= \textbf{map}\ f\ ls\ as\ rs,
\end{aligned}$$

where $\triangleright$ is the *pipe forward operator* which enables composing functions left-to-right, e.g., $x \triangleright f \triangleright g = g\,(f\,x)$.

This translation is AD-efficient, but requires 8 memory accesses per element in comparison to one in the original **reduce**.[11] Fortunately, standard operators admit more efficient translations. Specialized rules for addition, min, and max are known [43]. The return sweep for (vectorized) addition adds $\overline{y}$ to each element of $\overline{as}$. For min/max, the forward sweep computes the minimal/maximal element y together with its first occurring index i_y, e.g.,

$$\textbf{let}\ (y,\; i_y) = \textbf{argmin}\ as,$$

and the return sweep updates only the adjoint of that element:

$$\textbf{let}\ \overline{as}[i_y] \mathrel{\overline{+}}= \overline{y}.$$

When the reduction operator is multiplication, we have

$$\overline{a_i} \mathrel{\overline{+}}= \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i}\,\overline{y} = \frac{\partial(l_i \cdot a_i \cdot r_i)}{\partial a_i}\,\overline{y} = l_i \cdot r_i \cdot \overline{y},$$

and discriminate three cases:

1. If all of as's elements are nonzero, then $l_i \cdot r_i = {}^{y}/_{a_i}$ and $y \neq 0$, hence we update each element as $\overline{a_i}$ += ${}^{y}/_{a_i} \cdot \overline{y}$.
2. If exactly one element at index i_0 is zero, then $l_i \cdot r_i$ is zero for all other elements and $\overline{a_{i_0}}$ += $y \cdot \overline{y}$.
3. Otherwise, for all i, $l_i \cdot r_i = 0$ and $\overline{as}$ remains unchanged.

[10] $\textbf{scan}^{exc}\ \odot\ e_\odot\ [a_0,a_1,\cdots,a_{n-1}] \equiv [e_\odot,\; a_0,\; a_0 \odot a_1,\; \ldots,\; a_0 \odot \cdots \odot a_{n-2}]$.

[11] Two each for ls and rs and then four for as.

These three cases are enabled by augmenting the forward sweep to compute the product of non-zero elements as well as the number of zero elements in as ($prod$ and $nzeros$, respectively):

$$
\begin{aligned}
&\textbf{let}\ (prod, nzeros) = \\
&\quad \textbf{map}\ (\lambda a \to \textbf{if}\ a == 0\ \textbf{then}\ (1,1)\ \textbf{else}\ (a,0))\ as \\
&\qquad \triangleright\ \textbf{reduce}\ (\lambda(prod', nzeros')\ (a, isZero) \to (prod' \cdot a, nzeros' + isZero))\ (1,0) \\
&\textbf{let}\ y = \textbf{if}\ 0 == nzeros\ \textbf{then}\ prod\ \textbf{else}\ 0,
\end{aligned}
$$

followed by setting the reduced result y accordingly. The return sweep computes the contributions by a parallel **map** and updates adjoints as discussed previously for case 1 (with adjoint contributions in the other two cases as discussed above).

3.4.2 Histogram

hist generalizes a histogram computation [37] by allowing the values from an array (as below) that fall into the same bin (i.e., an index from is) to be reduced with an arbitrary associative and commutative operator $\odot$ with neutral element $e_\odot$:

$$
\begin{aligned}
&\textbf{hist}\ :\ (\odot :\ \alpha \to \alpha \to \alpha) \to (e_\odot :\ \alpha) \to (is :\ [n]\texttt{int}) \to (as :\ [n]\alpha) \to [m]\alpha \\
&\textbf{let}\ hs = \textbf{hist}\ \odot\ e_\odot\ is\ as,
\end{aligned}
$$

is and as must be the same length and together constitute an index-value pairing. (See Section 2.2.2 for precise semantics and more details.)

Similar reasoning to that used for **reduce** suggests that two scans need to be applied to each subset of elements that fall in the same bin in order to compute the l_i and r_i terms for each i. The contributions to the adjoint $\overline{as}$ are then computed—as in the **reduce** case—by **map** $f\ ls\ as\ rs$ where f is as defined in Section 3.4.1. Assuming a constant key size, the scans can be implemented with the right asymptotic complexity by radix sorting as according to the corresponding bins (to ensure elements falling in the same bin are consecutive) and then by applying irregular segmented scans[12] (forwards and in reverse, in analogy to the rule for **reduce**) on the result. Taken all together the reverse sweep looks as follows:

$$
\begin{aligned}
&\textbf{let}\ flag = \textbf{map}\ (\lambda i \to i\ ==\ 0\ ||\ sis[i]\ \neq\ sis[i-1])\ (\textbf{iota}\ n) \\
&\textbf{let}\ rflag = \textbf{map}\ (\lambda i \to i\ ==\ 0\ ||\ flag[n-1])\ (\textbf{iota}\ n) \\
&\textbf{let}\ ls = \textbf{seg\_scan}^{exc}\ \odot\ e_\odot\ flag\ sas \\
&\textbf{let}\ rs = \textbf{reverse}\ sas\ \triangleright\ \textbf{seg\_scan}^{exc}\ (\lambda x\ y \to y\ \odot\ x)\ e_\odot\ \triangleright\ \textbf{reverse} \\
&\textbf{let}\ \overline{sas}\ \overline{+}= \textbf{map}\ f_{\text{hist}}\ ls\ as\ rs \\
&\textbf{let}\ \overline{as}\ \overline{+}= \textbf{scatter}\ (\textbf{replicate}\ \alpha)\ siota\ \overline{sas},
\end{aligned}
$$

where the three arrays sis, sas, and $siota$ are is, as and **iota** n sorted with respect to is, respectively (using radix sort). **iota** n returns a length n array of consecutive integers starting from 0: **iota** $n = [0, 1, \ldots, n-1]$. f_{hist} is defined by

$$
f_{\text{hist}}\ \leftarrow\ \text{V}_{\text{JP}\lambda}\ (\overline{hs}[is[i]],\ \lambda(l_i,\ a_i,\ r_i) \to l_i \odot a_i \odot r_i),
$$

except that it's modified (as for **reduce**) to not return the adjoints of l_i and r_i.

[12] A segmented scan **seg_scan** is akin to **scan** except it is also passed a flag array—i.e, an array consisting of 1s and 0s that is the same shape as the input array to **scan**—that denotes where each segment of the input array begins. The segmented scan then computes the inclusive prefix scan of each segment; the scan is "reset" at the beginning of each new segment. It has the following type signature:

$$
\textbf{seg\_scan}\ :\ (\odot :\ \alpha \to \alpha \to \alpha) \to (e_\odot :\ \alpha) \to (flags :\ [n]\texttt{int}) \to (as :\ [n]\alpha) \to [n]\alpha
$$

Since sorting is expensive in practice, we support specialized implementations for common operators: addition, multiplication, min, and max. As with the specialized implementation for **reduce**, the forward sweep consists of the **hist** statement enhanced with extended operators (see [13] for further details), and the return sweep is similar to that of **reduce**, except that in the function that computes updates to $\overline{as}[i]$, $\overline{y}$ is replaced with $\overline{hs}[is[i]]$.

3.4.3 Scan

Recall that the **scan** combinator returns a **reduce** (with respect to a given binary associative operator $\odot$) of every non-empty prefix of a list:

$$\begin{aligned}&\textbf{scan} : (\alpha \to \alpha \to \alpha) \to \alpha \to [\,]\alpha \to [\,]\alpha\\ &\textbf{scan}\ \odot\ e_\odot\ [a_0, a_1, \ldots, a_{n-1}] \equiv [e_\odot \odot a_0, e_\odot \odot a_0 \odot a_1, \ldots, e_\odot \odot a_0 \odot a_1 \odot \cdots \odot a_{n-1}].\end{aligned}$$

To derive the rule for **scan**, we adapt the semantic definition above into a loop-based formulation:

$$\begin{aligned}&ys[0] = as[0]\\ &\textbf{for}\ i = 1 \ldots n-1\\ &\quad ys[i] = ys[i-1] \odot as[i],\end{aligned}$$

and then apply the reverse mode rewrite rule from Figure 3.3 to obtain

$$\begin{aligned}&\textbf{for}\ i = n-1 \ldots 1\ \textbf{do}\\ &\quad \overline{as}[i]\ \overline{+}=\ \frac{\partial(ys[i-1] \odot as[i])}{\partial as[i]} \cdot \overline{ys}[i]\\ &\quad \overline{ys}[i-1]\ \overline{+}=\ \frac{\partial(ys[i-1] \odot as[i])}{\partial ys[i-1]} \cdot \overline{ys}[i]\\ &\overline{as}[0]\ \overline{+}=\ \overline{ys}[0].\end{aligned}$$

Since neither statement in the body of the loop depends on the other, they can be separated:

$$\begin{aligned}&\textbf{for}\ i = n-1 \ldots 1\ \textbf{do}\\ &\quad \overline{as}[i]\ +=\ \frac{\partial(ys[i-1] \odot as[i])}{\partial as[i]} \cdot \overline{ys}[i]\\ &\textbf{for}\ i = n-1 \ldots 1\ \textbf{do}\\ &\quad \overline{ys}[i-1]\ +=\ \frac{\partial(ys[i-1] \odot as[i])}{\partial ys[i-1]} \cdot \overline{ys}[i]\\ &\overline{as}[0]\ \overline{+}=\ \overline{ys}[0].\end{aligned}$$

The first loop is parallel and hence can be computed by a **map**. The contributions to $\overline{ys}$ in the second loop can be used to derive a recurrence of the form

$$\overline{ys}[i-1] = \overline{ys'}[i-1] + b_{i-1} \cdot \overline{ys}[i] \quad \text{for } i = n-1 \ldots 1.$$

where $b_i = \frac{\partial(ys[i] \odot as[i])}{\partial ys[i]}$ (and is a constant). The recurrence bottoms out at $n-1$ because $\overline{ys}[n-1]$ is unchanged by the loop. Re-indexing with $i \to i+1$, we obtain

$$\overline{ys}[i] = \overline{ys'}[i] + b_i \cdot \overline{ys}[i+1] \quad \text{for } i = n-2 \ldots 0.$$

which can be computed with a parallel **scan** [8] over the initial $\overline{ys}$ (before the contributions from the loop) and bs, which is an array of each b_i, i.e., $bs[i] = b_i$:

$$\textbf{let } \overline{ys} = \textbf{scan } (\lambda(r, b_l)\ (y, b_r) \to (y + b_r \cdot r, b_l \cdot b_r))\ (0, 1)\ \overline{ys}\ bs \triangleright \textbf{map } (.0),$$

where $(.0)\ (x, y) = x$. Suppose each element of ys is a vector of length d such that each element of bs is a $d \times d$ matrix. If d is a constant (e.g., the array elements are tuples of scalars) the work-depth asymptotic is preserved, but the translation is still not AD-efficient. We do, however, support vectorized operators for scan efficiently by interchanging the **scan** inside the **map**:

$$\textbf{scan } (\textbf{map } \odot)\ \overline{e_\odot}\ xs \implies \textbf{transpose } xs \triangleright \textbf{map } (\textbf{scan } \odot\ e_\odot) \triangleright \textbf{transpose},$$

and generate specialized code for the vectorized addition operator.

3.4.4 Parallel Scatter

Recall that $\textbf{let } ys = \textbf{scatter } xs\ is\ vs$ produces an array ys by updating in-place the array xs at the m indices in is with corresponding values of vs. The forward sweep saves the elements of xs that are about to be overwritten prior to performing the update:

$$\begin{aligned}&\textbf{let } xs_{\text{saved}} = \textbf{gather } xs\ is\\ &\textbf{let } ys = \textbf{scatter } xs\ is\ vs,\end{aligned}$$

where $\textbf{gather } xs\ is$ has the same semantics as $\textbf{map } (\lambda i \to xs[i])\ is$. The return sweep (1) first updates the adjoint of vs with the elements gathered from $\overline{ys}$, then (2) creates the adjoint of xs by zeroing out the elements from $\overline{ys}$ that were subject to the update, and (3) restores xs to its state before the update (from xs_{saved}):

$$\begin{aligned}&\textbf{let } \overline{vs} \mathrel{\overline{+}}= \textbf{gather } \overline{ys}\ is\\ &\textbf{let } \overline{xs} = \textbf{scatter } \overline{ys}\ is\ (\textbf{replicate } m\ 0)\\ &\textbf{let } xs = \textbf{scatter } ys\ is\ xs_{\text{saved}}.\end{aligned}$$

Both sweeps have $O(m)$ work and $O(1)$ depth.

3.4.5 Map

Consider the **map**

$$\textbf{let } xs = \textbf{map } (\lambda a \to stms\ \textbf{in } x)\ as.$$

If the lambda has no free variables, the return sweep is simply

$$\textbf{let } \overline{as} = \textbf{map } (\lambda(a,\ \overline{a}_0,\ \overline{x}) \to \overrightarrow{stms}\ \overleftarrow{stms}\ \textbf{in } \overline{a}_0 + \overline{a})\ as\ \overline{as}\ \overline{xs},$$

where $\overline{a}$ is the new contribution to the adjoint of each a, computed in $\overleftarrow{stms}$.

Handling free variables is more challenging because the differentiated **map** must return the adjoint contributions to the free variables. A naive way of handling free variables is to turn them into bound variables. For example,

$$\textbf{map } (\lambda i \to as[i])\ is,$$

may be converted into

$$\textbf{map } (\lambda(i,\ as') \to as'[i])\ is\ (\textbf{replicate } n\ as),$$

where n is the size of is. Unfortunately, this is asymptotically inefficient for partially used arrays—as in the case above—because the adjoint will be mostly zeroes.

In an impure language, adjoint updates for free variables can be implemented as a generalized reduction [53][13] wherein the adjoint of a free array variable $as[i]$ is updated with an operation $\overline{as}[i]$ += v, implemented with atomics or locks in the parallel case. In our pure setting, we instead introduce *accumulators* that preserve purity and guarantee the generalized reduction properties at the type level. An array can be temporarily turned into an accumulator with **withacc**:[14]

$$\textbf{withacc} \ : \ [d]\alpha \ \to (\textbf{acc}(\alpha) \to \textbf{acc}(\alpha)) \to [d]\alpha.$$

Intuitively we view accumulators as a write-only view of an array. Semantically, accumulators are lists of index/value pairs, each denoting an update of an array. Accumulators are updating via an update function:

$$\textbf{upd} \ : \ \texttt{int} \to \alpha \to \textbf{acc}(\alpha) \to \textbf{acc}(\alpha).$$

When we use **upd** on an accumulator, we add an index/value pair to its list, returning a new accumulator. Operationally, **upd** immediately writes to the underlying array and does not actually maintain a list of updates in memory. The purpose of accumulators is to allow the compiler to reason purely functionally, in particular ensuring that all data dependencies are explicit and allowing efficient code generation. Accumulators are similar to the accumulation effects in Dex [71] and have the same motivation. The main difference is that Dex requires every part of the compiler to be effect-aware, whereas in Futhark accumulators are a realized via a couple of expressions confined to the IR.

Free array-typed variables in the body of **map**s are thus turned into accumulators while generating return sweep code for the **map**, during which we can perform adjoint updates directly via **upd**. We allow implicit conversion between accumulators and arrays of accumulators, as this allows us to directly map them. For example,

$$\textbf{let} \ xs = \textbf{map} \ (\lambda i \to as[i]) \ is,$$

results in the return sweep code

$$\begin{array}{l}\textbf{let} \ \overline{as} = \textbf{withacc} \ \overline{as} \ (\lambda \overline{as}_{\text{acc}} \to \\ \quad \textbf{map} \ (\lambda(i, \ \overline{x}, \ \overline{as}) \to \textbf{upd} \ i \ \overline{x} \ \overline{as}) \ is \ \overline{xs} \ \overline{as}_{\text{acc}}),\end{array}$$

where we treat $\overline{as}_{\text{acc}}$ as an array of accumulators when passed to **map** and treat the result of the **map** as a single accumulator. This is efficient because accumulators have no runtime representation and it avoids tedious boilerplate. The equivalent imperative (generalized reduction) code is

$$\begin{array}{l}\textbf{forall} \ k = 0 \ldots \texttt{length}(is) - 1 \\ \quad \overline{as}[is[k]] \ += \overline{xs}[k].\end{array}$$

During the lifetime of an accumulator, the underlying array may not be used—this prevents observation of intermediate state. Since the Futhark IR is typed, this is mechanically ensured via a simple linear type system.

[13] A loop is a generalized reduction if all its cross-iteration dependencies are due to array variables xs that only appear in reduction statements of the form $xs[is[i]] \ \odot= e$, where e does not contain xs, and $\odot$ is associative and commutative.

[14] For simplicity we treat only single-dimensional arrays in this section, but the idea also works in the multidimensional case.

Accumulators are sufficient to express the adjoint computation inside **map**s because (1) any read from an array $as[i]$ is turned into an accumulation on $\overline{as}[i]$ and (2) the only place on the return sweep where $\overline{as}$ can be read outside an accumulation statement is the definition of as, which by definition is the last use of $\overline{as}$, hence it is safe to turn it back into an array there.

3.5 Implementation and Optimizations

We have implemented the reverse mode AD transformation as a pass in the Futhark compiler. The presented transformation rules were tuned to preserve fusion opportunities, both with constructs from a statement's differentiation and across statements.

We added accumulator support through the compiler—for the GPU backends, they ultimately boil down to atomic updates, such as `atomicAdd` in CUDA. Unoptimized accumulators, however, often result in suboptimal performance because they access memory in an uncoalesced fashion and are subject to conflicts, e.g., threads simultaneously accessing the same location. To address this, Section 3.5.1 presents accumulator optimizations that convert them into more specialized constructs (e.g., **map**-**reduce**) that are easier to optimize. Section 3.5.2 discusses how to optimize checkpointing for arrays that are constructed by in-place updates inside loop nests and how to support while loops.

3.5.1 Optimizing Accumulators

We demonstrate our accumulator optimizations on matrix-matrix multiplication. To aid readability, we omit **withacc** and also use an imperative notation in which **forall** loops denote **map** operations. Since we're dealing with two-dimensional arrays in this section, the indices for accumulator updates are specified with a pair, e.g., $\mathbf{upd}\,(i,\,k)\,v\,\overline{ass}$ can be understood as the imperative $ass[i,k]$ += v.

Suppose our two input matrices are $ass : [m][q]\alpha$ and $bss : [q][n]\alpha$ (where α is a numeric type); the following code computes their product $css : [m][n]\alpha$ by taking the dot product of each row of ass and column of bss:

$$\begin{array}{l}\mathbf{forall}\ i = 0 \ldots m-1\\ \quad \mathbf{forall}\ j = 0 \ldots n-1\\ \quad\quad css[i,j] = \mathbf{sum}\ (\mathbf{map}\ (\cdot)\ ass[i,:]\ bss[:,j]),\end{array}$$

where $\mathbf{sum} \equiv \mathbf{reduce}\ (+)\ 0$. Differentiating the code above results in the return sweep

$$\begin{array}{l}\mathbf{forall}\ i = 0 \ldots m-1\\ \quad \mathbf{forall}\ j = 0 \ldots n-1\\ \quad\quad \mathbf{forall}\ k = 0 \ldots q-1\\ \quad\quad\quad \overline{ass}[i,k] \mathrel{+}= bss[k,j] \cdot \overline{css}[i,j]\\ \quad\quad\quad \overline{bss}[k,j] \mathrel{+}= ass[i,k] \cdot \overline{css}[i,j],\end{array}$$

which is not efficient because (temporal) locality can be improved. To address this, we have designed and implemented a pass that transforms common accumulator access patterns into reductions. The analysis searches for the first accumulator directly nested in a perfect map nest and checks whether its indices are invariant across any of the parallel dimensions. (In the example above, $\overline{ass}$ is accumulated on indices $[i,k]$ that are both invariant to the parallel index j.) In such a case, the map nest is split into

two: the code on which the accumulated statement depends and the code without the accumulator statement,[15] which is simplified and treated recursively. The map nest encapsulating the accumulation is reorganized such that the invariant dimension (j) is moved innermost. (It is always safe to interchange parallel loops inwards.) The accumulation statement is taken out of this innermost map, which is modified to only produce the accumulated values, whose sum is rewritten to be the value accumulated into $\overline{ass}$:

$$
\begin{aligned}
&\textbf{forall}\ i = 0 \ldots m-1 \\
&\quad \textbf{forall}\ k = 0 \ldots q-1 \\
&\qquad s_a \ =\ \textbf{sum}\,(\textbf{map}\ (\cdot)\ bss[k,:]\ \overline{css}[i,:]) \\
&\qquad \overline{ass}[i,k]\ \mathrel{+}= s_a \\
&\textbf{forall}\ k = 0 \ldots q-1 \\
&\quad \textbf{forall}\ j = 0 \ldots n-1 \\
&\qquad s_b \ =\ \textbf{sum}\,(\textbf{map}\ (\cdot)\ ass[:,k]\ \overline{css}[:,j]) \\
&\qquad \overline{bss}[k,j]\ \mathrel{+}= s_b.
\end{aligned}
$$

The code now consists of two matrix multiplication-like kernels (with different parallel **forall** dimensions than the original). These are optimized by a later pass that performs block and register tiling whenever it finds an innermost **map-reduce** whose input arrays are invariant to one of the outer parallel dimensions. We have extended this pass (1) to support accumulators, (2) to keep track of the array layout (transposed or not), (3) to copy from global to shared memory in coalesced fashion for any layout, and (4) to exploit some of the parallelism of the innermost dot product as done in [78].

This optimization achieves an order-of-magnitude speedup at the application level for benchmarks dominated by matrix multiplication—see the GMM and LSTM benchmarks in sections 3.6.6 and 3.6.7.

3.5.2 Loop Optimizations and Limitations

As discussed in section 3.3, loop-variant variables are saved by default at the entry of each iteration of a loop. This technique does not preserve the work asymptotic of the original program when a loop-variant array is modified in-place. For example, the loop below constructs an array of length n in $O(n)$ work, but the checkpointing of the forward sweep requires $O(n^2)$ work:

$$
\begin{aligned}
&\textbf{loop}\ xs = xs_0\ \textbf{for}\ i\ =\ 1 \ldots n-1\ \textbf{do} \\
&\quad \textbf{let}\ xs[i,j] = as[i,j] \cdot xs[i-1,j] \\
&\quad \textbf{in}\ xs.
\end{aligned}
$$

Iteration-level checkpointing is not needed if the loop nest does not have any false dependencies (WAR or WAW):[16] since no loop-variant values are "lost" through the loop nest, it is sufficient to checkpoint xs only once at the entry to the outermost loop of the nest. Moreover, re-execution is safe because all the overwrites are idempotent. To exploit this, we allow the user to annotate loop parameters that are free of false dependencies—they're checkpointed upon entry to the loop nest in the forward sweep and restored just before entering the return sweep of the nest. Illustrating with the loop

[15] The optimization fires only if the number of redundant access to global memory introduced by splitting the map nest is less than two.

[16] The absence of false dependencies means that the loop has only true (RAW) dependencies or no dependencies at all.

above, we have:

$$
\begin{aligned}
&\texttt{let } xs_{\text{saved}} = xs_0 \\
&\texttt{let } xs' = \\
&\quad \texttt{loop } (xs, acc) = (xs_0, 0) \texttt{ for } i = 1 \ldots n-1 \texttt{ do} \\
&\qquad \texttt{let } xs[i,j] = as[i,j] \cdot xs[i-1,j] \\
&\qquad \texttt{in } xs \\
&\texttt{let } xs = xs_{\text{saved}} \\
&\texttt{let } (\overline{xs''}, \overline{as}) = \\
&\quad \texttt{loop } (\overline{xs}, \overline{as}) = (\overline{xs'}, \overline{as}) \texttt{ for } i = n-1 \ldots 1 \texttt{ do} \\
&\qquad \texttt{let } xs[i,j] = as[i,j] \cdot xs[i-1,j] \\
&\qquad \texttt{let } \overline{as}[i,j] \mathrel{+}= xs[i-1,j] \cdot \overline{xs}[i,j] \\
&\qquad \texttt{let } \overline{xs}[i-1,j] = as[i,j] \cdot \overline{xs}[i,j] \\
&\qquad \texttt{in } (\overline{xs}, \overline{as'}) \\
&\texttt{let } \overline{xs_0} \mathrel{+}= \overline{xs''}.
\end{aligned}
$$

Techniques in automatic parallelization can be used to automatically check the safety of such annotations, statically [32], dynamically [19], and anywhere in between [67].

A second issue relates to while loops, on which we cannot perform AD directly because their statically unknown iteration count hinders the allocation of checkpointing arrays. To address this issue, the user may annotate a while loop with an iteration bound n. The while loop is then transformed into an n-iteration for loop that contains a perfectly nested if-then-else expression, which only executes the valid iterations of the while loop. In the absence of such an annotation, the loop count is computed dynamically by an inspector.

Finally, a limitation of the current implementation is that it does not support loop-variant parameters that change their shape throughout the loop. In principle this can be handled by dynamic re-allocations, but this might be expensive on GPUs.

3.6 Experimental Evaluation

3.6.1 Parallel Hardware and Methodology

System	CPU	GPU	API
A100	2 × AMD EPYC 7352	NVIDIA A100	CUDA 11.6
MI100	2 × AMD EPYC 7352	AMD MI100	ROCm 5.0.1
2080 Ti	2 × Intel E5-2650	NVIDIA 2080 Ti	CUDA 11.3

Figure 3.9: Systems used for benchmarking.

We benchmark on three different Linux systems, detailed in Figure 3.9. We report mean runtime for 10 runs (following an initial run that is discarded), which includes all overheads, except transferring program input and result arrays between device and host. We report the absolute runtime of the differentiated and primal program and the "overhead" of differentiation that corresponds to the ratio between the two. In optimal AD, this ratio (counted in number of operations) is supposed to be a small constant [30], hence the ratio serves as a good measure of the efficiency of an AD implementation. We also report the memory usage of the primal (when applicable) and differentiated program on the dataset with maximal memory consumption for each experiment.

3.6.2 ADBench: Sequential AD Overhead

ADBench is a collection of benchmarks for comparing different AD tools [93] to which we have added Futhark implementations. We compile to sequential CPU code on the A100 system and report the AD overhead using the largest default dataset for each benchmark. We compare against Tapenade [4] and manually differentiated programs. The results are shown in Figure 3.10.

Tool	BA	D-LSTM	GMM	HAND	
				Comp.	Simple
Futhark	13.0	3.2	5.1	49.8	45.4
Tapenade	10.3	4.5	5.4	3758.7	59.2
Manual	8.6	6.2	4.6	4.6	4.4

Figure 3.10: ADBench sequential overheads; lower is better.

Futhark does well, in particular managing to outperform Tapenade in four out of five cases. For the exception, BA, the bottleneck is packing the result (which is a sparse Jacobian) in the CSR format expected by the tooling, which is code that is not subject to AD. The HAND benchmark has two variants: a "simple" one that computes only the dense part of the Jacobian and a "complicated" one that also computes a sparse part. Tapenade handles the latter poorly. On HAND, both Tapenade and Futhark perform poorly compared to manually differentiated code. Both BA and HAND produce sparse Jacobians where the sparsity structure is known in advance, which is exploited by passing appropriate seed vectors to `jvp`/`vjp`.

3.6.3 Comparison with Enzyme

Benchmark	Primal runtimes (s)		AD overhead	
	Original	Futhark	Futhark	Enzyme
RSBench	2.311	2.127	3.9	4.2
XSBench	0.244	0.239	2.7	3.2
LBM	0.071	0.042	3.4	6.3

Figure 3.11: Enzyme results, showing absolute runtimes and AD overheads. The Enzyme AD overheads are taken from [63]. RSBench and XSBench were measured on the 2080 Ti, while LBM is measured on the A100, similar to the systems used in the Enzyme paper. For LBM the workload is $120 \times 120 \times 150$ for 100 iterations. RSBench and XSBench use the "small" datasets with $10,200,000$ and $17,000,000$ "lookups", respectively.

Enzyme is an LLVM compiler plugin that performs reverse mode AD, including support for GPU kernels [63]. We have ported several benchmarks in order to compare our solution with Enzyme, with results in Figure 3.11. The Enzyme overheads are copied directly from [63]. RSBench and XSBench each constitute a large parallel loop that contains inner sequential loops and control flow, as well as indirect indexing of arrays. Our overhead is slightly smaller, although this may come down to micro-optimizations. LBM comprises a sequential loop containing a parallel loop. As Enzyme currently only supports differentiation of a single kernel, this requires some manual bookkeeping of the tape, whereas Futhark automatically handles the loop. Our overhead is significantly lower than Enzyme's, possibly because we can handle the tape more efficiently across the outer sequential loop.

3.6.4 Case Study 1: Dense k-means Clustering

	Data	Futhark (ms)		PyTorch	JAX	JAX(`vmap`)
		Manual	AD	(ms)	(ms)	(ms)
A100	$\mathbf{D}_0$	12.6	41.1	41.1	15.5	27.5
	$\mathbf{D}_1$	19.0	10.6	8.7	2.1	107.9
	$\mathbf{D}_2$	94.3	108.9	922.0	206.5	976.4
MI100	$\mathbf{D}_0$	24.6	35.6	94.5	–	–
	$\mathbf{D}_1$	22.5	10.5	40.2	–	–
	$\mathbf{D}_2$	309.5	264.2	2303.2	–	–

Figure 3.12: k-means clustering performance measurements for three different workloads. The **JAX** and **JAX**(`vmap`) implementations use array intrinsics and a vectorizing map, respectively. $\mathbf{D}_0 = (5, 494019, 35)$, $\mathbf{D}_1 = (1024, 10000, 256)$, $\mathbf{D}_2 = (1024, 2000000, 10)$ where each tuple is formatted as (k, n, d); k is the number of clusters and n the number of d-dimensional points. $\mathbf{D}_0$ corresponds to the KDD Cup dataset [50]. $\mathbf{D}_1$ and $\mathbf{D}_2$ were randomly generated. All data consists of 32-bit floating points.

In this section, we benchmark the k-means example of Section 3.2.5. As shown in Figure 3.4, in Futhark the cost function (Equation (3.3)) is written via nested **`map`** and **`reduce`** operations. In first-order languages like PyTorch, the cost function must be realized via array primitives; to efficiently compute the cost function, we expand the all pairs norm between points P and centroids C: $||P - C||^2 = P^2 + C^2 - 2PC^T$. In expanded form, all terms can be computed using vectorized operations, with the PC^T term being computed by matrix multiplication.

We compare against a handwritten Futhark histogram-based implementation as well as AD-based implementations in PyTorch and JAX on three qualitatively different datasets. In PyTorch and JAX, array intrinsics like matrix multiplication (and the differentiation thereof) are compiled to extremely efficient hand-tuned GPU code; to better compare with Futhark's programming model, a second JAX implementation using JAX's vectorizing map operation, `vmap`, was written in close analog to the Futhark implementation. The results are shown in Figure 3.12. When the histograms benefit from the optimizations discussed in [37], the handwritten implementation can show significant speedup over our AD approach: up to 3.3× on $\mathbf{D}_0$ on the A100. When they cannot, the AD approach can be faster due to differing amounts of parallelism. Additionally, note that the MI100 uses Futhark's OpenCL backend, which—unlike CUDA—doesn't support floating-point atomic add operations. Instead, atomic updates are implemented via a spinlock which can result in significant additional overhead in the atomic histogram updates of the manual implementation and, to a lesser extent, accumulator updates in the AD implementation. Futhark AD is on par with or significantly faster than PyTorch on all datasets (as high as 8.5× on the A100 and 8.7× on the MI100). The intrinsics-based JAX implementation demonstrates significant speedup over Futhark on the $\mathbf{D}_0$ and $\mathbf{D}_1$ datasets, but Futhark demonstrates a 1.9× speedup on the larger $\mathbf{D}_2$ dataset. On $\mathbf{D}_0$, the `vmap`-based JAX implementation has a 1.5× speedup over Futhark, but Futhark demonstrates speedups of around 10× on the other two datasets, likely a product of the fact that preservation of nested parallelism becomes more impactful as the number of clusters increases—we surmise JAX's vectorizing/flat approach limits locality optimizations. Our approach pays further dividends still: if the number of points is made larger, beyond $\mathbf{D}_2$'s two million, the PyTorch and JAX implementations run out of memory due to manifesting the entire $n \times k$ array of point-cluster distances.

3.6.5 Case Study 2: Sparse k-means Clustering

		Futhark (s)		**PyTorch** (s)	**JAX** (s)
	Workload	**Manual**	**AD**		
A100	movielens	0.06	0.16	1.47	0.38
	nytimes	0.09	0.30	5.24	1.35
	scrna	0.16	0.58	9.32	8.91
MI100	movielens	0.44	5.32	3.24	–
	nytimes	0.44	9.55	11.58	–
	scrna	0.42	2.87	20.81	–

Figure 3.13: Sparse k-means performance measurements for three NLP workloads. $k = 10$ for all datasets, with a fixed iteration count of 10 and a 32-bit representation. The movielens dataset uses data from the ML 20M dataset described in [33] with dimensions $(139K, 131K)$ and a density of 0.11%. The nytimes and scrna datasets are the same as used in [65], with dimensions $(300K, 102K)$ and $(66K, 27K)$ and densities of 0.23% and 7.3%, respectively.

We have implemented a sparse formulation of k-means clustering, which uses a dense representation for the centroids and a sparse representation for the input points. The Futhark implementations use the CSR format, while PyTorch and JAX both use the COO format.[17] As explained in the previous section, we compute the cost with vectorized operations in the PyTorch and JAX implementations.

Figure 3.13 shows runtimes on three publicly available sparse NLP workloads. On the A100, our AD is slower than the manual code by a factor between $2.5 - 3.7\times$ due to an optimization allowing updates to fit in the L2 cache [37]. On the A100, Futhark AD demonstrates speedups as high as $17.5\times$ against AD competitors. On the MI100, PyTorch has modest speedup over Futhark on the movielens dataset, but Futhark is faster on the two other datasets.

3.6.6 Case Study 3: GMM

To evaluate the parallelism preservation of our AD transformation, we compile the Futhark implementation of the GMM benchmark from the ADBench suite to parallel CUDA. We compare against ADBench's implementation of GMM in PyTorch (also run on CUDA), which we have improved (by a $> 10\times$ factor) by vectorizing all comprehensions. We benchmark on a selection of 1,000 and 10,000-point datasets from ADBench, see Figure 3.14. The runtime of the primal program is dominated by matrix multiplication ($\sim 70\%$).

As discussed in Section 3.6.4, matrix multiplication is a primitive in PyTorch; we expect that differentiation of matrix multiplication is implemented very efficiently. In Futhark there are no such primitives: matrix multiplication is written with **`map`**s, whose differentiation yield accumulators, which are further optimized as described in Section 3.5.1. The benchmark results are shown in Figure 3.15. The results demonstrate significant speedups over PyTorch on both systems, with an average speedup of $1.65\times$ on the A100 and of $2.75\times$ on the MI100. This demonstrates the feasibility of competitive AD performance in the absence of array primitives.

[17] PyTorch's functional AD constructs (`jvp` and `hvp`) currently raise runtime errors with CSR format. JAX's transformations only support batch COO representation.

Data	n	d	K		n	d	K
$\mathbf{D}_0$	1k	64	200	$\mathbf{D}_3$	10k	64	25
$\mathbf{D}_1$	1k	128	200	$\mathbf{D}_4$	10k	128	25
$\mathbf{D}_2$	10k	32	200	$\mathbf{D}_5$	10k	128	200

Figure 3.14: GMM ADBench parameters for the datasets used in Figure 3.15; n is the number of points, d the dimensionality of the input data, and K the number of Gaussian distributions. All datasets use 64-bit floats. The corresponding datasets may be found on the ADBench GitHub repository (https://github.com/microsoft/ADBench).

	Measurement	$\mathbf{D}_0$	$\mathbf{D}_1$	$\mathbf{D}_2$	$\mathbf{D}_3$	$\mathbf{D}_4$	$\mathbf{D}_5$
A100	**PyT. Jacob.** (ms)	7.4	15.8	15.2	5.9	12.5	64.8
	Fut. Speedup	2.1	2.2	1.4	1.6	1.5	1.0
	PyT. Overhead	3.5	4.9	2.8	3.2	4.0	3.2
	Fut. Overhead	2.0	1.8	1.9	2.7	2.8	2.8
MI100	**PyT. Jacob.** (ms)	20.9	51.5	42.5	20.7	38.5	193.1
	Fut. Speedup	3.3	4.0	2.1	2.9	2.5	1.7
	PyT. Overhead	5.9	5.3	2.4	2.6	3.1	2.8
	Fut. Overhead	3.0	2.9	3.0	2.8	2.8	2.8

Figure 3.15: GMM benchmark results on the A100 and MI100 systems. **Fut.** and **PyT.** refer to Futhark and PyTorch, respectively. **PyT. Jacob.** is the time to compute the full Jacobian of the objective function in PyTorch. On Futhark, block and register tile sizes of 16 and 3 were used, respectively.

3.6.7 Case Study 4: LSTM

			Speedups			
		PyTorch Jacob.	**Futhark**	**nn.LSTM**	**JAX**	**JAX**(`vmap`)
A100	$\mathbf{D}_0$	45.4 ms	3.0	11.6	4.5	0.3
	$\mathbf{D}_1$	740.1 ms	3.3	22.1	6.4	0.9
MI100	$\mathbf{D}_0$	89.8 ms	2.6	4.0	–	–
	$\mathbf{D}_1$	1446.9 ms	1.8	5.4	–	–

			Overheads			
		PyTorch	**Futhark**	**nn.LSTM**	**JAX**	**JAX**(`vmap`)
A100	$\mathbf{D}_0$	4.1	2.1	2.7	3.5	1.4
	$\mathbf{D}_1$	4.3	3.9	2.2	3.7	0.8
MI100	$\mathbf{D}_0$	5.0	4.2	7.2	–	–
	$\mathbf{D}_1$	7.9	3.9	6.6	–	–

Figure 3.16: LSTM speedups and overheads on $\mathbf{D}_0 = (1024, 20, 300, 192)$ and $\mathbf{D}_1(1024, 300, 80, 256)$ where each tuple is formatted as (bs, n, d, h); bs is the batch size, n the sequence length, d the dimensionality of the input data, and h the dimensionality of the hidden state. All data consists of 32-bit floating points. **nn.LSTM** refers to the `torch.nn.LSTM` implementation. Futhark was run with block and register tile sizes of 16 and 4.

Long Short-Term Memory (LSTM) [81] is a type of recurrent neural network architecture popular in named entity recognition and part-of-speech tagging [17, 79]. We

benchmark two LSTM networks with hyper parameters common in natural language processing [75, 17, 56, 54]. The AD-based implementations are based on the architecture in [81]. We also compare against PyTorch's `torch.nn.LSTM` class, which wraps the NVIDIA cuDNN LSTM implementation [16] on the A100 and AMD's MIGraphX on the MI100; note that both implementations are handwritten/optimized and feature manual differentiation. The results are shown in Figure 3.16. Futhark is about 3× faster than PyTorch on the A100 and slightly less on the MI100. As with GMM, LSTM is dominated by matrix multiplications. None of the AD-based implementations are competitive against the manual `torch.nn.LSTM` implementations.[18] JAX performs up to two times faster than Futhark when matrix multiplication is a highly optimized primitive and up to ten times slower when it's not.

3.6.8 Depth and Memory Consumption

Benchmark	Mem.	Depth	Mem. Overhead
RSBench	9.9 MiB	6	1.4
XSBench	225 MiB	6	1.0
LBM	550 MiB	5	33.6
GMM	4.1 GiB	4	2.1
LSTM	0.84 GiB	4	2.1

Figure 3.17: Primal memory footprints, maximal program depth, and the memory AD overhead of the benchmarks. For benchmarks with multiple datasets, the memory footprint is reported for the dataset with the largest memory consumption ($\mathbf{D}_5$ for GMM and $\mathbf{D}_1$ for LSTM).

Benchmark	Dataset	Futhark Manual	Futhark AD
Dense k-means	$\mathbf{D}_0$	188 MiB	172 MiB
Sparse k-means	scrna	2.7 GiB	2.7 GiB

Figure 3.18: Memory footprint comparison between the manual and AD Futhark implementations for dense and sparse k-means.

Figure 3.17 shows the memory overhead (the ratio of the memory footprint of the differentiated and primal programs) and maximal program depths of the benchmarks. Futhark's memory overhead on LBM is large: this can be ameliorated by annotating the outer loop to be strip-mined; doing so modestly increases the AD overhead from 3.4 to 4.5, but decreases the memory overhead from 33.6 to 8.7. The memory overheads of the remaining benchmarks demonstrate the efficiency of our approach: the forward and reverse passes combined should use roughly twice as much memory as the primal program—the remaining benchmarks achieve this or better. The GPU benchmarks also feature non-trivial depth; nevertheless we achieve competitive results, in part because the forward sweep is often executed only once or twice, irrespective of depth—see Section 3.3.3. Memory overheads aren't applicable to the k-means benchmarks, but it's informative to compare the AD-based implementations to the manual ones; Figure 3.18 shows that the implementations all use a similar amount of memory.

[18] The results appear correlated with the ratio between peak FLOPS with and without the usage of tensor cores.

3.7 Related Work

Reverse mode differentiation of reduce and scan is discussed in [72]. Our rule for **`reduce`** is similar but was developed independently [66] and we handle scan differently: our approach is less efficient for complex operands because we manifest the Jacobian matrices, but more efficient for single-value operands on GPUs as it requires less shared memory to implement the derived scan operator. Neither our rule for **`scan`** nor the one from [72] is asymptotics-preserving in general, but they are for most scans that occur in practice.

$\widetilde{F}$ is a functional array language that supports nested parallelism. Its AD implementation uses the forward mode, along with rewrite rules for exploiting sparsity in some cases [86].

Dex is a recent language built specifically to support efficient AD. Empirical benchmarks for AD in Dex have not been published, but we can compare with their approach [71]. In contrast to our conventional "monolithic" approach where reverse mode AD is a transformation completely distinct from forward mode, Dex uses a technique where the program is first linearized, producing a linear map, after which this linear map is then transposed, producing the adjoint code. Like Dex, we do not support recursion nor AD of higher-order functions. Dex does not make direct use of a tape in the classical sense, but instead constructs arrays of closures followed by defunctionalization. The actual runtime data structures will conceptually consist of *multiple* tapes in the form of multidimensional irregular arrays. Dex does not report strategies for check-pointing, or optimization of particular accumulation patterns as in Section 3.5.1, or of tape accesses.

The time-space tradeoff for reverse mode AD is systematically studied by Siskind and Pearlmutter [89]. Tapenade [4] supports a wealth of checkpointing techniques; our loop strip-mining technique is a practical and simple special case of that.

Enzyme demonstrates the advantage of performing AD after standard compiler optimizations have simplified the program [62]. Like Enzyme, we apply our AD transformation on a program that has already been heavily optimized by the compiler. But where Enzyme is motivated by performing AD on a post-optimization low-level representation, our work takes advantage of the information provided by high-level parallel constructs and post-AD optimizations to generate efficient code.

Enzyme has also been applied to GPU kernels where it makes use of AD-specific GPU memory optimizations including caching tape values in thread-local storage as well as memory-aware adjoint updates [63]. We achieve equivalent performance, but our approach is not based on differentiating single kernels—indeed, the GPU code we generate for a differentiated program may have a significantly different structure than the original program. For example, the optimized adjoint code for a matrix multiplication requires *two* matrix multiplications, each its own kernel, as discussed in Section 3.5.1.

Recent AD work on OpenMP details an approach to reverse mode AD for parallel loops [44]. Updates to free variables in loops are handled by sharing adjoints across threads and atomic updates; our approach to map is similar, but preservation of nested parallelism allows us to identify and reduce some updates into a single atomic update.

ML practitioners use tools such as PyTorch [73] that incorporate AD. These are less expressive than our language and do not support true nested parallelism but instead require the program to use flat (although vectorized) constructs. On the other hand, they can provide hand-tuned adjoints for the primitives they do support. (However, it would be straightforward to augment Futhark to allow the programmer to replace

AD-generated derivatives from `jvp`/`vjp` with hand-tuned alternatives when desired.)

JAX is another such example; it supports automatic differentiation of pure Python code and just-in-time (JIT) compilation to XLA HLO [26, 12]. Unlike PyTorch, JAX also features a vectorizing map operation which often yields good performance on some workloads. However, applying (reverse mode) AD on flat-parallel vectorized code may prevent further memory-locality optimizations; our approach applies AD to nested-parallel code, which enables further optimization opportunities, as demonstrated by our treatment of matrix multiplication-like computations.

Reverse mode AD has also been implemented in DSLs for stencil computations [45]. The challenge here is to combine AD with loop transformations such that the resulting code is a stencil itself and thus can be optimized with the repertoire of existent optimizations. AD has also been described for tensor languages that support constrained forms of loops, which in particular has the benefit of not requiring the use of tapes [7].

3.8 Conclusions

We have presented a fully operational compiler implementation of both reverse and forward mode AD in a nested-parallel, hardware-neutral functional language. Our transformation is based on using redundant execution to eliminate the need for an explicit tape and is performed before the parallelism of a program is mapped to hardware. It thus benefits from specialized rules for parallel constructors and flexibility in aggressively optimizing the original and AD code independently. Our experimental evaluation shows that our approach is effective in practice and competitive with both well-established frameworks that encompass more specialized languages such as PyTorch and JAX and with newer research efforts aimed at a lower-level language, such as Enzyme.

Data-Availability Statement

An artifact of our AD prototype in Futhark that reproduces the benchmarking results of Section 3.6 is available on Zenodo [84]. Additionally, AD support has been integrated into the Futhark compiler and is now included in its standard releases.

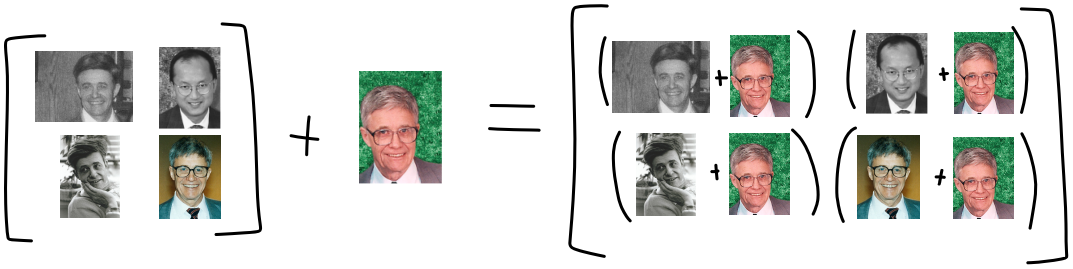
Acknowledgments

We are grateful to Rory Mitchell for suggesting the AD formulation of k-means, to Lotte Bruun and Ulrik Larsen for integrating the AD rules for `scan` and `hist`, and to Martin Elsman for using Futhark's AD infrastructure and providing valuable feedback.

This work has been supported by the Independent Research Fund Denmark (DFF) under the grants *Deep Probabilistic Programming for Protein Structure Prediction* and *FUTHARK: Functional Technology for High-performance Architectures*, and by the UCPH Data+ grant: *High-Performance Land Change Assessment*.

Chapter 4

AUTOMAP



This chapter is an adaptation of the following publication:

> Robert Schenck, Nikolaj Hey Hinnerskov, Troels Henriksen, Magnus Madsen, and Martin Elsman. "AUTOMAP: Inferring Rank-Polymorphic Function Applications with Integer Linear Programming". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA2 (2024). DOI: 10.1145/3689774. URL: https://doi.org/10.1145/3689774

The image above illustrates implicit replication and mapping, using photos of Kenneth E. Iverson (the creator of APL and J) and Roger Hui (co-creator of J) as the elements.

4.1 Introduction

Dynamically typed programming languages, such as Python or JavaScript, are frequently lauded for their ease-of-use, conciseness, and flexibility. Programmers using these languages often prefer to keep details implicit, in particular if the details can be determined uniquely from the context; however, even though the resulting code may be more ergonomic/compelling, bugs and ambiguities may be "swept under the carpet" if the programmer isn't careful. Static type systems can detect and prevent such bugs at compile-time, but often come with a notational cost: programmers are burdened by adding type annotations. A static type system with full type inference gives the best of both worlds; programmers have the same static guarantees but do not have to write any type annotations.

Unfortunately, static type systems sacrifice flexibility and many aspects of dynamic programming languages are difficult to model in static type systems. The implicit lifting of scalar operators across array operands in dynamic array languages like NumPy is one such aspect. For example, if xs and ys are vectors of equal length, one can write

$xs + ys$ for their element-wise addition. Further, when mixing arrays of different rank, NumPy performs *broadcasting*—which we refer to as *replication*—and implicitly adds extra dimensions to the smaller-rank array (by replicating its elements) to match the shape of the larger one.

Languages which allow functions to operate on arguments of any rank in this manner are *rank polymorphic*.[1] Rank polymorphism originated in mathematical notation for linear algebra and was first introduced as a programming construct in APL [46]. Most rank polymorphic languages are dynamically typed; while work exists on expressing rank polymorphism in statically typed languages [92, 87, 28], the resulting type systems are typically either complicated or leave out features such as parametric polymorphism or higher-order functions, which are difficult to integrate with rank polymorphism.

We investigate a form of rank polymorphism in which functions on their own do not have rank-polymorphic types, but function applications can have **`map`** and **`rep`** operations—which are used for function lifting and argument replication, respectively—inserted implicitly by the compiler as part of the type inference elaboration process. While such an elaboration is less expressive than having rank polymorphism as a first-class construct in the type system, we argue that it is sufficient for the majority of cases. Our approach also has the interesting property that rank polymorphism can be seen as a purely syntactical mechanism for leaving some operations implicit (namely, **`map`** and **`rep`**) at call sites, meaning that the program can also be expressed and understood in a fully explicit manner simply by inserting these operations. This is not the case for most rank-polymorphic languages.

Implicit programming constructs have a long history in several programming languages [48]. Most notably, implicits have been used to express type classes in the Scala programming language [70, 69]. In programming languages with implicit constructs, the compiler transforms a program where some information is missing (in our case **`map`** and **`rep`**) and elaborates the program into one where those implicit constructs have been made explicit. That is, the compiler translates a program with implicit constructs into one with explicit constructs. In some cases, the compiler may have to reject the input program: perhaps too much has been left implicit and the elaboration has become ambiguous.

An implicit programming construct is useful when it fits with the mental model of a programmer and thus allows the programmer to omit certain details without any unexpected surprises. For example, a Hindley-Milner type system is useful because if a program is well-typed we can (a) make an inferred type explicit and the program is still well-typed and (b) remove an explicit type annotation and the system can still infer the now omitted type. We would like similar properties for our system with implicit **`map`** and **`rep`** operations.

In this chapter, we propose Automap, a technique that enables implicit **`map`** and **`rep`** operations to be inferred by a polymorphic type system, using integer linear programming to express and solve the implicit constraints during inference. We present an elaboration algorithm that makes implicit **`map`** and **`rep`** operations explicit and formalize the elaboration with three languages: a *source language* where **`map`** and **`rep`** may be left implicit, an *internal language* where every function application is annotated with a shape annotation that represents the number of **`map`** and **`rep`** operations needed at that site, and, finally, a *target language* where every **`map`** and **`rep`** is explicit. We define a standard dynamic semantics for the target language. We define the meaning of a

[1] Not to be confused with the entirely unrelated, but confusingly similarly named, notion of *higher-rank polymorphism*, which is about nesting type variable quantifiers.

source program via translation to the internal language and then to the target language. However, we allow only source programs for which there is a unique way to insert **map** and **rep**, and we describe how to detect programs where elaboration is ambiguous. We also demonstrate that we accept all programs that would be supported under a type system that does not support Automap.

We implement Automap as an extension of Futhark. Futhark's constraint-based type system supports parametric polymorphism, higher-order functions, and top-level let-generalization; it deviates from classic Hindley-Milner in that it does not support local let-generalization (which would significantly complicate the type system with minimal benefit to the programmer [100, 101]).

We show that Automap is useful by removing "administrative" **map** operations from a collection of real-world benchmarks, where we judge it improves the readability of the code. Even with Automap confined to judicious use (i.e., **map**s are only removed when it improves readability), we are able to remove 54% of all map operations. We found Automap particularly useful for expressing mathematical code; for example, in expressions that implement matrix/vector computations.

While the type inference uses integer linear programming, which in theory has exponential complexity, we empirically show that most of the integer linear programs (ILPs) are small and, moreover, the full Automap technique imposes a modest average type checking overhead of 2.5×.

In summary, the contributions of the chapter are:

- **Motivation:** We motivate the need for Automap with several real-world Futhark programs.
- **Type System and Elaboration:** We present a simple type system for a core language reminiscent of Futhark where **map**s and **rep**s may be omitted and are inferred through a constraint-based elaboration process. We define the meaning of programs through the elaboration into a target language where all **map**s and **rep**s are made explicit.
- **Meta-Theory:** We show that the elaboration satisfies several important properties, including Well-Typedness, Determinism, Disambiguation, Forwards Consistency, and Backwards Consistency as explained in the following section.
- **Implementation:** We implement Automap as an extension of the Futhark compiler using integer linear programming.
- **Evaluation:** We evaluate the usefulness of Automap on a collection of real-world Futhark programs. Specifically, we empirically show that (1) Automap is effective (i.e., we can omit **map** and **rep** operations to make programs more concise and readable), (2) type inference with linear constraint solving is practically feasible, and (3) Automap is able to unambiguously capture those **map** operations that programmers want to leave implicit.

The chapter is organized as follows: Section 4.2 motivates the need for implicit **map**s and **rep**s in Futhark. In Section 4.3 through Section 4.7, we present a formalization of the mechanism, in terms of an Automap elaboration into a well-defined target language. Section 4.8 discusses the implementation of the constraint-based type system, in particular its interaction with other desirable type system features. Section 4.9 evaluates the usefulness of Automap on a collection of real-world Futhark programs. Finally, Section 4.10 discusses future work, Section 4.11 presents related work, and Section 4.12 concludes.

4.2 Motivation

This section explores how data-parallel programming problems are expressed with combinations of **map** and other parallel operations. We illustrate that being explicit about every **map** operation can be tedious and show how AUTOMAP can address this via automatic inference of **map** and **rep** operations. The examples use Futhark notation, but the programming model and syntax is quite similar to other languages in the Haskell or Standard ML tradition. In particular, we consider multidimensional arrays to be simply arrays of arrays, similarly to lists, and for the type of arrays with elements of type τ, we write $[\,]\tau$. When we **map** across an array, we traverse only the outermost dimension. To operate along inner dimensions, we nest **map**s. This differs from languages such as APL, where an array is modeled as a separate *shape vector* and *value vector*, and **map**-like operations apply directly to the elements of the value vector [47, 46].

4.2.1 Idea

Consider adding two vectors element-wise. In Futhark, we would use **map2** to map across two arrays simultaneously (**map2** is similar to Haskell's `zipWith`):[2]

$$\textbf{map2}\ (+)\ [1,\ 2,\ 3]\ [4,\ 5,\ 6].$$

This is awkward for longer expressions. While we can define a helper function `vecadd = map2 (+)`, it would be better if we could simply write [1, 2 ,3] + [4, 5, 6], using infix notation here purely as a syntactical convenience. Since the function (+) has type `int` $\rightarrow$ `int` $\rightarrow$ `int`, while the operands have type $[\,]$`int`, this application is not well-typed. However, it is not difficult to see that if we lift (+) to operate on arrays instead of scalars, the application becomes well-typed—and such lifting is exactly what **map**N does, where N denotes the arity of the function.[3] To operate on multidimensional arrays, **map**N can be nested, such as in **map2** (**map2** (+)) for matrix addition. A type checker could inspect every function application and decide, based on the rank of the operands, how much to lift the function. However, simply inserting a **map** when needed is not sufficient to support desirable programming patterns. In particular, we also want to mix scalar and array operands:

$$[3,\ 4,\ 5] + 1.$$

Given a function **rep** : $\tau \rightarrow [\,]\tau$ that replicates its argument an unspecified number of times,[4] the above expression can be rewritten as

$$[3,\ 4,\ 5] + \textbf{rep}\ 1,$$

and then further as

$$\textbf{map2}\ (+)\ [3,\ 4,\ 5]\ (\textbf{rep}\ 1).$$

AUTOMAP should perform the above rewriting automatically; essentially, AUTOMAP is an algorithm for inserting **map**N and **rep** operations in order to make the program type correct. While AUTOMAP is fundamentally just a syntactical convenience, the impact

[2] In chapter 3, we allowed SOACs to be called with k-ary functions for simplicity—here we use **map2** (instead of **map**) since AUTOMAP transforms source-level Futhark code (which doesn't support SOACs operating on an arbitrary number of arrays at once).

[3] In practice, the Futhark built-in library supports **map**N for $2 \leq N \leq 9$ and are trivially defined in terms of **zip** and **map**.

[4] The issue of how many elements should be replicated will be discussed briefly in Section 4.8.

on readability can be quite dramatic, as we will see in the following section and in Section 4.9. In particular, expressions that are transcriptions of mathematical formulae involving vectors and matrices will otherwise be littered with **map** and **rep** operations; for example, if A is a three-dimensional tensor and c is scalar, it is clearer to write $A \cdot c$ instead of **map** (**map** (**map** $(\cdot\ c)$)) A.

4.2.2 Examples

Consider the following definition of linear interpolation:

$$\textbf{def}\ L\ v\ w\ t = v + (w - v) \cdot t.$$

Because there are no type ascriptions, it would be valid to infer any rank for the parameter types, inserting **map** and **rep** as needed. How should AUTOMAP pick between them? A good strategy (and one that usually aligns with programmer intent and is easy to communicate to the programmer) is to always pick the solution that inserts the fewest operations. The *size* of a solution is equal to the number of inserted **map** and **rep** operations, so we'd like to find minimal solutions. If multiple solutions with the minimum possible size exist, the program is ambiguous and should be rejected. One useful property of this strategy is that a size zero solution is necessarily unique, and so ambiguous cases can always be addressed by the user manually inserting **map**s and **rep**s into the source program. We enshrine this strategy as the first rule of AUTOMAP:

Rule 1: minimize the number of inserted **map**s and **rep**s.

The second rule of the AUTOMAP strategy follows from the fact that it is never necessary to both **map** and **rep** a single application since there are only two possible ways that a rank mismatch can occur: either the argument is under-dimensioned (in which case one or more **rep**s are required) or over-dimensioned (in which case one or more **map**s are required).

Rule 2: a single application may have implicit **map**s or **rep**s but never both.

Note that Rule 1 does not imply Rule 2 (i.e., there are programs where the minimal solution violates Rule 2). Not only do these rules eliminate a lot of potential ambiguity, but they are easy to communicate to the programmer and are easy to understand—important qualities in any implicit programming system.

Returning to the linear interpolation example, following Rules 1 and 2, we infer

$$L\ :\ \texttt{float} \to \texttt{float} \to \texttt{float} \to \texttt{float}.$$

(That is, no implicit **map** or **rep** operations are inferred.) Despite the function itself being scalar, rank polymorphic applications are well-typed through AUTOMAP:

Source Application		**Elaborated Application**
$L\ vs\ ws\ t$	$\longrightarrow$	**map3** $L\ vs\ ws$ (**rep** t)
$L\ vs\ ws\ ts$	$\longrightarrow$	**map3** $L\ vs\ ws\ ts$
$L\ v\ w\ ts$	$\longrightarrow$	**map3** L (**rep** v) (**rep** w) ts,

where v, w,and t have type `float` and vs, ws, and ts have type [ ]`float` (i.e., they are vectors).

As a more complicated example, consider a program `check` for determining whether a square matrix A is an *X-matrix*, meaning it has nonzero elements on its diagonals and zero elements elsewhere. With explicit **map**s, an implementation could be:

```
def outerprod f xs ys = map (λx → map (f x) ys) xs
def bidd A = outerprod (==) (indices A) (indices A)
def xmat A = map2 (map2 (||)) (bidd A) (reverse (bidd A))
def check A = map2 (map2 (==)) (xmat A) (map (map (≠ 0)) A) ▷ flatten ▷ and.
```

The `indices` function accepts an array of size n and produces the array $[0, \ldots, n-1]$, `flatten` eliminates the outer dimension, and the function `and` takes a vector of booleans and returns **true** if all elements are **true**. The code is somewhat noisy, using **map**N eight times. Blindly removing all **map**s (and changing to infix notation), we obtain:

```
def outerprod f xs ys = (λx → f x ys) xs
def bidd A = outerprod (==) (indices A) (indices A)
def xmat A = bidd A || reverse (bidd A)
def check A = xmat A == (A ≠ 0) ▷ flatten ▷ and.
```

By the minimal solution strategy, we infer the type of `outerprod` to be the binary application function, which is not what we intended. As a fix, we put back a single **map**:

```
def outerprod f xs ys = map (λx → f x ys) xs.
```

Also, the body of the `check` function is ambiguous, with the following solutions:

1. **map2** (**map2** (==)) (`xmat` (**rep** A)) (**rep** (**rep** $(A \neq 0)$)) ▷ `flatten` ▷ `and`,
2. **map2** (**map2** (==)) (`xmat` A) (**rep** (**map2** $(\neq)$ A (**rep** 0))) ▷ `flatten` ▷ `and`.

In the first solution, we infer A to be a scalar and in the second to be a vector. Both of these solutions are minimal, but neither of them is actually what we want: we intended A to be a matrix! In this case, we can address the ambiguity by adding a type annotation on the parameter:[5]

```
def check (A : [][]int) = xmat A == (A ≠ 0) ▷ flatten ▷ and.
```

The additional type constraint ensures that only a single Automap elaboration is well-typed. The inferred **rep**s can be simplified away during elaboration, as we will discuss in Section 4.8.

4.2.3 Desired Properties

We consider five requirements that the design of Automap should satisfy. These requirements also ensure a reasonable mental model for programmers. We base the requirements on how type inference works in most programming languages:

- Well-Typedness: If the program, with explicit and implicit **map** and **rep** operations, is well-typed, then the fully elaborated program with only explicit operations is well-typed.

[5]Note that Automap never requires type annotations and `check` could also be disambiguated by inserting a **map** operation.

- DETERMINISM: For any program, the elaboration is unambiguous, or the program is rejected.[6]

- DISAMBIGUATION: If a program is ambiguous, the ambiguity can be resolved by explicit insertion of **`map`** and **`rep`** operations.

- FORWARDS CONSISTENCY: If the programmer inserts an otherwise inferred **`map`** or **`rep`** operation then the resulting program is unambiguous, and its elaboration is semantically equivalent to the elaboration of the original program.

- BACKWARDS CONSISTENCY: If the programmer removes an explicit **`map`** or **`rep`** operation then the resulting program is *either* ambiguous or unambiguous and its elaboration is semantically equivalent to the elaboration of the original program.

The (WELL-TYPEDNESS) and (DETERMINISM) properties are self-explanatory. The (DISAMBIGUATION) property is important because it means that a programmer can always resolve an ambiguous situation by being more explicit. Programmers are familiar with this situation from programming languages with partial type inference where some type annotations may be needed to guide the type checker. Bidirectional type checkers typically require some type annotations, but even languages like Haskell may require type annotations to resolve specific type class instances. The (FORWARDS CONSISTENCY) and (BACKWARDS CONSISTENCY) properties are a bit more involved, but they essentially state that the system is well-behaved when we add inferred or remove explicit **`map`** or **`rep`** operations. The idea is inspired by type annotations in languages with complete type inference like Standard ML [59]. For (FORWARDS CONSISTENCY), if Standard ML has inferred that an expression has a specific type then we as programmers can annotate that specific type and the newly annotated program still type checks. Similarly, for (BACKWARDS CONSISTENCY), if a Standard ML program has a type annotation, and it type checks then we can remove that type annotation and the updated program still type checks. AUTOMAP should have analogous properties for inserting inferred or removing explicit maps replicates, modulo ambiguity.

4.3 Formalization

In the following sections, we formalize the AUTOMAP mechanism for a polymorphic higher-order array language based on a small subset of Futhark/the language introduced in chapter 2. The language is simplistic—it features top-level polymorphic function definitions and a few basic operations on arrays including a built-in **`map`** construct and a **`rep`** construct for modeling arrays containing one replicated value. We further subdivide the language into three different but highly similar languages: (i) the source language, (ii) the internal language, and (iii) the target language. The idea is that the programmer writes programs in the source language; the source language is subsequently transformed into the internal language—which features annotations to keep track of implicit **`map`**s and **`rep`**s—during type checking. Finally, the internal language is elaborated into the target language, which corresponds directly to the source language program except with all **`map`** and **`rep`** operations explicit. Aside from the differences shown in Table 4.1, the three languages are identical and will be fully specified in the next section, but we'll first give an overview of each.

[6] A naive elaboration could reject *all* programs. We show that this is not the case for AUTOMAP.

	Source language	Internal language	Target language
Implicit **map** and **rep**s	✓	✓	✗
Explicit **map** and **rep**s	✓	✓	✓
$\triangle\ (M, R)$ annots.	✗	✓	✗
Example	`sqrt` [1, 2, 3]	`sqrt` [1, 2, 3] $\triangle\ (M, R)$	**map** `sqrt` [1, 2, 3]

Table 4.1: The differences between the source, internal, and target languages. The "Example" row shows how the source expression `sqrt` [1, 2, 3] is transformed into the internal language and then from the internal language into the target language; `sqrt` is an illustrative scalar function that computes the square root of its argument.

Source Language

The source language features implicit **map** and **rep** constructs. This means that inserting **map** and **rep** constructs is always optional. For example, in the "Example" row of Table 4.1, the source expression has an implicit **map**. Note that explicit **map**s and **rep**s are allowed in source expressions as well (and are necessary to address ambiguity).

Internal Language

The internal language allows for so-called *flexible function applications*, which are annotated with rank variables that specify the implicit **map**s and **rep**s. To be more precise, all applications are annotated with objects of the form $\triangle\ (M, R)$; M is a rank variable that represents the number of implicit **map**s and R is a rank variable that represents the number of implicit **rep**s. A constraint-based inference algorithm finds values for all the Ms and Rs in the $\triangle\ (M, R)$ objects of an internal language program, after which it is straightforward to translate an internal language program into a well-typed target program by simply inserting the number of **map** and **rep** constructs that the solved-for Ms and Rs dictate. In the "Example" row of Table 4.1, the source expression `sqrt` [1, 2, 3] is transformed to the internal expression `sqrt` [1, 2, 3] $\triangle\ (M, R)$ by annotating it with an $\triangle\ (M, R)$ object.

Target Language

In the target language—unlike in the source and internal languages—all **map** and **rep** constructs must be explicit. In the "Example" row of Table 4.1, the target expression **map** `sqrt` [1, 2, 3] is obtained from the internal expression `sqrt` [1, 2, 3] $\triangle\ (M, R)$ by using the inference algorithm to find the solution $M \mapsto 1, R \mapsto 0$ and then transforming into the corresponding target expression **map** `sqrt` [1, 2, 3]—one **map** and no **rep**s.

The mechanism we propose will infer **map** and **rep** constructs only at application sites, which in practice covers most needs. However, one may consider extending the mechanism to insert **map** and **rep** constructs also for other constructs such as immediate values and references to variables, which may be beneficial for elaborating programs with branches that have different ranks. In such cases, we leave it up to the programmer to make use of the polymorphic identity function for having Automap infer additional **map** and **rep** constructs.

4.3.1 Preliminaries and Language Grammars

In the following we shall assume a denumerably infinite set V_r of *rank variables*, ranged over by Q, M, and R, a denumerably infinite set V_t of *type variables*, ranged over by α and β, and a denumerably infinite set V_p of *program variables*, ranged over by x, y, z, and f.

$$
\begin{array}{rcll}
S^\oslash & ::= & [\,] & \text{dimension} \\
 & | & S^\oslash\, S^\oslash & \text{concatenation} \\
 & | & \bullet & \text{empty shape}
\end{array}
\qquad
\begin{array}{rcll}
S & ::= & S^\oslash & \text{shapes without rank vars} \\
 & | & [\,]^Q \quad (Q \in V_\mathrm{r}) & \text{rank power} \\
 & | & S\, S & \text{concatenation}
\end{array}
$$

$$
\begin{array}{rcll}
\tau^\oslash & ::= & \texttt{int} & \text{integers} \\
 & | & \tau^\oslash \to \tau^\oslash & \text{functions} \\
 & | & S^\oslash\, \tau^\oslash & \text{arrays} \\
 & | & \alpha \quad (\alpha \in V_\mathrm{t}) & \text{type variable}
\end{array}
\qquad
\begin{array}{rcll}
\tau & ::= & \tau^\oslash & \text{types without rank vars} \\
 & | & \tau \to \tau & \text{functions} \\
 & | & S\, \tau & \text{arrays} \\
\sigma & ::= & \tau & \text{type} \\
 & | & \forall\, \alpha\, .\, \sigma & \text{quantified type}
\end{array}
$$

Figure 4.1: Grammar for closed shapes ($S^\oslash$), shapes (S), closed types ($\tau^\oslash$), types (τ), and type schemes (σ).

Figure 4.1 shows the grammars for shapes, types, and type schemes. Notice that we distinguish between *closed shapes* ($S^\oslash$), that is, shapes that do not contain rank variables, and shapes that may contain rank variables (S). Similarly, we distinguish between *closed types* ($\tau^\oslash$) that do not contain rank variables and types (τ) that may contain rank variables. We also write $\mathrm{frv}(X)$ and $\mathrm{ftv}(X)$ to denote the free rank variables and the free type variables of the object X, respectively. We define equality on shapes and types as the smallest equivalence relation over these sets that includes $\bullet$ as an identity element for shape concatenation (thus, for any shape S, we have $S\,\bullet = \bullet\, S = S$.)

The *rank power* shape $[\,]^Q$—where Q is a rank variable—should be intuitively understood to be a sequence of Q $[\,]$. Correspondingly, when n is a nonnegative integer, $[\,]^n$ is sugar for a sequence of n $[\,]$. That is, $[\,]^0 = \bullet$, and $[\,]^{(n+1)} = [\,]^n[\,]$.

When $\sigma = \forall\, \alpha\, .\, \tau$, we consider α to be *bound* in τ, and we consider type schemes to be equal up to renaming of bound type variables. We write $\forall\, \alpha_1, \ldots, \alpha_k\, .\, \sigma$ as sugar for $\forall\, \alpha_1\, .\, \cdots \forall\, \alpha_k\, .\, \sigma$. We shall also sometimes write $\vec{X}^{(n)}$ to denote a sequence of n objects $X_1, ..., X_n$, and we may sometimes just write $\vec{X}$ if the length of the sequence is clear from the context.

$$
\begin{array}{rcll}
v & ::= & n \quad (n \in \mathbb{Z}) & \text{constant integer} \\
 & | & \lambda x.\, e & \text{function} \\
 & | & [v, \ldots, v] & \text{array} \\
 & | & \texttt{rep}\ v & \text{replicated array} \\
p & ::= & \texttt{def}\ f\ x\ \texttt{=}\ e\ ;\ p & \text{definition} \\
 & | & e & \text{expression}
\end{array}
\qquad
\begin{array}{rcll}
e & ::= & v & \text{value} \\
 & | & x \quad (x \in V_\mathrm{p}) & \text{program variable} \\
 & | & [e, \ldots, e] & \text{arrays} \\
 & | & \texttt{map}\ e\ e & \text{map} \\
 & | & \texttt{rep}\ e & \text{rep} \\
 & | & e\ e\ \triangle\ (M, R) & \text{(annotated) application}
\end{array}
$$

Figure 4.2: Grammar for values (v), expressions (e), and programs (p). In the internal language, applications are annotated with rank specifications (highlighted in blue).

Figure 4.2 shows the grammar for the source, internal, and target languages. Syntactically, all three languages are identical except that applications in the internal language are annotated with rank specifications (highlighted in blue in the figure). The expression $e\ e \triangle (M, R)$ is annotated with the rank variables M and R. After the AUTOMAP system determines integral values for the rank variables M and R, they will be substituted for integral ranks to obtain an expression of the form $e\ e \triangle (n, n)$.

For values of the form $\lambda x.\, e$ and programs of the form $\texttt{def}\ f\ x\ \texttt{=}\ e\ ;\ p$, the program variable x is considered *bound* in e and the program variable f is considered *bound* in p (function definitions may not be recursive). We assume that partial applications of **map** and **rep** are implicitly eta-converted. For instance, we write **map** e to mean $\lambda x.\, \texttt{map}\ e\ x$, if **map** e appears alone. For $n \in \mathbb{N}$, we also write $\texttt{map}^n\ e$ and $\texttt{rep}^n\ e$ as syntactic sugar for a **map** nest and a **rep** nest, respectively:

$$\texttt{map}^0\ e = e, \qquad \texttt{rep}^0\ e = e,$$
$$\texttt{map}^{n+1}\ e = \texttt{map}\ (\texttt{map}^n\ e), \qquad \texttt{rep}^{n+1}\ e = \texttt{rep}\ (\texttt{rep}^n\ e).$$

For the precise semantics that we give in the following section, replicated arrays have unbounded size, which is modeled by representing a replicated array by a single construct **rep** v, which is an array that has the value v at every index.

A *rank substitution* (s_r) maps rank variables to nonnegative integers, a *type substitution* (s_t) maps type variables to types, and a *value substitution* (s_v) maps program variables to values. The effect of applying a substitution s (of any kind) to an object X, written $s(X)$, is to replace (simultaneously) all free (i.e., non-bound) occurrences of variables in X with corresponding values in s (extended to be the identity outside its domain, written $\mathrm{dom}(s)$). When s and t are substitutions and X is some object, we write $(t \circ s)(X)$ to mean $t(s(X))$. Also, we use the notation $s|_D$ to denote the substitution s restricted to the domain D.

4.4 Target Language

We now present the target language. Recall that the only difference between the source language and the target language is that the target language does not feature implicit **map** and **rep** constructs. Instead, all **map** and **rep** constructs are explicit. By virtue of this, the target language is simpler and can be assigned a dynamic semantics—the semantics of the source language are only indirectly defined by first converting into the internal language during type checking and then subsequently into the target language by solving for the rank variables in the $\triangle\ (M, R)$ application annotations. For these reasons, we present the target language first.

Specifically, we define a type system for the language as well as a dynamic semantics, and we demonstrate a type safety property that says that "well-typed expressions do not get stuck". Note that the target language does not track array sizes statically (this is handled by other work [5, 36]) but it does track array ranks. Essentially, the type safety property guarantees that function application always involves applying a function to an argument and that operations that require array arguments indeed are passed arrays at runtime.

Environments (Γ) map program variables to type schemes, and we write $\Gamma, x : \sigma$ to specify that the environment that extends Γ to map x to σ, assuming $x \notin \mathrm{dom}(\Gamma)$. In this section, we shall often write τ to mean $\tau^{\oslash}$, as all types are closed with respect to rank variables and environments shall implicitly contain only *closed type schemes* (i.e.,

$\boxed{\vdash v \,:\, \tau}$

$$\frac{}{\vdash n \,:\, \texttt{int}}\ \text{SV-Int} \qquad \frac{\forall i \in \{1,\ldots,n\}\,.\ \vdash v_i \,:\, \tau}{\vdash [v_1,\ldots,v_n] \,:\, [\,]\,\tau}\ \text{SV-Array} \qquad \frac{\vdash v \,:\, \tau}{\vdash \textbf{rep}\ v \,:\, [\,]\,\tau}\ \text{SV-Rep}$$

$$\frac{x : \tau \vdash e \,:\, \tau'}{\vdash \lambda x.e \,:\, \tau \to \tau'}\ \text{SV-Fun}$$

$\boxed{\Gamma \vdash e : \sigma}$

$$\frac{}{\Gamma, x : \sigma \vdash x \,:\, \sigma}\ \text{S-Var} \qquad \frac{\Gamma \vdash e \,:\, \sigma \qquad \sigma \geq \tau}{\Gamma \vdash e \,:\, \tau}\ \text{S-Inst}$$

$$\frac{\forall i \in \{1,\ldots,n\}\,.\,\Gamma \vdash e_i \,:\, \tau}{\Gamma \vdash [e_1,\ldots,e_n] \,:\, [\,]\,\tau}\ \text{S-Array} \qquad \frac{\vdash v \,:\, \tau}{\Gamma \vdash v \,:\, \tau}\ \text{S-Val} \qquad \frac{\Gamma, x : \tau \vdash e \,:\, \tau'}{\Gamma \vdash \lambda x.\, e \,:\, \tau \to \tau'}\ \text{S-Fun}$$

$$\frac{\Gamma \vdash e_1 \,:\, \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \,:\, \tau_1}{\Gamma \vdash e_1\ e_2 \,:\, \tau_2}\ \text{S-App} \qquad \frac{\Gamma \vdash e_1 \,:\, \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \,:\, [\,]\tau_1}{\Gamma \vdash \textbf{map}\ e_1\ e_2 \,:\, [\,]\tau_2}\ \text{S-Map}$$

$$\frac{\Gamma \vdash e \,:\, \tau}{\Gamma \vdash \textbf{rep}\ e \,:\, [\,]\tau}\ \text{S-Rep}$$

$\boxed{\Gamma \vdash p : \sigma}$

$$\frac{\Gamma, x : \tau \vdash e \,:\, \tau' \qquad \{\vec{\alpha}\} \cap \text{ftv}(\Gamma, \sigma') = \emptyset \qquad \Gamma, f : \forall\vec{\alpha}.\tau \to \tau' \vdash p \,:\, \sigma'}{\Gamma \vdash \textbf{def}\ f\ x = e\ ;\ p \,:\, \sigma'}\ \text{S-Def}$$

Figure 4.3: Target language typing rules.

type schemes containing closed types). We say that a type scheme $\sigma = \forall\vec{\alpha}.\tau'$ *generalizes* a type τ, written $\sigma \geq \tau$, if there exists a type substitution s_t such that $\text{dom}(s_t) = \{\vec{\alpha}\}$ and $s_t(\tau') = \tau$.

Figure 4.3 gives typing rules for the target language. The typing rules allow inferences among sentences of the form $\Gamma \vdash p \,:\, \sigma$, which says that under Γ, p has type scheme σ. Note that an expression is also a program and that we do not show the implicit rule for typing a program that is an expression. The typing rules are closed under type substitution:

Proposition 1 (Typing Closed Under Type Substitution)**.** *If* $\Gamma \vdash p : \sigma$ *then* $s_t(\Gamma) \vdash p : s_t(\sigma)$*, for any type substitution* s_t*.*

Proof Sketch. By induction over the typing derivation (see appendix A for a complete proof). The S-Inst case requires exploiting the fact that $\sigma \leq \tau$ relation is closed under substitution. The S-Def case requires the renaming of the bound variables $\{\vec{\alpha}\}$ to a new set $\{\vec{\alpha'}\}$ to ensure that $\{\vec{\alpha'}\} \cap \text{ftv}(s_t(\Gamma), s_t(\sigma')) = \emptyset$. The remainder of the cases follow straightforwardly by the inductive hypothesis. □

The dynamic semantics is specified as a small-step contextual semantics and the

soundness result is demonstrated using well-known techniques [61, 103]. We define the notions of *contexts* (K) and *redexes* (r) according to the grammars in Figure 4.4 below.

$$
\begin{array}{rcll}
K & ::= & \langle\cdot\rangle \mid K\ e \mid v\ K \mid [v, \ldots, v, K, e, \ldots, e] & \text{Contexts} \\
 & \mid & \texttt{rep}\ K \mid \texttt{map}\ K\ e \mid \texttt{map}\ v\ K & \\
r & ::= & (\lambda x.e)\ v \mid \texttt{map}\ (\lambda x.e)\ [v_1, \cdots, v_n] & \text{Redexes} \\
 & \mid & \texttt{map}\ (\lambda x.e)\ (\texttt{rep}\ v) & \\
 & \mid & \texttt{def}\ f\ x = e\ ;\ p & \text{Program redex}
\end{array}
$$

Figure 4.4: Context and redex grammars for the target language.

When K is a context and e is some expression, we write $K\langle e\rangle$ to denote the expression obtained by filling the hole in the context K with the expression e. Also, when p is a program, we write $K\langle p\rangle$ to denote the program obtained by filling the hole in the context K with the program p. A redex may either be a program or an expression (also considered a program). In particular, during evaluation, top-level functions are first substituted into the program expression before any proper evaluation occurs. Reduction rules for programs (and expressions) are defined as:

$$
\begin{array}{rcll}
\texttt{def}\ f\ x = e\ ;\ p & \rightsquigarrow & p[\lambda x.e/f], & \\
(\lambda x.e)\ v & \rightsquigarrow & e[v/x], & \\
\texttt{map}\ (\lambda x.e)\ [v_1, \cdots, v_n] & \rightsquigarrow & [e[v_1/x], \cdots, e[v_n/x]], & \\
\texttt{map}\ (\lambda x.e)\ (\texttt{rep}\ v) & \rightsquigarrow & \texttt{rep}\ (e[v/x]), & \\
K\langle p\rangle & \rightsquigarrow & K\langle p'\rangle & \text{if } p \rightsquigarrow p' \text{ and } K \neq \langle\cdot\rangle.
\end{array}
$$

Figure 4.5: Reduction rules for the target language.

The proofs for the following propositions are standard, so we relegate full proofs to appendix A and only include proof sketches here. With redexes in hand, we can now express a property saying that any well-typed program p can be decomposed into an evaluation context and a redex:

Proposition 2 (Unique Decomposition)**.** *If $\Gamma \vdash p : \sigma$ then either p is a value or there exists a type scheme σ', a unique expression e, and a unique context K such that $p = K\langle e\rangle$ and $\Gamma \vdash e : \sigma'$ and e is a redex.*

Proof Sketch. By induction over the typing derivation. All the cases are fairly straightforward (and follow by the inductive hypothesis). The C-App case requires casing on whether or not e_1 and e_2 are values. □

Another central property of the type system is that it is closed under value substitution:

Proposition 3 (Typing Closed Under Value Substitution)**.** *If $\Gamma, x : \sigma' \vdash p : \sigma$ and $\vdash v : \sigma'$ then $\Gamma \vdash p[v/x] : \sigma$.*

Proof Sketch. By straightforward induction over the typing derivation. □

The following two properties state progress and preservation, which together express type safety for the target language.

Proposition 4 (Progress)**.** *If $\vdash p : \sigma$ then either p is a value or there exists p' such that $p \rightsquigarrow p'$.*

Proof Sketch. Follows by Proposition 2. □

Proposition 5 (Preservation)**.** *If* $\vdash p : \sigma$ *and* $p \rightsquigarrow p'$ *then* $\vdash p' : \sigma$.

Proof Sketch. By induction over the typing derivation, using Propositions 2 and 3. □

4.5 Internal Language

When the source language is transformed into the internal language during type checking, constraints involving rank variables are generated at each function application site. The rank variables specific to a given application are what populate the $\triangle\ (M, R)$ annotations in the internal language and constraints involving M and R are emitted during type checking. We now precisely define constraints and the internal type system.

4.5.1 Constraints

A *constraint* is a relation defined by the grammar in Figure 4.6. Constraints are generated by the type rules in Figure 4.7 and capture relationships that must hold between types or rank variables—either equality between types ($\tau_1 \doteq \tau_2$) or that one of two rank variables M, R must be assigned zero ($M \curlyvee R$).

$$
\begin{array}{rcll}
c & ::= & \tau_1 \doteq \tau_2 & \text{type equality} \\
 & | & M \curlyvee R & \text{zero-rank disjunction} \\
C & ::= & \emptyset \mid \{c, \ldots\} & \text{set of constraints}
\end{array}
$$

Figure 4.6: Grammar of constraints and constraint sets.

More formally, the constraint $\tau_1 \doteq \tau_2$ is *satisfiable* by a substitution s if $s(\tau_1) = s(\tau_2)$ and both $s(\tau_1)$ and $s(\tau_2)$ are closed (i.e., don't contain any rank variables). The zero-rank disjunction constraint $M \curlyvee R$ is satisfiable by a substitution s if $s(M) = 0$ or $s(R) = 0$. Intuitively, the purpose of the zero-rank disjunction constraint is to enforce Rule 2 of Section 4.2.2—namely that each application can have either implicit **`map`**s or implicit **`rep`**s, but not both.

For a set of constraints C, the substitution s is a *satisfier* of C if every constraint in C is satisfiable by s. Constraint sets also have a notion of equivalence; two constraint sets C, C' are *equivalent*, written $C \simeq C'$, if a substitution s is a satisfier of C if and only if it is a satisfier of C'.

4.5.2 Internal Type System

Figure 4.7 shows the typing rules for constraint-based judgments; the judgment $\Gamma \vdash e :_S \sigma \parallel C$ says that under environment Γ, e has scheme σ with *frame* S when the constraints in C are satisfied. If a frame is the empty shape ($\bullet$) it may be omitted from the judgment; $\Gamma \vdash e : \sigma \parallel C$ is syntactic sugar for $\Gamma \vdash e :_\bullet \sigma \parallel C$. Frames are the extra leading dimensions on a result type generated by preceding implicit **`map`**s; in the relation $\Gamma \vdash e :_S \sigma \parallel C$, frames function to syntactically separate these leading dimensions from the scheme σ, but you can think of the expression e as having type $S\sigma$. Tracking frames separately from the rest of a type is important not only for some

$$\boxed{\Gamma \vdash e :_S \sigma \parallel C}$$

$$\frac{}{\vdash n \,:\, \texttt{int} \parallel \emptyset}\ \text{C-Int} \qquad \frac{\Gamma \vdash e \,:\, \sigma \parallel \emptyset \qquad \sigma \geq \tau}{\Gamma \vdash e \,:\, \tau \parallel \emptyset}\ \text{C-Inst}$$

$$\frac{}{\Gamma, x : \sigma \vdash x \,:\, \sigma \parallel \emptyset}\ \text{C-Var}$$

$$\frac{\forall k \in \{1, \ldots, n\} \,.\, \Gamma \vdash e_k :_{S_k} \tau_k \parallel C_k}{\Gamma \vdash [e_1, e_2, \ldots, e_n] \,:\, [\,]S_1\tau_1 \parallel \{S_1\tau_1 \doteq S_k\tau_k \mid k \in \{2, \ldots, n\}\} \cup C_1 \cup \cdots \cup C_n}\ \text{C-Array}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 :_{S_1} \tau_1 \to \tau_2 \parallel C_1 \qquad \Gamma \vdash e_2 :_{S_2} \tau_3 \parallel C_2 \\ M, R \text{ fresh} \qquad C = \{M \vee R,\ [\,]^M S_1\,\tau_1 \doteq [\,]^R S_2\,\tau_3\}\end{array}}{\Gamma \vdash e_1\, e_2 \vartriangle (M, R) :_{[\,]^M S_1} \tau_2 \parallel C \cup C_1 \cup C_2}\ \text{C-App}$$

$$\frac{\Gamma, x : \tau_1 \vdash e :_S \tau_2 \parallel C}{\Gamma \vdash \lambda x.\, e \,:\, \tau_1 \to S\,\tau_2 \parallel C}\ \text{C-Fun}$$

$$\frac{\Gamma \vdash e_1 :_{S_1} \tau_1 \to \tau_2 \parallel C_1 \qquad \Gamma \vdash e_2 :_{S_2} \tau_3 \parallel C_2}{\Gamma \vdash \textbf{map}\ e_1\ e_2 :_{[\,]S_1} \tau_2 \parallel \{[\,]S_1\tau_1 \doteq S_2\tau_3\} \cup C_1 \cup C_2}\ \text{C-Map}$$

$$\frac{\Gamma \vdash e :_S \tau \parallel C}{\Gamma \vdash \textbf{rep}\ e \,:\, [\,]S\tau \parallel C}\ \text{C-Rep}$$

$$\boxed{\Gamma \vdash p \,:\, \sigma}$$

$$\frac{\begin{array}{c}\Gamma, x : \tau \vdash e \,:\, \tau' \parallel C \qquad s \text{ satisfies } C \qquad \{\vec{\alpha}\} \cap \text{ftv}(s(\Gamma), \sigma') = \emptyset \\ s(\Gamma), f : \forall\vec{\alpha}.s(\tau) \to s(\tau') \vdash p \,:\, \sigma'\end{array}}{s(\Gamma) \vdash \textbf{def}\ f\ x\ \textbf{=}\ s(e)\ ;\ p \,:\, \sigma'}\ \text{C-Def}$$

Figure 4.7: Constraint-based typing rules.

of the proofs of the propositions that follow, but is also critical for the implementation (see Section 4.8.2).

The M, R fresh premise of the C-App introduces two globally fresh rank variables M and R which correspond to the number of implicit **map**s and **rep**s of the application, respectively. The $M \vee R$ constraint encodes that one of M or R must be zero since a flexible function application allows *either* implicit **rep**s or an implicit **map**s, but not both (i.e., it enforces Rule 2 of Section 4.2.2). The constraint $[\,]^M S_1\,\tau_1 \doteq [\,]^R S_2\,\tau_3$ encodes type equality between the parameter of the function (on the LHS) and the argument (on the RHS). The important bits are the tacked on shapes—$[\,]^M$ on the LHS and $[\,]^R$ on the RHS. An under-dimensioned parameter is "lifted" to the correct rank via $[\,]^M$ (corresponding to M implicit **map**s), and an under-dimensioned argument has $[\,]^R$ extra dimensions tacked on (corresponding to R implicit **rep**s). Notice that the resulting frame is $[\,]^M S_1$; each implicit **map** of the current application (the number of which is given by M) increases the dimensionality of the frame (and hence the application). S_1 consists of

any dimensions added by implicit maps of the function e_1. (E.g., $(\lambda x.\, \lambda y.\, x + y)$ [1, 2, 3] has a function type with frame [ ] because an implicit **map** is required to lift + over the vector.)

The C-Map rule can be thought of as a specialization of the C-App rule where $R = 0$ and $M = 1$. The C-Def rule, which types programs, does not return constraint sets. Constraint sets are solved individually for each top-level body. Splitting constraint sets up at the granularity of top-level definitions ensures that **map** and **rep** elaboration is solely a function of the body of the definition, rather than of the program as a whole. As with the source language, we do not show the implicit rule for typing a program that is an expression.

As an example of how a source language expression is transformed into an internal language expression using the rules of Figure 4.7, consider the source language expression `sqrt` [1, 2, 3] introduced in Table 4.1. Via C-Var, C-Array, and C-App we have

$$\begin{aligned}&\texttt{sqrt} : \texttt{int} \to \texttt{int}\\ &\quad \vdash \texttt{sqrt}\ [1,\ 2,\ 3] \triangle (M, R) : [\,]^M\ \texttt{int} \parallel \{M \vee R,\ [\,]^M\ \texttt{int} \doteq [\,]^R\ [\,]\ \texttt{int}\}.\end{aligned}$$

4.6 Rank Analysis

Finding a satisfier s of a constraint set C can always be done in two phases: first, a compatible rank substitution s_r is found by first relaxing the constraint sets generated during type checking into rank-based constraints and then solving them using integer linear programming; the rank substitution is then used to eliminate all rank variables in C. We call this process *rank analysis*. Finally, a satisfying type substitution s_t is found for $s_\mathrm{r}(C)$; the satisfier s can then be formed by the composition $s = s_\mathrm{t} \circ s_\mathrm{r}$. We start by precisely defining a notion of rank for the language.

4.6.1 Rank

The *rank* of a shape or of a type, written $|\cdot|$, is defined in Figure 4.8 and denotes the dimensionality of the shape or type. Because a shape S or a type τ may contain

$$\begin{array}{ll} |[\,]^Q| = Q, & |S\ \tau| = |S| + |\tau|,\\ |[\,]| = 1, & |\alpha| = \overline{\alpha},\\ |S_1\ S_2| = |S_1| + |S_2|, & |\tau_1 \to \tau_2| = 0,\\ |\bullet| = 0, & |\texttt{int}| = 0. \end{array}$$

Figure 4.8: Definition of rank; $\overline{\alpha}$ is the associated rank variable of α.

(rank and type) variables, $|S|$ and $|\tau|$ will be linear sums of integers and rank variables. For each type variable, an *associated rank variable* that represents the rank of the given type variable is introduced. Associated rank variables are distinguished by an overline: $V_{|\mathrm{t}|} := \{\overline{\alpha} \mid \alpha \in V_\mathrm{t}\}$ is the set of associated rank variables for type variables. As an example, the rank of $[\,]^n\,\texttt{int}$ is given by $|[\,]^n\,\texttt{int}| = |[\,]^n| + |\texttt{int}| = n + 0 = n$. Another example—this time including a rank and a type variable—is $|[\,]^M \alpha| = M + \overline{\alpha}$.

4.6.2 Rank Constraints

This section introduces a rank-based relaxation of a constraint set C, used to solve for the rank variables in C. The type equality constraint of Figure 4.6 can be relaxed to *rank equality* constraints via application of $|\cdot|$, which only enforce the ranks of the types either side of the $\doteq$ to be equal; Figure 4.9 shows the constraint grammar extended with rank equality constraints.

$$\begin{array}{rcll} c & ::= & \ldots & \\ & | & |\tau_1| \doteq |\tau_2| & \text{rank equality} \end{array}$$

Figure 4.9: Constraint grammar extended with the rank equality constraint.

The rank constraint $|\tau_1| \doteq |\tau_2|$ is satisfied by a rank substitution s_r if $s_\mathrm{r}(|\tau_1|) = s_\mathrm{r}(|\tau_2|)$ and neither $s_\mathrm{r}(|\tau_1|)$ nor $s_\mathrm{r}(|\tau_2|)$ contain any rank variables (i.e., they're both closed). We also extend $|\cdot|$ to work over constraints:

$$\begin{aligned} |\tau_1 \doteq \tau_2| &= |\tau_1| \doteq |\tau_2|, \\ |M \vee R| &= M \vee R. \end{aligned}$$

Intuitively, $|\cdot|$ is the identity operation on zero-rank disjunction constraints ($M \vee R$) because these constraints are already over rank variables. Finally, if C is a set of constraints, $|C|$ is the corresponding set of corresponding rank constraints: $|C| = \{|c| \mid c \in C\}$.

As an example, consider the constraint set $C = \{M \vee R,\ [\,]^M \texttt{ int} \doteq [\,]^R [\,]\texttt{int}\}$ from the typing of `sqrt` $[1, 2, 3] \triangle (M, R)$ just before Section 4.6 on the previous page. The corresponding rank constraint set is given by $|C| = \{M \vee R, |[\,]^M \texttt{ int} \doteq [\,]^R [\,]\texttt{int}|\} = \{M \vee R, M \doteq R + 1\}$, which is satisfied by $s_\mathrm{r} = [M \mapsto 1, R \mapsto 0]$. This solution indicates that there is an implicit map (because M is assigned a value of 1) and aligns with the target language elaboration shown in Table 4.1.

4.6.3 Size and Ambiguity

There may be many different rank substitutions that can satisfy the rank constraint set $|C|$. To stratify them, we quantify them via a notion of size. The *size* of a rank substitution s_r relative to a rank constraint set $|C|$ is defined as

$$\mathrm{size}(s_\mathrm{r}, |C|) = \sum_{Q \in \mathrm{frv}(|C|)} s_\mathrm{r}(Q).$$

Note that $\mathrm{size}(s_\mathrm{r}, |C|)$ doesn't count the rank of any associated rank variables for type variables (i.e., variables of the form $\overline{\alpha}$) because $\mathrm{frv}(|C|)$ doesn't include associated rank variables. Since all the rank variables in $|C|$ originate from flexible function-application annotations (i.e. $\triangle\ (M, R,)$), $\mathrm{size}(s_\mathrm{r}, |C|)$ expresses the total number of **`map`**s and **`rep`**s that s_r assigns. The intent here is that smaller sized solutions are preferable—in accordance with Rule 1—and the size function provides a means of ranking solutions.

With a notion of size, we can be more specific about the ambiguity of a rank constraint set. We say that $|C|$ is *ambiguous at size k* if there exist satisfying rank substitutions $s_\mathrm{r}, s'_\mathrm{r}$, of $|C|$ with $\mathrm{size}(s_\mathrm{r}, |C|) = \mathrm{size}(s'_\mathrm{r}, |C|) = k$ and $s_\mathrm{r} \neq s'_\mathrm{r}$. Otherwise, we say that $|C|$ is *unambiguous at size k*; if $|C|$ is unambiguous at size k, it has at most one satisfier with size k.

For an example, consider the constraint set $\{M \vee R, [\,]^M \alpha \doteq [\,]^R [\,][\,]\texttt{int}\}$. The corresponding rank constraint set is given by $\{M \vee R, M + \overline{\alpha} \doteq R + 2\}$; one possible satisfier with size 1 is $s_r = [M \mapsto 1, R \mapsto 0, \overline{\alpha} \mapsto 1]$, but another is $s_r = [M \mapsto 0, R \mapsto 1, \overline{\alpha} \mapsto 3]$ and hence $|C|$ is ambiguous at size 1. On the other hand, $|C|$ is unambiguous at size 0 and $s_r = [M \mapsto 0, R \mapsto 0, \overline{\alpha} \mapsto 2]$ is a satisfier. (All size 0 solutions are always unique since they correspond to no implicit **map**s and **rep**s).

4.6.4 Rank Constraint Set Solving using Integer Linear Programming

To find a satisfying rank substitution s_r for a rank constraint set $|C|$, we construct an integer linear program (ILP) from the constraints of $|C|$. The rank variables of $|C|$ constitute unknown non-negative integral variables in the ILP and the constraints themselves are linearized and then added as constraints to the ILP. The ILP is constructed such that any solution of the ILP (i.e., a mapping from rank variables to integer ranks) constitutes a satisfying substitution s_r of $|C|$. Table 4.2 shows how rank constraints are converted into constraints of the ILP.

Rank constraint	ILP constraint
$\lvert\tau_1\rvert \doteq \lvert\tau_2\rvert$	$\lvert\tau_1\rvert = \lvert\tau_2\rvert$
	$M \leq U \cdot b_M$
$M \vee R$	$R \leq U \cdot b_R$
	$b_M + b_R \leq 1$

Table 4.2: Mapping of rank constraints to ILP constraints.
U is a constant upper bound on M and R.

Rank equality constraints ($|\tau_1| \doteq |\tau_2|$) are already linear and added to the ILP directly (with $\doteq$ simply replaced by standard equality). Recall that shape rank disjunction constraints ($M \vee R$) are satisfied by a rank substiution s_r when $s_r(M) = 0$ or $s_r(R) = 0$. To encode this constraint in the ILP, binary variables b_M and b_R are introduced where $M = 0$ if $b_M = 0$ and, analogously, $R = 0$ if $b_R = 0$. The relationship between b_M and M is enforced by the constraint $M \leq U \cdot b_M$ where U is a constant global upper bound on all rank variables (and analogously for b_R and R). The constraint $b_M + b_R \leq 1$ then enforces that one of b_M, b_R is 0 and hence that either M or R is 0.

In correspondence with our desire to insert the minimal number of **map**s and **rep**s necessary (Rule 1), the objective of the ILP is to minimize the sum of all the rank variables in flexible function application annotations ($\triangle\ (M, R)$) of an expression; this corresponds to minimizing size($s_r, |C|$) where s_r is the solution of the ILP. To illustrate, Figure 4.10 shows how the rank ILP is generated for the expression `sum (length` xss), where xss is a matrix.

A rank constraint set $|C|$ is ambiguous at size k where k is the minimal solution size if and only if its corresponding ILP has multiple solutions; ambiguity of constraint set at the minimal size can therefore be detected by enumerating two distinct solutions to the ILP with the same minimal size; this can be accomplished by adding constraints to enforce a second solution to be distinct from the first but with the same size.

$$\dfrac{\dfrac{\Gamma \vdash \texttt{length} : [\,]\alpha \to \texttt{int} \parallel \emptyset}{\Gamma \vdash xss : [\,][\,]\texttt{int} \parallel \emptyset \qquad C_1 = \{M_1 \vee R_1,\ [\,]^{M_1}[\,]\alpha \doteq [\,]^{R_1}[\,][\,]\texttt{int}\}}{\Gamma \vdash \texttt{length}\ xss \vartriangle (M_1, R_1) : [\,]^{M_1}\ \texttt{int} \parallel C_1}\ \text{C-App} \qquad \Gamma \vdash \texttt{sum} : [\,]\texttt{int} \to \texttt{int} \parallel \emptyset \qquad C_2 = \{M_2 \vee R_2,\ [\,]^{M_2}[\,]\texttt{int} \doteq [\,]^{R_2}[\,]^{M_1}\texttt{int}\} \cup C_1}{\Gamma \vdash \texttt{sum}\,(\texttt{length}\ xss) \vartriangle (M_2, R_2) : [\,]^{M_2}[\,]^{M_1}\ \texttt{int} \parallel C_2}\ \text{C-App}$$

(a) Typing derivation for `sum (length xss)`. It is assumed that `sum` : $[\,]\texttt{int} \to \texttt{int} \in \Gamma$, `length` : $\forall\alpha.[\,]\alpha \to \texttt{int} \in \Gamma$, and $xss : [\,][\,]\texttt{int} \in \Gamma$; the derivations for these terms (i.e., applications of C-Var and C-Inst) have be omitted for brevity.

$$|C_2| = \left\{ \begin{array}{l} M_1 \veebar R_1, \\ M_1 + 1 + \overline{\alpha} \doteq R_1 + 2, \\ M_2 \veebar R_2, \\ M_2 + 1 \doteq R_2 + M_1 \end{array} \right\}$$

(b) The associated rank constraint set of C_2 from Figure 4.10a.

$$\begin{array}{rrl} \text{minimize} & R_1 + M_1 + R_2 + M_2 & \\ \text{subject to} & & \\ & M_1 & \leq U \cdot b_{M_1} \\ & R_1 & \leq U \cdot b_{R_1} \\ & b_{M_1} + b_{R_1} & \leq 1 \\ & M_1 + 1 + \overline{\alpha} & = R_1 + 2 \\ & M_2 & \leq U \cdot b_{M_2} \\ & R_2 & \leq U \cdot b_{R_2} \\ & b_{M_2} + b_{R_2} & \leq 1 \\ & M_2 + 1 & = R_2 + M_1 \end{array}$$

(c) The ILP corresponding to $|C_2|$ in Figure 4.10b. All variables are additionally constrained to be non-negative (e.g., $0 \leq \overline{R_1}$); these constraints are omitted for brevity.

Figure 4.10: ILP generation for `sum (length xss)`. One solution sets $b_{M_1} \mapsto 1$, $M_1 \mapsto 1$ and all other variables to 0, corresponding to the elaboration `sum (`**`map`**` length xss)`. This ILP is ambiguous, the other size 1 solution sets $\overline{\alpha} \mapsto 1$, $b_{R_2} \mapsto 1$, $R_2 \mapsto 1$ and all other variables to 0, corresponding to the elaborated expression `sum (`**`rep`**` (length xss))`. The $\overline{\alpha}$ assignment can be understood as the minimal rank of the type that a satisfying type substitution s_t of $s_\text{r}(C)$ must assign to α; in the first elaboration, $s_\text{t}(\alpha) = \texttt{int}$, in the second, $s_\text{t}(\alpha) = [\,]\texttt{int}$.

4.6.5 Constraint Set Solving

Our constraint solving algorithm Solve is shown in Figure 4.11 and uses rank analysis and conventional type unification to find a satisfier s of a constraint set C. If Solve(C) succeeds, it returns a substitution s such that s satisfies C and hence illustrates how to realize the C-Def rule of Figure 4.7 into a practical type system (by providing a means of finding the satisfier s in its premises). Note that Unify on line 11 of the procedure is standard structural type unification as in [58], except it ignores the now-trivial zero-rank disjunction constraints (which only contain integers, not rank variables, after s_r is applied).

Given a constraint set C and a satisfying rank substitution s_r of $|C|$, the constraint set $s_\text{r}(C)$ is a closure of C; that is, all of its rank variables have been instantiated with integral ranks. Hence, the constraint set $s_\text{r}(C)$ on line 10 of the Solve procedure is closed.

If C is satisfiable, so must be $s_\text{r}(C)$ and a satisfying type substitution s_t can be found via standard syntactic type unification. This means that the substitution $s_\text{t} \circ s_\text{r}$—where s_t is returned by Unify—is a satisfier of the original constraint set C. Propositions 6 and 7 show that every satisfier of C can be found in this manner; their full proofs can be found in appendix A.

Proposition 6. *If s satisfies C, there exists a rank substitution s_r that satisfies $|C|$ and there exists a closed type substitution s_t such that $s|_{\text{ftv}(C) \cup \text{frv}(C)} = s_\text{t} \circ s_\text{r}$.*

Proof Sketch. Follows by constructing a rank substitution s_r defined by $s_\text{r}(Q) = s(Q)$ and $s_\text{r}(\overline{\alpha}) = |s(\alpha)|^\oslash$ (where $|\cdot|^\oslash$ works like $|\cdot|$ except it assigns type variables the rank 0 and expects a closed argument) and a type substitution $s_\text{t} = s|_{\text{ftv}(C)}$. □

Procedure SOLVE

input : A constraint set C.
output: A satisfying substitution s.

```
1  |C| ← construct the associated rank constraint set from C; // Section 4.6.2
2  I ← construct the corresponding ILP from |C|;             // Section 4.6.4
3  s_r ← solve I using an ILP solver;
4  if s_r then
5  |  I' ← add constraints to I to ban solution s_r and enforce a size of size(s_r, |C|);
6  |  s'_r ← solve I' using an ILP solver;
7  |  if s'_r then
8  |  |  return ⊥;                                  // |C| is ambiguous; fail
9  |  else
10 |  |  return UNIFY(s_r(C)) ∘ s_r
11 else
12 |  return ⊥
```

Figure 4.11: Our algorithm for constraint solving.

Proposition 7. *If C is satisfiable and s_r satisfies $|C|$ then there is a closed type substitution s_t such that the substitution $s = s_t \circ s_r$ satisfies C.*

Proof Sketch. Since C is satisfiable, a satisfier s for it exists; s_t is constructed using both s_r and s: $s_t(\alpha) = [\,]^{s_r(\overline{\alpha})}\text{basetype}(s(\alpha))$ where $\text{basetype}(S\tau) = \text{basetype}(\tau)$ and is the identity otherwise (that is, basetype strips all array dimensions). □

Note that Propositions 6 and 7 only hold because our language does not track sizes. In a more general setting with sizes, given a satisfiable constraint set C and a rank substitution s_r that satisfies $|C|$, $s_r(C)$ may not be satisfiable due to size mismatches.

4.7 Transformation to the Target Language

$$
\begin{array}{lcllcl}
\text{AM}(n) & = & n, & \text{AM}([e_1, \ldots, e_m]) & = & [\text{AM}(e_1), \ldots, \text{AM}(e_m)], \\
\text{AM}(v) & = & v, & \text{AM}(e_1\ e_2\ \triangle\ (n_\text{M}, n_\text{R})) & = & \texttt{map}^{n_\text{M}}\ \text{AM}(e_1)\ (\texttt{rep}^{n_\text{R}}\ \text{AM}(e_2)), \\
\text{AM}(x) & = & x, & \text{AM}(\texttt{map}\ e_1\ e_2) & = & \texttt{map}\ \text{AM}(e_1)\ \text{AM}(e_2), \\
\text{AM}(\lambda x.\, e) & = & \lambda x.\, \text{AM}(e), & \text{AM}(\texttt{rep}\ e) & = & \texttt{rep}\ \text{AM}(e), \\
 & & & \text{AM}(\texttt{def}\ f\ x = e\ ;\ p) & = & \texttt{def}\ f\ x = \text{AM}(e)\ ;\ \text{AM}(p).
\end{array}
$$

Figure 4.12: The AM transformation from the internal language to the target language; note that $n_\text{M}, n_\text{R} \in \mathbb{N}$.

C-DEF dispatches the constraint sets of each top-level definition and applies a satisfying substitution to each top-level body. This replaces the rank variables of the flexible function applications ($\triangle\ (M, R)$) in each top-level body with integral ranks . Hence, each internal program typed via the C-DEF judgment is closed and only contains integral ranks at the flexible function annotation sites.

The job of the AM transformation is to elaborate these integral annotations ($\triangle\ (n_\text{M}, n_\text{R})$)—corresponding to implicit numbers of **map**s and **rep**s—into explicit **map**s

and **rep**s. That is, AM converts programs from the internal language into the target language. It is defined in Figure 4.12.

Recall that the $R \vee M$ constraint in C-App forces one of R or M to be 0; hence at least n_M or n_R in the figure must be 0 and the expansion of $e_1\ e_2 \vartriangle (n_M, n_R)$ will result in either a **map** or a **rep** (or neither) but never both (in accordance with Rule 2 of Section 4.2.2).

Returning to the example from Table 4.1 (`sqrt` [1, 2, 3] $\vartriangle (M, R)$), in Section 4.6.2 we found that the rank substitution $s_r = [M \mapsto 1, R \mapsto 0]$ satisfied its rank constraint set (this solution is also minimal and would be the same one returned by a solution to the corresponding ILP formulation). Since the constraint set doesn't feature any type variables, the Unify call in Figure 4.11 will return an empty substitution and hence the satisfier for the constraint set is simply the substitution $s = s_r$.[7] The substitution is applied to the expression to obtain `sqrt` [1, 2, 3] $\vartriangle (1, 0)$, which is the form the expression would take in the conclusion of the C-Def rule. To now convert the internal expression `sqrt` [1, 2, 3] $\vartriangle (1, 0)$ to the target language, we simply apply the AM transformation:

$$\begin{aligned} \text{AM}(\texttt{sqrt}\ [1,\ 2,\ 3] \vartriangle (1,0)) &= \mathbf{map}^1\ \text{AM}(\texttt{sqrt})\ (\mathbf{rep}^0\ \text{AM}([1,\ 2,\ 3])) \\ &= \mathbf{map}\ \texttt{sqrt}\ [1,\ 2,\ 3]. \end{aligned}$$

4.7.1 Well-Typedness

Proposition 9 shows the Well-Typedness property from Section 4.2.3. That is, if p is a well-typed program in the internal language then AM(p) is a well-typed program in the target language. First, an equivalent property is needed for expressions.

Proposition 8 (Well-Typedness for Expressions)**.** *If* $\Gamma \vdash e :_S \sigma \parallel C$ *and* s *is a satisfier of* C*, then* $s(\Gamma) \vdash \text{AM}(s(e)) : s(S\ \sigma)$

Proof Sketch. Follows by induction on over the typing derivation. The core part of the proof is the C-App case (i.e., for expressions of the form $e_1\ e_2 \vartriangle (M, R)$), where we exploit the fact that s is a satisfier of C and hence can conclude that either $M = 0$ or $R = 0$ and case on each of these possibilities. □

Proposition 9 (Well-Typedness)**.** *If* $\Gamma \vdash p : \sigma$ *then* $\Gamma \vdash \text{AM}(p) : \sigma$.

Proof Sketch. Straightforward induction over the typing derivation, using Proposition 8. □

Appendix A features proofs for the above propositions.

4.7.2 Backwards Consistency

In this section, we show that rank analysis correctly reconstructs a removed explicit **map** or **rep** from an expression. To talk about specific **map**s or **rep** in a program, we introduce internal language contexts with the following grammar:

$$\begin{aligned} \mathcal{K} \quad ::= \quad & \langle\cdot\rangle \mid \mathcal{K}\ e \vartriangle (M,R) \mid e\ \mathcal{K} \vartriangle (M,R) \mid [e,\ldots,e,\mathcal{K},e,\ldots,e] \\ \mid \quad & \mathbf{rep}\ \mathcal{K} \mid \mathbf{map}\ \mathcal{K}\ e \mid \mathbf{map}\ e\ \mathcal{K} \end{aligned}$$

Figure 4.13: Internal language contexts grammar.

[7] Since expressions only contain rank variables—and not type variables—this technically doesn't matter and applying either $s = s_t \circ s_r$ (where s_t is returned by Unify) or just s_r yields the same result.

If e is an explicit **map** or **rep** in some context $\mathcal{K}$, the relation $\mathcal{K}\langle e\rangle \prec_{\text{rem}} \mathcal{K}\langle e'\rangle$ says that $\mathcal{K}\langle e'\rangle$ is semantically equivalent equivalent expression to $\mathcal{K}\langle e\rangle$, shown in Figure 4.14 below.

$$\boxed{\mathcal{K}\langle e\rangle \prec_{\text{rem}} \mathcal{K}\langle e'\rangle}$$

$$\frac{M, R \text{ fresh}}{\mathcal{K}\langle \textbf{map}\ e_1\ e_2\rangle \prec_{\text{rem}} \mathcal{K}\langle e_1\ e_2 \triangle (M, R)\rangle}\ \textsc{Rem-Map} \qquad \frac{M, R \text{ fresh}}{\mathcal{K}\langle \textbf{rep}\ e\rangle \prec_{\text{rem}} \mathcal{K}\langle (\lambda x.\, x)\ e \triangle (M, R)\rangle}\ \textsc{Rem-Rep}$$

Figure 4.14: The removal relation.

The following proposition says that we can remove a **map** or a **rep** from a well-typed expression and obtain a well-typed expression. Additionally, their constraint sets are in correspondence in the sense that a rank substitution that appropriately assigns the newly-introduced rank variables (in the flexible function application that replaced the **map** or **rep**) can be applied to make the two constraint sets equivalent.

Proposition 10 (Removal Well-Typedness)**.** *If* $\mathcal{K}\langle e\rangle \prec_{rem} \mathcal{K}\langle e'\rangle$ *and* $\Gamma \vdash \mathcal{K}\langle e\rangle :_S \sigma \parallel C$ *then there exists* S', σ', *and* C' *such that*

(a) $\Gamma \vdash \mathcal{K}\langle e'\rangle :_{S'} \sigma' \parallel C'$.

(b) If s_{r} *satisfies* $|C|$, *then there exists* $\hat{s}'_{\text{r}}$ *such that* $s_{\text{r}} \circ \hat{s}'_{\text{r}}$ *satisfies* $|C'|$.

(c) $s_{\text{r}}(C) \simeq (s_{\text{r}} \circ \hat{s}'_{\text{r}})(C')$.

Proof Sketch. By induction over $\mathcal{K}$. The meat of the proof is the case where $\mathcal{K} = \langle\cdot\rangle$, which proceeds by casing on the removal relation (Rem-Map or Rem-Rep). For both cases, part (a) follows by applying C-App appropriately, (b) by constructing $\hat{s}'_{\text{r}}$ to assign the introduced rank variables from the new flexible function application to correctly reconstruct either the removed **map** or **rep**, and (c) by the fact that the introduced rank disjunction constraints ($M \veebar R$) become trivial after application of $\hat{s}'_{\text{r}}$ (wherein they reduce to either $1 \veebar 0$ or $0 \veebar 1$). The remaining cases for $\mathcal{K}$ follow by straightforward application of the inductive hypothesis. □

Proposition 11 states the Backwards Consistency property of Section 4.2.3— namely that when a **map** or **rep** is removed, and the resulting program is unambiguous, then the elaboration of that program is equivalent to the original program with the explicit **map** or **rep**.

Proposition 11 (Backwards Consistency)**.** *If* $\mathcal{K}\langle e\rangle \prec_{rem} \mathcal{K}\langle e'\rangle$, $\Gamma \vdash \mathcal{K}\langle e\rangle :_S \sigma \parallel C$, *and* s_{r} *is unambiguous at size* k *for* $|C|$, *then there exists* S', σ', C', s'_{r} *such that* $\Gamma \vdash \mathcal{K}\langle e'\rangle :_{S'} \sigma' \parallel C'$ *and* s'_{r} *is unambiguous at size* $k+1$ *for* $|C'|$ *with* $\text{AM}(s_{\text{r}}(\mathcal{K}\langle e\rangle)) = \text{AM}(s'_{\text{r}}(\mathcal{K}\langle e'\rangle))$.

Proof Sketch. By induction over $\mathcal{K}$; the $\mathcal{K} = \langle\cdot\rangle$ case is the most interesting, and we proceed by casing on the removal relation. In each case, we exploit the unambiguity of s'_{r} to conclude that $s'_{\text{r}} = s_{\text{r}} \circ \hat{s}'_{\text{r}}$ where $\hat{s}_{\text{r}}$ is as constructed in the relevant cases of the proof of Proposition 10, from which we obtain the required equality by the definition of AM. □

It should be noted that as stated proposition 11 isn't as strong as we'd like— instead of S', σ', C', s'_{r} being existentially quantified in the above proposition, they should be universally quantified. While we conjecture this to be true, we leave its proof to future work.

4.7.3 Forwards Consistency

In this section, we show that an inferred **map** or **rep** can always be made explicit. In analog to $\prec_{\mathrm{rem}}$, we introduce a new relation $\mathcal{K}\langle e\rangle \succ_{\mathrm{add}} \mathcal{K}\langle e'\rangle$ that says that $\mathcal{K}\langle e'\rangle$ is a semantically equivalent equivalent expression to $\mathcal{K}\langle e\rangle$, shown in Figure 4.15 below. Note that, unlike $\prec_{\mathrm{rem}}$, $\succ_{\mathrm{add}}$ only operates on closed expressions (i.e., expressions post rank analysis).

$$\boxed{\mathcal{K}\langle e\rangle \succ_{\mathrm{add}} \mathcal{K}\langle e'\rangle}$$

$$\frac{}{\mathcal{K}\langle e_1\ e_2 \vartriangle (n_{\mathrm{M}}+1, n_{\mathrm{R}})\rangle \succ_{\mathrm{add}} \mathcal{K}\langle (\mathbf{map}\ e_1)\ e_2 \vartriangle (n_{\mathrm{M}}, n_{\mathrm{R}})\rangle}\ \textsc{Add-Map}$$

$$\frac{}{\mathcal{K}\langle e_1\ e_2 \vartriangle (n_{\mathrm{M}}, n_{\mathrm{R}}+1)\rangle \succ_{\mathrm{add}} \mathcal{K}\langle e_1\ (\mathbf{rep}\ e_2) \vartriangle (n_{\mathrm{M}}, n_{\mathrm{R}})\rangle}\ \textsc{Add-Rep}$$

Figure 4.15: The add relation.

Proposition 12 (Forwards Consistency)**.** *If* $\mathcal{K}\langle e\rangle \succ_{add} \mathcal{K}\langle e'\rangle$ *then* $\mathrm{AM}(\mathcal{K}\langle e\rangle) = \mathrm{AM}(\mathcal{K}\langle e'\rangle)$.

Proof Sketch. Straightforward induction over $\mathcal{K}$. □

4.8 Implementation

We have implemented Automap in a compiler for the functional array language Futhark. The implementation consists of four phases, shown in Figure 4.16 below.

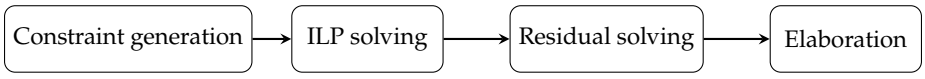


Figure 4.16: The four phases of Automap in the implementation.

4.8.1 Constraint Generation

This part of the implementation largely follows the structure given in Section 4.5: each top-level definition is type checked individually, application nodes are annotated with the equivalent of the $\vartriangle$ $(M, R,)$ annotations in Section 4.5,[8] and constraints are collected.

[8] Actually, in the implementation, applications are annotated with triples of the form $\vartriangle(M, R, S)$ where the third component is the frame, which circumvents the problem that the Futhark compiler does not have the facility to track the frame at the type level.

4.8.2 ILP Solving

The implementation mostly follows Section 4.6.5, utilizing the GNU Linear Programming Kit[9] to solve the ILPs. One notable difference from Section 4.6.5 is that the ILP is modified to avoid counting *induced* **rep**s, which are **rep**s that are required as a consequence of previous **map**s. For example, **map** $(\lambda y.\, xs \cdot y)\ ys$ (noting the existent **map** is an explicit one) can be elaborated as

$$\textbf{map}\ (\lambda y.\, \textbf{map}\ (\cdot)\ xs\ (\textbf{rep}\ y))\ ys \quad \text{or} \quad \textbf{map}\ (\lambda y.\, \textbf{map}\ (\cdot)\ xs\ y)\ (\textbf{rep}\ ys),$$

where xs and ys are vectors. In the first example, the **rep** is induced by the inner **map**. In the second, the **rep** is required because of the outer **map**, which is explicit and hence the **rep** is non-induced. Both of these elaborations are associated with size 2 rank substitutions and hence the type checker would reject **map** $(\lambda y.\, xs \cdot y)\ ys$ as ambiguous. However, induced reps can always be removed by ***rep** fusion*; that is, pushing the **rep** down the **map** nest. Doing so for the first example yields

$$\textbf{map}\ (\lambda y.\, \textbf{map}\ (\lambda x.\, x\ \cdot\ y)\ xs)\ ys,$$

which has size 1 and is the only elaboration of **map** $(\lambda y.\, xs \cdot y)\ ys$ with size 1 and hence is unambiguous. Disambiguation by **rep** fusion in this manner tends to also better align with programmer intent by virtue of corresponding with a smaller solution. Because frames are the concatenation of all previous map shapes in a series of applications, at each application, the number of non-induced **rep**s can be recovered by subtracting the rank of the current frame from the total number of **rep**s. More precisely, if e_1 is a function with frame S_1, then the non-induced **rep**s for the flexible function application $e_1\ e_2 \triangle (M, R)$ is given by $\max(0, |R| - |S_1|)$. In this scheme, each application contributes $M + \max(0, |R| - |S_1|)$ to the ILP's objective; the linearization of $\max(0, |R| - |S_1|)$ adds a few additional constraints and two additional ILP variables. We found only counting non-induced **rep**s in the ILP objective to greatly diminish the frequency of ambiguity in practice (although it raises issues in other cases, see Section 4.10).

To check for ambiguity, we discriminate on the binary variables used to enforce the $M \vee R$ constraints; see section 4.10.3 for more details.

4.8.3 Residual Solving

The interaction of Automap with other nonstandard type system features is challenging. In particular, Futhark supports *size-dependent types*, where array sizes are tracked in the type system [5], and functions can impose size constraints. For example, `zip` requires that the input arrays have the same length. Automap is completely oblivious to size types, and Section 4.9 contains an example where the elaborated program is ill-typed when considering sizes.

Concretely, we first infer ground types via the first two phases of Figure 4.16, where the specific sizes of arrays are not tracked, but only their rank. Then, in the residual solving phase, we perform size-type inference on the resulting elaborated program as in [5], using the Automap-inferred types as a starting point. This stratification is largely for simplicity, as the size-dependent type system has features that are difficult to integrate in our constraint language. For example, when `if` branches return arrays of differing size, a new "existential size" is implicitly created and used to assign a type to the expression. However, such a size mismatch can only be detected after the array

[9] https://www.gnu.org/software/glpk/glpk.html

```
def main [nK][nX]
         (kx: [nK]f32) (ky: [nK]f32) (kz: [nK]f32)
         (x: [nX]f32) (y: [nX]f32) (z: [nX]f32)
         (phiR: [nK]f32) (phiI: [nK]f32)
       : ([nX]f32, [nX]f32) =
  let phiM = map2 (λr i → r*r + i*i) phiR phiI
  let as = map3 (λx_e y_e z_e →
             map (2*pi*)
               (map3 (λkx_e ky_e kz_e →
                  kx_e*x_e + ky_e*y_e + kz_e*z_e)
                 kx ky kz))
            x y z
  let qr = map (λa → sum(map2 (*) phiM (map cos a))) as
  let qi = map (λa → sum(map2 (*) phiM (map sin a))) as
  in (qr, qi)
```

(a) `mri-q` with explicit maps.

```
def main [nK][nX]
         (kx: [nK]f32) (ky: [nK]f32) (kz: [nK]f32)
         (x: [nX]f32) (y: [nX]f32) (z: [nX]f32)
         (phiR: [nK]f32) (phiI: [nK]f32)
       : ([nX]f32, [nX]f32) =
  let phiM = phiR*phiR + phiI*phiI
  let as = 2*pi*(kx*transpose (rep x)
           + ky*transpose (rep y)
           + kz*transpose (rep z))
  let qr = sum (cos as * phiM)
  let qi = sum (sin as * phiM)
  in (qr, qi)
```

(b) `mri-q` with implicit maps.

Figure 4.17: Two versions of the `mri-q` benchmark from the Parboil benchmark suite, implemented in Futhark. On the left, the original version with explicit **map**s, and on the right the version with Automap.

ranks are known. It is perhaps possible to encode this situation as a form of conditional rule in the vein of [101], but we did not find it necessary in order to obtain a useful system.

4.8.4 Elaboration

In the final phase, flexible function applications are elaborated into **map**s and **rep**s. The phase operates similarly to the AM transformation of Section 4.7 except only non-induced **rep**s are elaborated due to rep fusion (i.e., pushing **rep**s to the bottom of the elaborated **map** nest).

4.9 Evaluation

We evaluate the practical merits of Automap on a collection of Futhark programs ported from the benchmark suites Parboil [94], PBBS [2], FinPar [3], Accelerate [14], and Rodinia [15]. This collection comprises 8600 source lines of code spread across 67 files. We investigate the following questions:

1. How many **map**s do we eliminate through Automap?
2. Why have the remaining **map**s not been eliminated?
3. How much does Automap slow down type checking?

The reason we focus on **map** and not **rep** is that implicit **map**s are far more common, and implicit **rep**s tend to occur as a consequence of an implicit **map** in the same application. The results are quantified in Figure 4.19.[10]

In principle, we could expect that any expression **map** $f\ x$ could be replaced with $f\ x$, as Automap makes the **map** implicit. In practice, this can lead to an ambiguous or ill-typed program. Even when it does not, the smallest Automap solution might be semantically different. Consider a term such as

$$\textbf{map}\ (\lambda x.\ \textbf{map}\ (\cdot x)\ ys)\ xs,$$

[10] We have prepared an artifact for reproduction of these results.

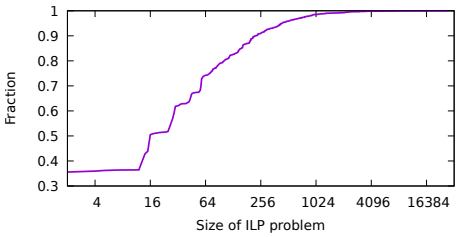


Figure 4.18: Proportion of ILP problems that have less than some given number of constraints. Note that almost all problems contain at most 1024 constraints.

Number of programs: 67

Change in lines of code: 8623 ⇒ 8517

Change in `map`s: 467 ⇒ 213

Largest ILP size: 28104 constraints

Median ILP size: 18 constraints

Mean ILP size: 121 constraints

Mean type checking slowdown: 2.50×

Figure 4.19: Changes in various metrics resulting from the addition of Automap to the language and type checker. There is no change to application run-time performance.

corresponding to an outer product of the vectors xs and ys producing a matrix. Removing the innermost **`map`** produces

$$\textbf{map}\ (\lambda x.\, x \cdot ys)\ xs,$$

which Automap handles as expected, but further removing the outer **`map`** leads to

$$(\lambda x.\, x \cdot ys)\ xs\ ys = xs \cdot ys,$$

which is elaborated by Automap to **`map`** $(\cdot)$ xs ys, yielding a vector. Further, this program also requires that xs and ys have the same size, which was not the case in the original expression—meaning it may no longer be well-typed under size-dependent typing (Section 4.8.3).

Even when Automap elaboration is unambiguous, removing **`map`**s can decrease readability. This is of course a subjective judgment, but we have only removed **`map`**s when we judge that this results in an increase in readability. Unsurprisingly, this is often the case for linear algebra expressions.

Figure 4.17 shows an example based on the `mri-q` benchmark from Parboil. The original program is shown in Figure 4.17a, with a total of ten instances of **`map`**, somewhat cluttering the program. The modified program on Figure 4.17b contains no explicit **`map`**s, and particularly the computation of `qr` and `qi` is much more concise. This program implements a single mathematical formula, and so represents a "best case" scenario for Automap.

4.9.1 Quantifying `map`s

The unmodified benchmarks contain 467 applications of the **`map`** function and its variants `map2` up to `map5`. After manual rewrites to take advantage of Automap where appropriate, 213 applications are left, corresponding to a 54% reduction.

Futhark supports higher-order functions, and so programmers may define **`map`** variants that are not included in the above count. However, this is not a common programming style in the Futhark benchmark suite, and so we consider our count to be accurate.

The programming style in the benchmark suite already uses type annotations for most top level functions, and we did not find it necessary to add any annotations in order to resolve ambiguities.[11]

4.9.2 Impact on Type Checking

Our AUTOMAP-enabled type checker is unoptimized, but still shows the practicality of our approach. For most programs, a little over half of the total time spent on type checking is taken up by rank analysis; specifically on constructing and solving ILP problems. Compared to the unmodified type checker, we see less than $3\times$ slowdown on most programs. The outlier is `myocyte`, which is about $13\times$ slower. This program contains a dense 437 line function with many arithmetic operations. As each application gives rise to six ILP variables and associated constraints, AUTOMAP produce an ILP program with 28104 constraints, which is slow to solve. This could be optimized by not generating constraints for applications whose implicit **`map`**s and **`rep`**s can be determined locally at the application site (as is commonly the case with arithmetic operations). The distribution of ILP sizes for the entire benchmark suite is shown in Figure 4.18.

4.9.3 Programmer Experience

In practice, we found AUTOMAP to be unsurprising: if a program is unambiguous, it generally elaborated as we expected. Since real programs tend to sufficiently constrain rank polymorphic function applications to be unambiguous, ambiguity in practice is relatively rare.

It should also be emphasized that the system is both transparent and flexible: any program can always be elaborated into a version with all **`map`**s and **`rep`**s explicit, so programmers can always validate that a program elaborates as they intended. Programmers can also use AUTOMAP to whatever degree they wish by omitting all **`map`**s/**`rep`**s (up to ambiguity), only some, or none.

Handling ambiguity in elaboration systems like AUTOMAP is straightforward: signal an error and report each possible elaboration. When there are no solutions to the ILP (a consequence of an ill-typed program), the debugging work flow is reminiscent of debugging in any ML-style language with type inference; instead of just inserting type annotations (i.e., making implicit type annotations explicit) to narrow down the location of the bug, **`map`**s and **`rep`**s are also inserted to make these constructs explicit as well.

4.10 Future Work

4.10.1 Higher-order Functions

While AUTOMAP works in the presence of higher-order functions, their use sometimes results in ambiguity. As an example, suppose we have a polymorphic function `pipe` : $\alpha \to (\alpha \to \beta) \to \beta$ (the prefix form of $\triangleright$) and a function `f` : `int` $\to$ `int`, the term `pipe` (`indices` A) `f` is ambiguous, with the following elaborations:

1. **`map`** `pipe` (`indices` A) (**`rep`** `f`),
2. `pipe` (`indices` A) (**`rep`** `f`).

[11] Type annotations refine the constraints in the ILP and can thereby eliminate additional solutions.

These elaborations have the same size, due to not counting induced **`rep`**s (Section 4.8.2). Piping is a common pattern, so better interaction with AUTOMAP is desirable—possibly by refining the cost function to count induced **`rep`**s in those cases where the result is otherwise ambiguous.

4.10.2 Solving Constraints Locally

As mentioned in Section 4.9.2, there are also opportunities to reduce the size of the ILP program by eliding constraints for applications that can be locally solved.

4.10.3 Efficient Ambiguity Checking

To detect ambiguity we add constraints to the ILP to ban an initial solution and look for a second of the same size. If $\{M_0, R_0, \ldots, M_n, R_n\}$ are the rank variables of an ILP and s_r and s'_r are two solutions of the ILP, we require that $\sum_{Q \in \{M_0,R_0,\ldots,M_n,R_n\}} |s_\mathrm{r}(Q) - s'_\mathrm{r}(Q)| \geq 1$. This constraint can be linearlized [97], but requires introducing new variables and constraints to the ILP.

Alternatively, rather than discriminating on the rank variables themselves, the binary variables b_R, b_M used to encode the $M \Vdash R$ constraint (see section 4.6.4) may be discriminated on instead: $\sum_{Q \in \{M_0,R_0,\ldots,M_n,R_n\}} |s_\mathrm{r}(b_Q) - s'_\mathrm{r}(b_Q)| \geq 1$. This constraint can be linearized without introducing any new variables nor constraints (by exploiting the fact that b_Q is binary) because it's equivalent to

$$\sum_{\substack{Q \in \{M_0,R_0,\ldots,M_n,R_n\} \\ s_\mathrm{r}(b_Q)=0}} s'_\mathrm{r}(b_Q) + \sum_{\substack{Q \in \{M_0,R_0,\ldots,M_n,R_n\} \\ s_\mathrm{r}(b_Q)=1}} 1 - s'_\mathrm{r}(b_Q) \geq 1.$$

We conjecture that any rank constraint set $|C|$ that is ambiguous at size k (where k is minimal) must have two distinct solutions s_r and s'_r (both with size k) such that

$$\sum_{Q \in \{M_0,R_0,\ldots,M_n,R_n\}} |s_\mathrm{r}(b_Q) - s'_\mathrm{r}(b_Q)| \geq 1,$$

if each b_Q is additionally constrained with $b_Q \leq Q$, but we've been unable to find a proof (nor counter-example).

Another unexplored avenue to detect ambiguity is to eschew adding constraints to the ILP to ban solutions altogether. Instead, the ILP solver itself can be modified to further explore the solution space after finding an initial solution [20, 21].

4.11 Related Work

4.11.1 Data Parallelism

NumPy [34] is likely the most popular rank-polymorphic programming system in current use, and AUTOMAP largely targets the same kinds of applications as NumPy. One important difference is that NumPy's implicit rank-polymorphic behavior cannot be manually or systematically elaborated—while guiding principles exist,[12] each NumPy function has complete freedom to inspect the ranks and shapes of array arguments and make arbitrary control flow decisions. In contrast, anything expressible with AUTOMAP

[12] https://numpy.org/doc/stable/user/basics.broadcasting.html

can always be rewritten with explicit **`map`**s and **`rep`**s at application sites, without any change in run-time performance. As a minor difference, NumPy also allows broadcasting where a unit dimension is implicitly expanded as needed to make otherwise rank-compatible operands have the same size.

Originating from Iverson's *scalar multiple* [47], APL only allows broadcasting (or more properly, scalar extension [46]) of scalar elements, and does not for example allow a vector to be added to a matrix, which is allowed in Automap and NumPy. Whereas APL is traditionally a dynamically-typed language, approaches exist to infer scalar extensions and map nests statically [31, 22]. The limitation to scalars also exist in work by Thatte [96], which uses subtyping for inferring coercions for adjusting function applications to match call sites, although Thatte does support higher order functions.

Single Assignment C is perhaps the most well-developed statically typed rank-polymorphic language, although it does not support parametric polymorphism or higher-order functions. It does however support a particularly flexible form of rank polymorphism, including rank specialization, which provides a powerful form of control flow [88, 87]. Also related to this work is the work on Remora [92], which allows for expressing many aspects of APL, including rank-polymorphism, but in an explicitly typed context. In the context of Remora, work has been proposed for using constraint solving for type checking rank-polymorphic programs [91]. Gibbons has shown how to encode rank polymorphism in Glasgow Haskell through the use of Naperian functors [28], which also supports richer structures than just arrays. One limitation of Gibbons' approach is that the functorial map operation always operates the full shape, whilst the present work allows only some dimensions of a multidimensional array to be mapped. The encoding also requires a very rich type system.

4.11.2 Type Systems and Type Inference

The Hindley-Damas-Milner type system [41, 58, 18] has long been used as the theoretical foundation for a class of programming languages with polymorphic types and complete type inference, including for the OCaml and Haskell programming languages. Over the years, many extensions to the HM system have been proposed notably HM(X) which is a general framework where HM is parameterized by constraints [68]. The original HM type system supports local let-generalization, which means that polymorphic types are inferred not only for top-level functions, but for every let-binding. However, local let-generalization leads to a number of problems, most famously that it is unsound in the presence of mutable reference cells [27]. The trouble with local let-generalization has led a number of papers to propose that "let should not be generalized" [100, 101]. While local let-generalization causes many difficulties, a study of the Haskell ecosystem found that local let-generalization is rarely used, and when used, the programs are often straightforward to refactor. In this thesis, and in Futhark in general, generalization is only supported at the top-level.

There is a large body of related work on constraint-based type systems [95, 25, 49], including work on implementing type classes [76]. Whereas Automap is based on a constraint-based type system, constraints are local to function definitions, which simplifies the type system significantly compared to much other work.

4.11.3 Implicit Program Constructs

Implicit program constructs, like Automap, are found in several programming languages. Implicit parameters are known from Haskell [52] and Scala [69] where in the

latter they are used to support type classes [70, 69]. A large-scale study of real-world Scala code found that implicit parameters are widely showing that programmers want to use implicit constructs to make their code shorter and more concise [51]. An overview of implicit programming constructs is provided by [48].

4.12 Conclusions

We have presented an extension of an ML-style type system that supports a limited form of rank polymorphism, while still retaining support for parametric polymorphism (with top-level let-generalization) and higher-order functions. The type inference algorithm is based on generating and solving an ILP problem. We have formally proven the soundness and other properties of the system. Through an evaluation based on 8600 lines of code, we have demonstrated that the type system results in a significant reduction in the amount of explicit **`map`**s, arguably a significant increase in readability of parallel expressions, and that the ILP problems can be solved in reasonable time.

Data-Availability Statement

An artifact of our Automap prototype in Futhark that reproduces the benchmarking results of Section 4.9 is available on Zenodo [85].

Bibliography

[1] Martín Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[2] Daniel Anderson et al. "The problem-based benchmark suite (PBBS), V2". In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 445–447. ISBN: 9781450392044. DOI: 10.1145/3503221.3508422. URL: https://doi.org/10.1145/3503221.3508422.

[3] Christian Andreetta et al. "FinPar: A Parallel Financial Benchmark". In: *ACM Trans. Archit. Code Optim.* 13.2 (June 2016), 18:1–18:27. ISSN: 1544-3566.

[4] M. Araya-Polo and Laurent Hascoët. "Data Flow Algorithms in the Tapenade Tool for Automatic Differentiation". In: *Proceedings of the European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS 2004)*. Ed. by P. Neittaanmäki et al. online at http://www.mit.jyu.fi/eccomas2004/proceedings/pdf/550.pdf. Jyväskylä, Finland: University of Jyväskylä, 2004. ISBN: 951-39-1868-8.

[5] Lubin Bailly, Troels Henriksen, and Martin Elsman. "Shape-Constrained Array Programming with Size-Dependent Types". In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*. FHPNC 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, 29â€"41. ISBN: 9798400702969. DOI: 10.1145/3609024.3609412. URL: https://doi.org/10.1145/3609024.3609412.

[6] Atılım Günes Baydin et al. "Automatic Differentiation in Machine Learning: A Survey". In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), pp. 5595–5637. ISSN: 1532-4435.

[7] Gilbert Bernstein et al. *Differentiating a Tensor Language*. 2020. DOI: 10.48550/ARXIV.2008.11256. URL: https://arxiv.org/abs/2008.11256.

[8] Guy E. Blelloch. "Prefix sums and their applications". In: 1990.

[9] Guy E. Blelloch et al. "Implementation of a portable nested data-parallel language". In: *SIGPLAN Not.* 28.7 (July 1993), pp. 102–111. ISSN: 0362-1340. DOI: 10.1145/173284.155343. URL: https://doi.org/10.1145/173284.155343.

[10] Uday Bondhugula et al. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 101–113. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375595. URL: http://doi.acm.org/10.1145/1375581.1375595.

[11] Léon Bottou and Yoshua Bengio. "Convergence Properties of the K-Means Algorithms". In: *Advances in Neural Information Processing Systems*. Ed. by G. Tesauro, D. Touretzky, and T. Leen. Vol. 7. MIT Press, 1994. URL: https://proceedings.neurips.cc/paper/1994/file/a1140a3d0df1c81e24ae954d935e8926-Paper.pdf.

[12] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: http://github.com/google/jax.

[13] Lotte Maria Bruun et al. "Reverse-Mode AD of Multi-Reduce and Scan in Futhark". In: *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*. IFL '23. Braga, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400716317. DOI: 10.1145/3652561.3652575. URL: https://doi.org/10.1145/3652561.3652575.

[14] Manuel MT Chakravarty et al. "Accelerating Haskell array codes with multicore GPUs". In: *Proc. of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 3–14.

[15] S. Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Oct. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[16] Sharan Chetlur et al. *cuDNN: Efficient Primitives for Deep Learning*. 2014. DOI: 10.48550/ARXIV.1410.0759. URL: https://arxiv.org/abs/1410.0759.

[17] Jason PC Chiu and Eric Nichols. "Named entity recognition with bidirectional LSTM-CNNs". In: *Transactions of the Association for Computational Linguistics* 4 (2016), pp. 357–370.

[18] Luis Damas. "Type assignment in programming languages". PhD thesis. The University of Edinburgh, 1984.

[19] Francis Dang, Hao Yu, and Lawrence Rauchwerger. "The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops". In: *Int. Par. and Distr. Processing Symp. (PDPS)*. 2002, pp. 20–29.

[20] Emilie Danna and David L Woodruff. "How to select a small set of diverse solutions to mixed integer programming problems". In: *Operations Research Letters* 37.4 (2009), pp. 255–260.

[21] Emilie Danna et al. "Generating multiple solutions for mixed integer programming problems". In: *International Conference on Integer Programming and Combinatorial Optimization*. Springer. 2007, pp. 280–294.

[22] Martin Elsman and Martin Dybdal. "Compiling a Subset of APL Into a Typed Intermediate Language". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 101–106. ISBN: 9781450329378. DOI: 10.1145/2627373.2627390. URL: https://doi.org/10.1145/2627373.2627390.

[23] Martin Elsman et al. "Combinatory Adjoints and Differentiation". In: *Electronic Proceedings in Theoretical Computer Science* 360 (June 2022), pp. 1–26. ISSN: 2075-2180. DOI: 10.4204/eptcs.360.1. URL: http://dx.doi.org/10.4204/EPTCS.360.1.

[24] Martin Elsman et al. "Static Interpretation of Higher-order Modules in Futhark: Functional GPU Programming in the Large". In: *Proceedings of the ACM on Programming Languages* 2.ICFP (July 2018), 97:1–97:30. ISSN: 2475-1421.

[25] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. "Flow-Sensitive Type Qualifiers". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI '02. Berlin, Germany: Association for Computing Machinery, 2002, pp. 1–12. ISBN: 1581134630. DOI: 10.1145/512529.512531. URL: https://doi.org/10.1145/512529.512531.

[26] Roy Frostig, Matthew James Johnson, and Chris Leary. "Compiling machine learning programs via high-level tracing". In: *Systems for Machine Learning* (2018), pp. 23–24.

[27] Jacques Garrigue. "Relaxing the value restriction". In: *International Symposium on Functional and Logic Programming*. Springer. 2004, pp. 196–213.

[28] Jeremy Gibbons. "APLicative Programming with Naperian Functors". In: *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*. Uppsala, Sweden: Springer-Verlag, 2017, pp. 556–583. ISBN: 978-3-662-54433-4. DOI: 10.1007/978-3-662-54434-1_21. URL: https://doi.org/10.1007/978-3-662-54434-1_21.

[29] Andreas Griewank, David Juedes, and Jean Utke. "Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++". In: *ACM Transactions on Mathematical Software (TOMS)* 22.2 (1996), pp. 131–167.

[30] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[31] Leo J. Guibas and Douglas K. Wyatt. "Compilation and delayed evaluation in APL". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '78. Tucson, Arizona: Association for Computing Machinery, 1978, pp. 1–8. ISBN: 9781450373487. DOI: 10.1145/512760.512761. URL: https://doi.org/10.1145/512760.512761.

[32] Mary W. Hall et al. "Interprocedural Parallelization Analysis in SUIF". In: *Trans. on Prog. Lang. and Sys. (TOPLAS)* 27(4) (2005), pp. 662–731.

[33] F. Maxwell Harper and Joseph A. Konstan. "The MovieLens Datasets: History and Context". In: *ACM Trans. Interact. Intell. Syst.* 5.4 (Dec. 2015). ISSN: 2160-6455. DOI: 10.1145/2827872. URL: https://doi.org/10.1145/2827872.

[34] Charles R Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362.

[35] Troels Henriksen. "Design and Implementation of the Futhark Programming Language". PhD thesis. University of Copenhagen, Faculty of Science [Department of Computer Science], 2017.

[36] Troels Henriksen and Martin Elsman. "Towards Size-Dependent Types for Array Programming". In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ARRAY 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 1–14. ISBN: 9781450384667. DOI: 10.1145/3460944.3464310. URL: https://doi.org/10.1145/3460944.3464310.

[37] Troels Henriksen et al. "Compiling Generalized Histograms for GPU". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[38] Troels Henriksen et al. "Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. URL: http://doi.acm.org/10.1145/3062341.3062354.

[39] Troels Henriksen et al. "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 556–571. ISSN: 0362-1340. DOI: 10.1145/3140587.3062354. URL: https://doi.org/10.1145/3140587.3062354.

[40] Troels Henriksen et al. "Incremental Flattening for Nested Data Parallelism". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: ACM, 2019, pp. 53–67. ISBN: 978-1-4503-6225-2. DOI: 10.1145/3293883.3295707. URL: http://doi.acm.org/10.1145/3293883.3295707.

[41] Roger Hindley. "The principal type-scheme of an object in combinatory logic". In: *Transactions of the American Mathematical Society (AMS)* (1969).

[42] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. "High-performance defunctionalization in Futhark". In: *Symposium on Trends in Functional Programming (TFP'18)*. Sept. 2018.

[43] P. Hovland and C. Bischof. "Automatic differentiation for message-passing parallel programs". In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. 1998, pp. 98–104. DOI: 10.1109/IPPS.1998.669896.

[44] Jan Hückelheim and Laurent Hascoët. *Source-to-Source Automatic Differentiation of OpenMP Parallel Loops*. 2021. DOI: 10.48550/ARXIV.2111.01861. URL: https://arxiv.org/abs/2111.01861.

[45] Jan Hückelheim et al. "Automatic Differentiation for Adjoint Stencil Loops". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450362955. DOI: 10.1145/3337821.3337906. URL: https://doi.org/10.1145/3337821.3337906.

[46] Roger K. W. Hui and Morten J. Kromberg. "APL since 1978". In: *Proc. ACM Program. Lang.* 4.HOPL (June 2020). DOI: 10.1145/3386319. URL: https://doi.org/10.1145/3386319.

[47] Kenneth E. Iverson. *A programming language*. USA: John Wiley & Sons, Inc., 1962. ISBN: 0471430145.

[48] Alexander Paul Jeffery and Martin Berger. "On Implicit Program Constructs". PhD thesis. University of Sussex, 2020.

[49] Mark P. Jones. "A theory of qualified types". In: *Science of Computer Programming* 22.3 (1994), pp. 231–256. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(94)00005-0. URL: https://www.sciencedirect.com/science/article/pii/0167642394000050.

[50] *KDD Cup 1999 Data*. 1999. URL: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html (visited on 05/15/2022).

[51] Filip Křikava, Heather Miller, and Jan Vitek. "Scala implicits are everywhere: A large-scale study of the use of scala implicits in the wild". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–28.

[52] Jeffrey R Lewis et al. "Implicit parameters: Dynamic scoping with static types". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2000, pp. 108–118.

[53] B. Lu and J. Mellor-Crummey. "Compiler optimization of implicit reductions for distributed memory multiprocessors". In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. 1998, pp. 42–51. DOI: 10.1109/IPPS.1998.669887.

[54] Andrew Maas et al. "Learning word vectors for sentiment analysis". In: *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. 2011, pp. 142–150.

[55] Oleksandr Manzyuk et al. "Perturbation confusion in forward automatic differentiation of higher-order functions". In: *Journal of Functional Programming* 29 (2019), e12. DOI: 10.1017/S095679681900008X.

[56] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. "Building a Large Annotated Corpus of English: The Penn Treebank". In: *Computational Linguistics* 19.2 (1993), pp. 313–330. URL: https://aclanthology.org/J93-2004.

[57] Charles C. Margossian. "A review of automatic differentiation and its efficient implementation". In: *WIREs Data Mining and Knowledge Discovery* 9.4 (2019), e1305. DOI: https://doi.org/10.1002/widm.1305. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1305. URL: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1305.

[58] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* (1978).

[59] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

[60] Torben Ægidius Mogensen. *Introduction to Compiler Design*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 0857298283.

[61] Greg Morrisett. "Compiling with Types". PhD thesis. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, Dec. 1995.

[62] William S. Moses and Valentin Churavy. "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients". In: *Advances in Neural Information Processing Systems 33*. 2020.

[63] William S. Moses et al. "Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476165. URL: https://doi.org/10.1145/3458817.3476165.

[64] Uwe Naumann. *The Art of Differentiating Computer Programs*. Society for Industrial and Applied Mathematics, 2011. DOI: 10.1137/1.9781611972078. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611972078. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611972078.

[65] Corey J. Nolet et al. *GPU Semiring Primitives for Sparse Neighborhood Methods*. 2021. DOI: 10.48550/ARXIV.2104.06357. URL: https://arxiv.org/abs/2104.06357.

[66] Cosmin E. Oancea, Troels Henriksen, and Robert Schenck. *Reverse Mode Automatic Differentiation.* Lecture Slides for the Parallel Functional Programming MSc Course. Dec. 2020. URL: https://github.com/diku-dk/pfp-e2020-pub/blob/master/slides/L8-reverse-ad.pdf.

[67] Cosmin E. Oancea and Lawrence Rauchwerger. "Logical Inference Techniques for Loop Parallelization". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012, pp. 509–520. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254124. URL: http://doi.acm.org/10.1145/2254064.2254124.

[68] Martin Odersky, Martin Sulzmann, and Martin Wehr. "Type inference with constrained types". In: *Theory and Practice of Object Systems* 5.1 (1999), pp. 35–55. DOI: https://doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4.

[69] Martin Odersky et al. "Simplicitly: Foundations and applications of implicit function types". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–29.

[70] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. "Type classes as objects and implicits". In: *ACM Sigplan Notices* 45.10 (2010), pp. 341–360.

[71] Adam Paszke et al. "Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming". In: *Proc. ACM Program. Lang.* 5.ICFP (Aug. 2021). DOI: 10.1145/3473593. URL: https://doi.org/10.1145/3473593.

[72] Adam Paszke et al. "Parallelism-Preserving Automatic Differentiation for Second-Order Array Languages". In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*. FHPNC 2021. Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 13–23. ISBN: 9781450386142. DOI: 10.1145/3471873.3472975. URL: https://doi.org/10.1145/3471873.3472975.

[73] Adam Paszke et al. "PyTorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019), pp. 8026–8037.

[74] Barak A. Pearlmutter and Jeffrey Mark Siskind. "Reverse-Mode AD in a Functional Framework: Lambda the Ultimate Backpropagator". In: *ACM Trans. Program. Lang. Syst.* 30.2 (Mar. 2008). ISSN: 0164-0925. DOI: 10.1145/1330017.1330018. URL: https://doi.org/10.1145/1330017.1330018.

[75] Jeffrey Pennington, Richard Socher, and Christopher D Manning. "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[76] John Peterson and Mark Jones. "Implementing Type Classes". In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI '93. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1993, pp. 227–236. ISBN: 0897915984. DOI: 10.1145/155090.155112. URL: https://doi.org/10.1145/155090.155112.

[77] Simon Peyton Jones, Mark Jones, and Erik Meijer. "Type classes: an exploration of the design space". In: *Haskell workshop*. Jan. 1997. URL: https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/.

[78] Ari Rasch, Richard Schulze, and Sergei Gorlatch. "Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms". In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2019, pp. 354–369. DOI: 10.1109/PACT.2019.00035.

[79] Ola Rønning, Daniel Hardt, and Anders Søgaard. "Sluice resolution without hand-crafted features over brittle syntax trees". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. 2018, pp. 236–241.

[80] Amr Sabry and Matthias Felleisen. "Reasoning About Programs in Continuation-passing Style." In: *SIGPLAN Lisp Pointers* V.1 (Jan. 1992), pp. 288–298. ISSN: 1045-3563.

[81] Haşim Sak, Andrew Senior, and Françoise Beaufays. *Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition*. 2014. DOI: 10.48550/ARXIV.1402.1128. URL: https://arxiv.org/abs/1402.1128.

[82] Robert Schenck et al. "AD for an Array Language with Nested Parallelism". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022. ISBN: 9784665454445.

[83] Robert Schenck et al. "AUTOMAP: Inferring Rank-Polymorphic Function Applications with Integer Linear Programming". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA2 (2024). DOI: 10.1145/3689774. URL: https://doi.org/10.1145/3689774.

[84] Robert Schenck et al. *futhark-ad-sc22*. Version v1.0.4. July 2022. DOI: 10.5281/zenodo.6853848. URL: https://doi.org/10.5281/zenodo.6853848.

[85] Robert Schenck et al. *futhark-oopsla24*. Version 1.0.4. July 2024. DOI: 10.5281/zenodo.12775308. URL: https://doi.org/10.5281/zenodo.12775308.

[86] Amir Shaikhha et al. "Efficient Differentiable Programming in a Functional Array-Processing Language". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341701. URL: https://doi.org/10.1145/3341701.

[87] Artjoms Šinkarovs, Thomas Koopman, and Sven-Bodo Scholz. "Rank-Polymorphism for Shape-Guided Blocking". In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*. FHPNC 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 1–14. ISBN: 9798400702969. DOI: 10.1145/3609024.3609410.

[88] Artjoms Šinkarovs and Sven-Bodo Scholz. "Parallel Scan as a Multidimensional Array Problem". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ARRAY 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 1–11. ISBN: 9781450392693. DOI: 10.1145/3520306.3534500. URL: SinkarovsScholzARRAY22.pdf.

[89] Jeffrey Mark Siskind and Barak A. Pearlmutter. "Divide-and-Conquer Checkpointing for Arbitrary Programs with No User Annotation". In: *Optimization Methods and Software* 33.4-6 (2018), pp. 1288–1330. DOI: 10.1080/10556788.2018.1459621. eprint: https://doi.org/10.1080/10556788.2018.1459621. URL: https://doi.org/10.1080/10556788.2018.1459621.

[90] Justin Slepak. "A Typed Programming Language". PhD thesis. Northeastern University Boston, 2020.

[91] Justin Slepak, Panagiotis Manolios, and Olin Shivers. "Rank polymorphism viewed as a constraint problem". In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 34–41. ISBN: 9781450358521. DOI: 10.1145/3219753.3219758. URL: https://doi.org/10.1145/3219753.3219758.

[92] Justin Slepak, Olin Shivers, and Panagiotis Manolios. "An array-oriented language with static rank polymorphism". In: *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 23*. Springer. 2014, pp. 27–46. DOI: 10.1007/978-3-642-54833-8_3.

[93] Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. "A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning". In: *Optimization Methods & Software* 33.4–6 (2018). Ed. by Bruce Christianson, Shaun A. Forth, and Andreas Griewank, pp. 889–906. DOI: 10.1080/10556788.2018.1435651. eprint: https://doi.org/10.1080/10556788.2018.1435651. URL: https://doi.org/10.1080/10556788.2018.1435651.

[94] John A Stratton et al. "Parboil: A revised benchmark suite for scientific and commercial throughput computing". In: *Center for Reliable and High-Performance Computing* 127 (2012).

[95] Martin Franz Sulzmann and Paul Hudak. "A General Framework for Hindley/Milner Type Systems with Constraints". AAI9973781. PhD thesis. USA, 2000. ISBN: 0599791896.

[96] Satish Thatte. "A type system for implicit scaling". In: *Science of Computer Programming* 17.1 (1991), pp. 217–245. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(91)90040-5. URL: https://www.sciencedirect.com/science/article/pii/0167642391900405.

[97] Jung-Fa Tsai, Ming-Hua Lin, and Yi-Chung Hu. "Finding multiple solutions to general integer linear programs". In: *European Journal of Operational Research* 184.2 (2008), pp. 802–809.

[98] Nicolas Vasilache et al. "The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically". In: *ACM Trans. Archit. Code Optim.* 16.4 (Oct. 2019). ISSN: 1544-3566. DOI: 10.1145/3355606. URL: https://doi.org/10.1145/3355606.

[99] Sven Verdoolaege et al. "Polyhedral Parallel Code Generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 54:1–54:23. ISSN: 1544-3566. DOI: 10.1145/2400682.2400713. URL: http://doi.acm.org/10.1145/2400682.2400713.

[100] Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. "Let should not be generalized". In: *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '10. Madrid, Spain: Association for Computing Machinery, 2010, pp. 39–50. ISBN: 9781605588919. DOI: 10.1145/1708016.1708023. URL: https://doi.org/10.1145/1708016.1708023.

[101] Dimitrios Vytiniotis et al. "OutsideIn (X) Modular type inference with local assumptions". In: *Journal of functional programming* 21.4-5 (2011), pp. 333–412.

[102] Fei Wang et al. "Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341700. URL: https://doi.org/10.1145/3341700.

[103] A.K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness". In: *Inf. Comput.* 115.1 (Nov. 1994), pp. 38–94. ISSN: 0890-5401. DOI: 10.1006/inco.1994.1093. URL: https://doi.org/10.1006/inco.1994.1093.

Appendix A

Proofs for Automap

This appendix includes proofs for Chapter 4. Each proposition is numbered and stated as it is in that chapter and in the same corresponding sections. (For example, Proposition 1 from Section 4.4 will also be denoted Proposition 1 in this appendix and will appear under Appendix A.4.) The appendix will also introduce lemmas that do not have corresponding lemmas in chapter 4.

A.4 Target Language

The following lemma shows that the type scheme generalization is closed under type substitutions; this is needed to show that the target language typing derivation is closed under substitution.

Lemma 1 ($\geq$ Closed Under Type Substitution)**.** *If $\sigma \geq \tau$ then $s(\sigma) \geq s(\tau)$ for any non-capturing substitution s.*

Proof. Suppose $\sigma = \forall\vec{\alpha}.\tau'$. Since $\sigma \geq \tau$, there exists s_t' such that $\lceil 1 \rceil$ $s_t'(\tau') = \tau$ with $\mathrm{dom}(s_t') = \{\vec{\alpha}\}$. By $\lceil 1 \rceil$, we have $s(s_t'(\tau')) = s(\tau)$.

We need to find s_t'' with $\mathrm{dom}(s_t'') = \{\vec{\alpha}\}$ such that $s_t''(s(\tau')) = s(\tau) = s(s_t'(\tau'))$. Define

$$s_t''(\beta) = \begin{cases} \beta & \text{if } \beta \notin \{\vec{\alpha}\}, \\ s(s_t'(\beta)) & \text{if } \beta \in \{\vec{\alpha}\}. \end{cases}$$

Then, if $\beta \notin \{\vec{\alpha}\}$,

$$s_t''(s(\beta)) = s(\beta) = s(s_t'(\beta)),$$

and if $\beta \in \{\vec{\alpha}\}$,

$$s_t''(s_t(\beta)) = s_t''(\beta) = s(s_t'(\beta)),$$

as required. □

Proposition 1 (Typing Closed Under Type Substitution)**.** *If $\Gamma \vdash p : \sigma$ then $s_t(\Gamma) \vdash p : s_t(\sigma)$, for any type substitution s_t.*

Proof. By induction over the typing derivation. The SV- rules for values are skipped because they're either trivial or analogous to the corresponding expression rules.

Case S-Var ($\Gamma, x : \sigma \vdash x \ : \ \sigma$)**:**
Immediate.

Case S-Inst ($\Gamma \vdash e \ : \ \tau$)**:**
We have $\lceil 1 \rceil\ \Gamma \vdash e \ : \ \sigma$ and $\lceil 2 \rceil\ \sigma \geq \tau$. By the IH, $\lceil 3 \rceil\ s_t(\Gamma) \vdash e \ : \ s_t(\sigma)$, Since $\geq$ is closed under substitution, $\lceil 1 \rceil$ yields $\lceil 4 \rceil\ s_t(\sigma) \geq s_t(\tau)$. Applying S-Inst to $\lceil 3 \rceil$ and $\lceil 4 \rceil$ gives $s_t(\Gamma) \vdash e \ : \ s_t(\tau)$.

Case S-Array ($\Gamma \vdash [e_1, \ldots, e_n] \ : \ [\,]\,\tau$)**:**
We have $\lceil 1 \rceil\ \forall i \in \{1, \ldots, n\}\,.\,\Gamma \vdash e_i \ : \ \tau$. Applying the IH to $\lceil 1 \rceil$, $\lceil 2 \rceil$ $\forall i \in \{1, \ldots, n\}\,.\,s_t(\Gamma) \vdash e_i \ : \ s_t(\tau)$. Applying S-Array to $\lceil 2 \rceil$, $s_t(\Gamma) \vdash [v_1, \ldots, v_n] \ : \ s_t([\,]\,\tau)$.

Case S-Val ($\Gamma \vdash v \ : \ \tau$)**:**
We have $\lceil 1 \rceil \vdash v \ : \ \tau$. Applying the IH to $\lceil 1 \rceil$, $\lceil 2 \rceil \vdash v \ : \ s_t(\tau)$. Applying S-Val to $\lceil 2 \rceil$, $s_t(\Gamma) \vdash v \ : \ s_t(\tau)$.

Case S-Fun ($\Gamma \vdash \lambda x.\, e \ : \ \tau \to \tau'$)**:**
We have $\lceil 1 \rceil\ \Gamma, x : \tau \vdash e \ : \ \tau'$. Applying the IH to $\lceil 1 \rceil$, $\lceil 2 \rceil\ s_t(\Gamma), x : s_t(\tau) \vdash e \ : \ s_t(\tau')$. Applying S-Fun to $\lceil 2 \rceil$, $s_t(\Gamma) \vdash \lambda x.\, e \ : \ s_t(\tau \to \tau')$.

Case S-App ($\Gamma \vdash e_1\ e_2 \ : \ \tau_2$)**:**
We have $\lceil 1 \rceil\ \Gamma \vdash e_1 \ : \ \tau_1 \to \tau_2$ and $\lceil 2 \rceil\ \Gamma \vdash e_2 \ : \ \tau_1$. Applying the IH to $\lceil 1 \rceil$ and $\lceil 2 \rceil$, we obtain $\lceil 3 \rceil\ s_t(\Gamma) \vdash e_1 \ : \ s_t(\tau_1 \to \tau_2)$ and $\lceil 4 \rceil\ s_t(\Gamma) \vdash e_2 \ : \ s_t(\tau_1)$. Applying S-App to $\lceil 3 \rceil$ and $\lceil 4 \rceil$, $s_t(\Gamma) \vdash e_1\ e_2 \ : \ s_t(\tau_2)$

Case S-Map ($\Gamma \vdash \mathbf{map}\ e_1\ e_2 \ : \ [\,]\tau_2$)**:**
We have $\lceil 1 \rceil\ \Gamma \vdash e_1 \ : \ \tau_1 \to \tau_2$ and $\lceil 2 \rceil\ \Gamma \vdash e_2 \ : \ [\,]\tau_1$. Applying the IH to $\lceil 1 \rceil$ and $\lceil 2 \rceil$, we obtain $\lceil 3 \rceil\ s_t(\Gamma) \vdash e_1 \ : \ s_t(\tau_1 \to \tau_2)$ and $\lceil 4 \rceil\ s_t(\Gamma) \vdash e_2 \ : \ s_t([\,]\tau_1)$. Applying S-Map to $\lceil 3 \rceil$ and $\lceil 4 \rceil$, $s_t(\Gamma) \vdash \mathbf{map}\ e_1\ e_2 \ : \ s_t([\,]\tau_2)$

Case S-Rep ($\Gamma \vdash \mathbf{rep}\ e \ : \ [\,]\,\tau$)**:**
We have $\lceil 1 \rceil\ \Gamma \vdash e \ : \ \tau$. Applying the IH to $\lceil 1 \rceil$, $\lceil 2 \rceil\ s_t(\Gamma) \vdash e \ : \ s_t(\tau)$. Applying S-Rep to $\lceil 2 \rceil$, $s_t(\Gamma) \vdash \mathbf{rep}\ e \ : \ s_t([\,]\,\tau)$.

Case S-Def ($\Gamma \vdash \mathbf{def}\ f\ x = e\ ;\ p \ : \ \sigma'$)**:**
We have $\lceil 1 \rceil\ \Gamma, x : \tau \vdash e \ : \ \tau'$, $\lceil 2 \rceil\ \{\vec{\alpha}\} \cap \mathrm{ftv}(\Gamma, \sigma') = \emptyset$, and $\lceil 3 \rceil\ \Gamma, f : \forall \vec{\alpha}.\tau \to \tau' \vdash p \ : \ \sigma'$. Choose $\{\vec{\alpha'}\}$ such that $\lceil 4 \rceil\ \{\vec{\alpha'}\} \cap \mathrm{ftv}(s_t(\Gamma), s_t(\sigma), s_t) = \emptyset$ (where $\mathrm{ftv}(s_t) = \{\mathrm{ftv}(s_t(\beta)) \mid \beta \in \mathrm{dom}(s_t)\}$) and there exists a bijective substitution s_{rename} between $\{\vec{\alpha}\}$ and $\{\vec{\alpha'}\}$. Since type schemes are equal up to bound variable renaming, we have $\lceil 5 \rceil\ \Gamma, f : \forall \vec{\alpha'}.s_{\text{rename}}(\tau \to \tau') \vdash p \ : \ \sigma'$. Applying the IH to $\lceil 1 \rceil$ with the substitution $s_t \circ s_{\text{rename}}$ yields $\lceil 6 \rceil\ s_t(s_{\text{rename}}(\Gamma)), x : s_t(s_{\text{rename}}(\tau)) \vdash e \ : \ s_t(s_{\text{rename}}(\tau'))$. By $\lceil 2 \rceil$ this is equivalent to $\lceil 7 \rceil\ s_t(\Gamma), x : s_t(s_{\text{rename}}(\tau)) \vdash e \ : \ s_t(s_{\text{rename}}(\tau'))$. Applying the IH to $\lceil 5 \rceil$ with the substitution s_t yields $s_t(\Gamma), f : \forall \vec{\alpha'}.s_t|_{V_t \setminus \{\vec{\alpha'}\}}(s_{\text{rename}}(\tau) \to s_{\text{rename}}(\tau'))) \vdash p \ : \ s_t(\sigma')$ which is equivalent to $\lceil 8 \rceil\ s_t(\Gamma), f : \forall \vec{\alpha'}.s_t(s_{\text{rename}}(\tau)) \to s_t(s_{\text{rename}}(\tau'))) \vdash p \ : \ s_t(\sigma')$ by $\lceil 4 \rceil$. Applying S-Def to $\lceil 7 \rceil$, $\lceil 4 \rceil$, and $\lceil 8 \rceil$ yields $s_t(\Gamma) \vdash \mathbf{def}\ f\ x = e\ ;\ p \ : \ s_t(\sigma')$.

□

Proposition 2 (Unique Decomposition)**.** *If $\vdash p \ : \ \sigma$ then either p is a value or there exists a type scheme σ', a unique expression e, and a unique context K such that $p = K\langle e \rangle$ and $\vdash e \ : \ \sigma'$ and e is a redex.*

Proof. By induction over the typing derivation.

Case S-VAR**:**
Impossible because an empty context is assumed.

Case S-INST ($\vdash e : \tau$)**:**
We have $\lceil 1 \rceil \vdash e : \sigma$ and $\lceil 2 \rceil\ \sigma \geq \tau$. By the IH, there exists σ' and a unique redex e' and context K such that $e = K\langle e' \rangle$ and $\vdash e' : \sigma'$.

Case S-ARRAY ($\vdash [e_1, \ldots, e_n] : [\,]\tau$)**:**
We have $\lceil 1 \rceil\ \forall i \in \{1, \ldots, n\}\ .\ \vdash e_i : \tau$. Applying the IH to $\lceil 1 \rceil$, for all $i \in \{1, \ldots, n\}$ either e_i is a value or there exists σ_i along with a unique redex e_i' and context K_i such that $\lceil 2 \rceil\ e_i = K_i\langle e_i' \rangle$ and $\vdash e_i' : \sigma_i$. If every e_i is a value, we're done. Otherwise, choose the i where e_i isn't a value and set $K = [e_1, \ldots, e_{i-1}, C_i, e_{i+1}, \ldots, e_n]$. By $\lceil 2 \rceil$, $[e_1, \ldots, e_n] = K\langle e_i' \rangle$ and K is unique by the uniqueness of K_i and the definition of contexts.

Case S-VAL ($\vdash v : \tau$)**:**
v is a value.

Case S-FUN ($\vdash \lambda x.\, e : \tau \to \tau'$)**:**
$\lambda x.\, e$ is a value.

Case S-APP ($\vdash e_1\ e_2 : \tau_2$)**:**
We have $\lceil 1 \rceil \vdash e_1 : \tau_1 \to \tau_2$ and $\lceil 2 \rceil \vdash e_2 : \tau_1$.

Subcase e_1 isn't a value**:**
Applying the IH on $\lceil 1 \rceil$, there exists σ_1' and a unique redex e_1' and context K_1 such that $\lceil 3 \rceil\ e_1 = K_1\langle e_1' \rangle$ and $\lceil 4 \rceil \vdash e_1' : \sigma_1'$. Choosing $K = K_1\ e_2$, we have $e_1\ e_2 = K\langle e_1' \rangle$ as required.

Subcase e_1 is a value and e_2 isn't a value**:**
Analogous to the previous case, except $K = e_1\ K_2$ where K_2 is the context obtained by invoking the IH on $\lceil 2 \rceil$.

Subcase e_1 is a value and e_2 is a value**:**
Since $e_1\ e_2$ is a redex, we simply choose $K = \langle \cdot \rangle$.

Case S-MAP ($\vdash$ **map** $e_1\ e_2 : [\,]\tau_2$)**:**
The proof for this case proceeds nearly identically to the one for S-APP (except a **map**-context is chosen except for an application context).

Case S-REP ($\vdash$ **rep** $e : [\,]\tau$)**:**
We have $\lceil 1 \rceil \vdash e : \tau$. If e is a value, we're done. Otherwise, applying the IH to $\lceil 1 \rceil$, there exists σ' and a unique redex e' and context K' such that $\lceil 2 \rceil\ e = K'\langle e' \rangle$ and $\vdash e' : \sigma'$. By $\lceil 2 \rceil$. Choosing $K =$ **rep** C' we have $e = K\langle e' \rangle$.

Case S-DEF ($\vdash$ **def** $f\ x$ **=** e **;** $p : \sigma'$)**:**
This case is a redex, so we simply choose $K = \langle \cdot \rangle$.

□

Proposition 3 (Typing Closed Under Value Substitution)**.** *If* $\Gamma, x : \sigma' \vdash p : \sigma$ *and* $\vdash v : \sigma'$ *then* $\Gamma \vdash p[v/x] : \sigma$.

Proof. By induction over $\Gamma, x : \sigma' \vdash p : \sigma$. The SV- rules for values are skipped because they're either trivial or analogous to the corresponding expression rules.

Case S-Var ($\Gamma, y : \sigma \vdash y \ : \ \sigma$)**:**
If $x \neq y$ then $\Gamma, y : \sigma \vdash y[v/x] \ : \ \sigma$ is equivalent to $\Gamma, y : \sigma \vdash y \ : \ \sigma$, which clearly holds by S-Var. If $x = y$, then $\Gamma, y : \sigma \vdash y[v/x] \ : \ \sigma$ is equivalent to $\Gamma, x : \sigma \vdash v \ : \ \sigma$, which holds by assumption.

Case S-Inst ($\Gamma, x : \sigma' \vdash e \ : \ \tau$)**:**
We have $\lceil 1 \rceil \ \Gamma, x : \sigma' \vdash e \ : \ \sigma$ and $\lceil 2 \rceil \ \sigma \geq \tau$. By the IH, $\lceil 3 \rceil \ \Gamma, x : \sigma' \vdash e[v/x] \ : \ \sigma$ and applying S-Inst to $\lceil 2 \rceil$ and $\lceil 3 \rceil$ yields $\Gamma, x : \sigma' \vdash e[v/x] \ : \ \tau$.

Case S-Array ($\Gamma, x : \sigma' \vdash [e_1, \ldots, e_n] \ : \ [\,]\,\tau$)**:**
We have $\lceil 1 \rceil \ \forall i \in \{1, \ldots, n\} \,.\, \Gamma, x : \sigma' \vdash e_i \ : \ \tau$. Applying the IH to $\lceil 1 \rceil$, $\lceil 2 \rceil$ $\forall i \in \{1, \ldots, n\} \,.\, \Gamma, x : \sigma' \vdash e_i[v/x] \ : \ \tau$. Applying S-Array to $\lceil 2 \rceil$, $\Gamma, x : \sigma' \vdash [e_1[v/x], \ldots, e_n[v/x]]) \ : \ [\,]\,\tau$.

Case S-Val ($\Gamma, x : \sigma' \vdash v' \ : \ \tau$)**:**
$\Gamma, x : \sigma' \vdash v' \ : \ \tau$ is equal to $\Gamma, x : \sigma' \vdash v[v'/x] \ : \ \tau$ because values cannot contain free variables.

Case S-Fun ($\Gamma, x : \sigma' \vdash \lambda y. e \ : \ \tau \to \tau'$)**:**
If $y = x$, then $(\lambda y. e)[v/x]$ is equivalent to $\lambda y. e$ and we're done. Otherwise, assume $y \neq x$. We have $\lceil 1 \rceil$ which is equivalent to $\lceil 2 \rceil \ \Gamma, y : \tau, x : \sigma' \vdash e \ : \ \tau'$ (since $y \neq x$). Applying the IH to $\lceil 2 \rceil$, $\lceil 3 \rceil \ \Gamma, y : \tau, x : \sigma' \vdash e[v/x] \ : \ \tau'$, which is equivalent to $\lceil 4 \rceil \ \Gamma, x : \sigma', y : \tau \vdash e[v/x] \ : \ \tau'$. Applying S-Fun to $\lceil 4 \rceil$, $\Gamma, x : \sigma' \vdash \lambda y. (e[v/x]) \ : \ \tau \to \tau'$, which is equivalent to $\Gamma, x : \sigma' \vdash (\lambda y. e)[v/x] \ : \ \tau \to \tau'$.

Case S-App ($\Gamma, x : \sigma' \vdash e_1 \, e_2 \ : \ \tau_2$)**:**
We have $\lceil 1 \rceil \ \Gamma, x : \sigma' \vdash e_1 \ : \ \tau_1 \to \tau_2$ and $\lceil 2 \rceil \ \Gamma, x : \sigma' \vdash e_2 \ : \ \tau_1$. Applying the IH to $\lceil 1 \rceil$ and $\lceil 2 \rceil$, we obtain $\lceil 3 \rceil \ \Gamma, x : \sigma' \vdash e_1[v/x] \ : \ \tau_1 \to \tau_2$ and $\lceil 4 \rceil$ $\Gamma, x : \sigma' \vdash e_2[v/x] \ : \ \tau_1$. Applying S-App to $\lceil 3 \rceil$ and $\lceil 4 \rceil$, $\Gamma, x : \sigma' \vdash (e_1 \, e_2)[v/x] \ : \ \tau_2$.

Case S-Map ($\Gamma, x : \sigma' \vdash \texttt{map} \, e_1 \, e_2 \ : \ [\,]\tau_2$)**:**
We have $\lceil 1 \rceil \ \Gamma, x : \sigma' \vdash e_1 \ : \ \tau_1 \to \tau_2$ and $\lceil 2 \rceil \ \Gamma, x : \sigma' \vdash e_2 \ : \ [\,]\tau_1$. Applying the IH to $\lceil 1 \rceil$ and $\lceil 2 \rceil$, we obtain $\lceil 3 \rceil \ \Gamma, x : \sigma' \vdash e_1[v/x] \ : \ \tau_1 \to \tau_2$ and $\lceil 4 \rceil \ \Gamma, x : \sigma' \vdash e_2[v/x] \ : \ [\,]\tau_1$. Applying S-Map to $\lceil 3 \rceil$ and $\lceil 4 \rceil, \Gamma), x : \sigma' \vdash (\texttt{map} \, e_1 \, e_2)[v/x] \ : \ [\,]\tau_2$.

Case S-Rep ($\Gamma, x : \sigma' \vdash \texttt{rep} \, e \ : \ [\,]\,\tau$)**:**
We have $\lceil 1 \rceil \ \Gamma, x : \sigma' \vdash e \ : \ \tau$. Applying the IH to $\lceil 1 \rceil$, $\lceil 2 \rceil \ \Gamma, x : \sigma' \vdash e[v/x] \ : \ \tau$. Applying S-Rep to $\lceil 2 \rceil$, $\Gamma, x : \sigma' \vdash (\texttt{rep} \, e)[v/x] \ : \ [\,]\,\tau$.

Case S-Def ($\Gamma, x : \sigma' \vdash \texttt{def} \, f \, y = e \, ; \, p \ : \ \sigma'$)**:**
We have $\lceil 1 \rceil \ \Gamma, y : \tau \vdash e \ : \ \tau'$, $\lceil 2 \rceil \ \{\vec{\alpha}\} \cap \text{ftv}(\Gamma, \sigma') = \emptyset$, and $\lceil 3 \rceil \ \Gamma, f : \forall \vec{\alpha}.\tau \to \tau' \vdash p \ : \ \sigma'$. If $x = y$, $\Gamma, x : \sigma' \vdash (\texttt{def} \, f \, y = e \, ; \, p)[v/x] \ : \ \sigma'$ is equivalent to $\Gamma, x : \sigma' \vdash \texttt{def} \, f \, y = e \, ; \, p[v/x] \ : \ \sigma'$ which holds by applying the IH to $\lceil 2 \rceil$ to obtain $\lceil 4 \rceil \ \Gamma, x : \sigma', f : \forall \vec{\alpha}.\tau \to \tau' \vdash p[v/x] \ : \ \sigma'$ and then applying S-Def to $\lceil 1 \rceil$, $\lceil 2 \rceil$, and $\lceil 4 \rceil$. If $x \neq y$, then applying the IH to $\lceil 2 \rceil$ yields $\lceil 5 \rceil \ \Gamma, y : \tau, x : \sigma' \vdash e[v/x] \ : \ \tau'$, from which $\Gamma, x : \sigma' \vdash (\texttt{def} \, f \, y = e \, ; \, p)[v/x] \ : \ \sigma'$ follows by applying S-Def to $\lceil 5 \rceil$, $\lceil 2 \rceil$, and $\lceil 4 \rceil$.

□

Proposition 4 (Progress)**.** *If* $\vdash p : \sigma$ *then either* p *is a value or there exists* p' *such that* $p \rightsquigarrow p'$.

Proof. If p isn't a value, by Proposition 2 there exists a unique e' and K such that $e = K\langle e' \rangle$, $\vdash e' \ : \ \sigma'$, and e' is a redex. Because e' is a redex, there exists e'' such that $e' \rightsquigarrow e''$. Hence, $e = K\langle e' \rangle \rightsquigarrow K\langle e'' \rangle$. □

Proposition 5 (Preservation). *If* $\vdash p : \sigma$ *and* $p \rightsquigarrow p'$ *then* $\vdash p' : \sigma$.

Proof. By induction over the typing derivation.

Case S-Var**:**
Impossible because an empty context is assumed.

Case S-Inst ($\vdash e \ : \ \tau$)**:**
We have $\lceil 1 \rceil \vdash e \ : \ \sigma$, $\lceil 2 \rceil \ \sigma \geq \tau$, and $\lceil 4 \rceil \ e \rightsquigarrow e'$. Applying the IH, to $\lceil 1 \rceil$ and $\lceil 4 \rceil$ yields $\lceil 5 \rceil \vdash e' \ : \ \sigma$. Applying S-Inst to $\lceil 5 \rceil$ and $\lceil 2 \rceil$ yields $\vdash e' \ : \ \tau$

Case S-Array ($\vdash [e_1, \ldots, e_n] \ : \ [\,]\,\tau$)**:**
We have $\lceil 1 \rceil \ \forall i \in \{1, \ldots, n\} \, . \ \vdash e_i \ : \ \tau$ By Proposition 2, there is a σ_i, a unique context K, and unique expressions e, e' such that $\lceil 3 \rceil \ [e_1, \ldots, e_n] = K\langle e \rangle$, $\lceil 4 \rceil \vdash e \ : \ \sigma'$, $\lceil 5 \rceil \ e \rightsquigarrow e'$, and $\lceil 6 \rceil \ K\langle e \rangle \rightsquigarrow K\langle e' \rangle$. From the reduction rules, we have $\lceil 7 \rceil \ K = [e_1, \ldots, e_{i-1}, K', e_{i+1}, \ldots, e_n]$ where $\lceil 8 \rceil \ e_i = K'\langle e \rangle$. By $\lceil 6 \rceil$ and $\lceil 7 \rceil$, $\lceil 9 \rceil$ $K'\langle e \rangle \rightsquigarrow K'\langle e' \rangle$. Applying the IH using $\lceil 1 \rceil$ and $\lceil 9 \rceil$, we have $\lceil 10 \rceil \vdash K'\langle e' \rangle \ : \ \sigma_i$. By $\lceil 3 \rceil$, $\lceil 10 \rceil$, and $\lceil 7 \rceil$ we have $\vdash K\langle e' \rangle \ : \ [\,]\,\tau$.

Case S-Val ($\vdash v \ : \ \tau$)**:**
v is a value, so no reduction is possible.

Case S-Fun ($\vdash \lambda x . e \ : \ \tau \to \tau'$)**:**
$\lambda x . e$ is a function, so no reduction is possible.

Case S-App ($\vdash e_1 \ e_2 \ : \ \tau_2$)**:**
We have $\lceil 1 \rceil \vdash e_1 \ : \ \tau_1 \to \tau_2$ and $\lceil 2 \rceil \vdash e_2 \ : \ \tau_1$.

Subcase e_1 isn't a value**:**
By Proposition 2, there is a σ, a unique context K, and unique expressions e, e' such that $\lceil 3 \rceil \ e_1 \ e_2 = K\langle e \rangle$, $\lceil 4 \rceil \vdash e \ : \ \sigma'$, $\lceil 5 \rceil \ e \rightsquigarrow e'$, and $\lceil 6 \rceil \ K\langle e \rangle \rightsquigarrow K\langle e' \rangle$. Since e_1 isn't a value, by the uniqueness of K, there exists unique K' such that $\lceil 7 \rceil \ K = K' \ e_2$, $\lceil 8 \rceil \ e_1 = K'\langle e \rangle$, and $\lceil 9 \rceil \ K'\langle e \rangle \rightsquigarrow K'\langle e' \rangle$. Applying the IH to $\lceil 1 \rceil$ and $\lceil 9 \rceil$ yields $\lceil 10 \rceil \vdash K'\langle e' \rangle \ : \ \tau_1 \to \tau_2$. Applying S-App to $\lceil 9 \rceil$ and $\lceil 2 \rceil$ yields $\vdash K\langle e' \rangle \ : \ \tau_2$.

Subcase e_1 is a value and e_2 isn't**:**
Analogous to the previous case with $K = e_1 \ K'$.

Subcase e_1 and e_2 are values**:**
In this case, $e_1 \ e_2$ is a redex and the property follows by Proposition 3.

Case S-Map ($\Gamma \vdash$ **map** $e_1 \ e_2 \ : \ [\,]\tau_2$)**:**
Analogous to S-App except with a **map** context.

Case S-Rep ($\Gamma \vdash$ **rep** $e \ : \ [\,]\,\tau$)**:**
We have $\lceil 1 \rceil \ \Gamma \vdash e \ : \ \tau$. If e is a value, there's nothing to show, so assume e isn't a value. The proof proceeds similarly to the other cases, just with the context $K =$ **rep** K'.

Case S-Def ($\Gamma \vdash$ **def** $f \ x$ **=** e ; $p \ : \ \sigma'$)**:**
A **def** is always a redex and the property follows by Proposition 3.

□

A.6 Rank Analysis

A.6.5 Constraint Set Solving

In this section, we prove Proposition 6 and Proposition 7 from chapter 4. To prove the propositions, we introduce the notion of a *closed rank*. The *closed rank* of a closed shape or closed type, written $|\cdot|^{\oslash}$, denotes the non-negative integral rank of a closed shape ($S^{\oslash}$) or a closed type ($\tau^{\oslash}$). It is defined as

$$\begin{array}{ll} |[\,]|^{\oslash} = 1, & |S\ \tau|^{\oslash} = |S|^{\oslash} + |\tau|^{\oslash}, \\ |S_1\ S_2|^{\oslash} = |S_1|^{\oslash} + |S_2|^{\oslash}, & |\alpha|^{\oslash} = 0, \\ |\bullet|^{\oslash} = 0, & |\tau_1 \to \tau_2|^{\oslash} = 0, \\ & |\mathtt{int}|^{\oslash} = 0. \end{array}$$

Lemma 2. *Given a closed substitution s and rank substitution s_{r}, if $s_{\mathrm{r}}(Q) = s(Q)$ and $s_{\mathrm{r}}(\overline{\alpha}) = |s(\alpha)|^{\oslash}$ then $|s(S)|^{\oslash} = s_{\mathrm{r}}(|S|)$ and $|s(\tau)|^{\oslash} = s_{\mathrm{r}}(|\tau|)$.*

Proof. The first statement, $\lceil 1 \rceil\ |s(S)|^{\oslash} = s_{\mathrm{r}}(|S|)$, follows by induction over S.

Case closed shapes ($S^{\oslash}$)**:**
Immediate, since $|s(S^{\oslash})|^{\oslash} = |S^{\oslash}| = s_{\mathrm{r}}(|S^{\oslash}|)$.

Case rank power ($[\,]^{Q}$)**:**
We have $|s([\,]^{Q})|^{\oslash} = s(Q) = s_{\mathrm{r}}(Q) = s_{\mathrm{r}}(|[\,]^{Q}|)$.

Case concatenation ($S_1\ S_2$)**:**
We have $|s(S_1 S_2)|^{\oslash} = |s(S_1)|^{\oslash} + |s(S_2)|^{\oslash} = s_{\mathrm{r}}(|S_1|) + s_{\mathrm{r}}(|S_2|) = s_{\mathrm{r}}(|S_1\ S_2|)$ by the IH.

The second statement, $|s(\tau)|^{\oslash} = s_{\mathrm{r}}(|\tau|)$, follows by induction over τ, using the first statement.

Case function type ($\tau_1 \to \tau_2$)**:**
We have $|s(\tau_1 \to \tau_2)|^{\oslash} = |s(\tau_1) \to s(\tau_2)|^{\oslash} = 0 = s_{\mathrm{r}}(0) = s_{\mathrm{r}}(|\tau_1 \to \tau_2|)$.

Case array type ($S'\tau'$)**:**
By $\lceil 1 \rceil$, we have $\lceil 2 \rceil\ |s(S')|^{\oslash} = s_{\mathrm{r}}(|S'|)$. By the IH, we have $\lceil 3 \rceil\ |s(\tau')|^{\oslash} = s_{\mathrm{r}}(|\tau'|)$. Combining $\lceil 2 \rceil$ and $\lceil 3 \rceil$ gives $|s(S'\tau')|^{\oslash} = |s(S')|^{\oslash} + |s(\tau')|^{\oslash} = s_{\mathrm{r}}(|S'|) + s_{\mathrm{r}}(|\tau'|) = s_{\mathrm{r}}(|S'\tau'|)$.

Case integer (`int`)**:**
Trivial.

Case closed function type ($\tau_1^{\oslash} \to \tau_2^{\oslash}$)**:**
Analogous to the function type case.

Case closed array type ($S^{\oslash\prime}\tau^{\oslash\prime}$)**:**
Analogous to the array type case.

Case type variable (α)**:**
We have $|s(\alpha)|^{\oslash} = s_{\mathrm{r}}(\overline{\alpha}) = s_{\mathrm{r}}(|\alpha|)$.

□

Lemma 3. *If s satisfies C then the rank substitution s_{r} defined by $s_{\mathrm{r}}(Q) = s(Q)$ and $s_{\mathrm{r}}(\overline{\alpha}) = |s(\alpha)|^{\oslash}$ satisfies $|C|$.*

Proof. Case analysis on the constraints of C. Since s satisfies C, note that $s|_{\mathrm{ftv}(C)}$ must be closed.

Case $M \vee R$**:**
Immediate by the definition of s_r.

Case $\tau_1 \doteq \tau_2$**:**
By Lemma 2, $|s(S)|^\varnothing = s_\mathrm{r}(|S|)$ and $|s(\tau)|^\varnothing = s_\mathrm{r}(|\tau|)$. Hence, $s_\mathrm{r}(|\tau_1|) = |s(\tau_1)|^\varnothing = |s(\tau_2)|^\varnothing = s_\mathrm{r}(|\tau_2|)$, as required.

□

Proposition 6. *If s satisfies C, there exists a rank substitution s_r that satisfies $|C|$ and there exists a closed type substitution s_t such that $s|_{\mathrm{ftv}(C)\cup\mathrm{frv}(C)} = s_\mathrm{t} \circ s_\mathrm{r}$.*

Proof. By Lemma 3, the rank substitution s_r defined by $s_\mathrm{r}(Q) = s(Q)$ and $s_\mathrm{r}(\overline{\alpha}) = |s(\alpha)|^\varnothing$ satisfies $|C|$. Define $s_\mathrm{t} = s|_{\mathrm{ftv}(C)}$. Since s satisfies C, s_t must be closed (i.e., s_t does not map any type variables to types with rank variables). Now, $(s_\mathrm{t} \circ s_\mathrm{r})(Q) = s_\mathrm{r}(Q) = s(Q)$. Additionally, $(s_\mathrm{t} \circ s_\mathrm{r})(\alpha) = s_\mathrm{t}(\alpha) = s|_{\mathrm{ftv}(C)}(\alpha)$ for each $\alpha \in \mathrm{ftv}(C)$. □

Lemma 4. *Consider a constraint set C and suppose s satisfies C and s_r satisfies $|C|$. Define*

$$\begin{array}{lcllcl} s'(Q) & = & s_\mathrm{r}(Q), & \mathrm{basetype}(S\tau) & = & \mathrm{basetype}(\tau), \\ s'(\alpha) & = & [\,]^{s_\mathrm{r}(\overline{\alpha})}\mathrm{basetype}(s(\alpha)), & \mathrm{basetype}(\tau) & = & \tau. \end{array}$$

Then s' satisfies C.

Proof. By case analysis over the constraints of C.

Case $M \vee R$**:**
Immediate by the definition of s'.

Case $\tau_1 \doteq \tau_2$**:**
We need to show that $\lceil 1 \rceil\ \mathrm{basetype}(s'(\tau_1)) = \mathrm{basetype}(s'(\tau_2))$ and $\lceil 2 \rceil\ |s'(\tau_1)|^\varnothing = |s'(\tau_2)|^\varnothing$, since this implies $s'(\tau_1) = s'(\tau_2)$. $\lceil 1 \rceil$ is immediate by the definition of s' and basetype. Note that s' must be closed over C because s_r satisfies $|C|\mathrm{r}$ and s satisfies C. We have $\lceil 3 \rceil\ s_\mathrm{r}(Q) = s'(Q)$ and $\lceil 4 \rceil\ s_\mathrm{r}(\overline{\alpha}) = |[\,]^{s_\mathrm{r}(\overline{\alpha})}\mathrm{basetype}(s(\alpha))|^\varnothing = |s'(\alpha)|^\varnothing$. Applying $\lceil 3 \rceil$ and $\lceil 4 \rceil$ to Lemma 2 yields $s_\mathrm{r}(|\tau|) = |s'(\tau)|^\varnothing$ for any τ, and hence $\lceil 2 \rceil$ reduces to $s_\mathrm{r}(|\tau_1|) = s_\mathrm{r}(|\tau_2|)$, which follows by the fact that s_r satisfies $|C|$.

□

Proposition 7. *If C is satisfiable and s_r satisfies $|C|$ then there is a closed type substitution s_t such that the substitution $s = s_\mathrm{t} \circ s_\mathrm{r}$ satisfies C.*

Proof. Using the construction from Lemma 4, there is a satisfier s' of C with $s'(Q) = s_\mathrm{r}(Q)$ and $|s'(\alpha)|^\varnothing = s_\mathrm{r}(\overline{\alpha})$. Because s' satisfies C, $s'|_{\mathrm{ftv}(C)}$ is closed. Defining $s_\mathrm{t} = s'|_{\mathrm{ftv}(C)}$, we have $s(\alpha) = (s_\mathrm{t} \circ s_\mathrm{r})(\alpha) = s_\mathrm{t}(\alpha) = s'(\alpha)$ for each $\alpha \in \mathrm{ftv}(C)$ and hence $s = s_\mathrm{t} \circ s_\mathrm{r}$ satisfies C. □

A.7 Transformation to the Target Language

A.7.1 Well-Typedness

Lemma 5. *If $\sigma \geq \tau$ then $S\ \sigma \geq S\ \tau$.*

Proof. Let $\sigma = \forall \vec{\alpha}.\tau'$. By definition, there exists a type substitution s_t such that $\mathrm{dom}(s_t) = \{\vec{\alpha}\}$ and $s_t(\tau') = \tau$. But then $Ss_t(\tau') = S\tau$ and hence $s_t(S\ \tau') = S\tau$ because s_t has no effect on shapes. □

Proposition 8 (Well-Typedness for Expressions)**.** *If $\Gamma \vdash e :_S \sigma \parallel C$ and s is a satisfier of C, then $s(\Gamma) \vdash \mathrm{AM}(s(e)) : s(S\ \sigma)$*

Proof. By induction over the typing derivation.

Case C-Int ($\vdash n : \texttt{int} \parallel \emptyset$)**:**
Immediate by the definition of AM.

Case C-Inst ($\Gamma \vdash e : \tau \parallel \emptyset$)**:**
We have $\lceil 1 \rceil\ \Gamma \vdash e : \sigma \parallel \emptyset$ and $\lceil 2 \rceil\ \sigma \geq \tau$. By the IH, $\lceil 3 \rceil\ s(\Gamma) \vdash \mathrm{AM}(s(e)) : s(S\ \sigma)$. Since $\geq$ is closed, applying s to $\lceil 2 \rceil$ yields $\lceil 4 \rceil\ s(\sigma) \geq s(\tau)$. By Lemma 5 and $\lceil 4 \rceil$, $\lceil 5 \rceil\ s(S\sigma) \geq s(S\tau)$. Applying S-Inst to $\lceil 3 \rceil$ and $\lceil 5 \rceil$ yields $s(\Gamma) \vdash \mathrm{AM}(s(e)) : s(S\ \tau)$ as required.

Case C-Var ($\Gamma, x : \sigma \vdash x : \sigma \parallel \emptyset$)**:**
Immediate by the definition of AM.

Case C-Array ($\Gamma \vdash [e_1, e_2, \ldots, e_n] : [\,]S_1\tau_1 \parallel \{S_1\tau_1 \doteq S_k\tau_k \mid k \in \{2, \ldots, n\}\} \cup C_1 \cup \cdots \cup C_n$)**:**

We have $\lceil 1 \rceil\ \forall k \in \{1, \ldots, n\}\ .\ \Gamma \vdash e_k :_{S_k} \tau_k \parallel C_k$. Applying the IH to $\lceil 1 \rceil$ yields $\lceil 2 \rceil$ $\forall k \in \{1, \ldots, n\}\ .\ s(\Gamma) \vdash \mathrm{AM}(s(e_k)) : s(S_k\tau_k)$. Since s satisfies C, $s(S_k\tau_k) = s(S_1\tau_1)$ for all $k \in \{1, \ldots, n\}$ and hence $\lceil 3 \rceil\ \forall k \in \{1, \ldots, n\}\ .\ s(\Gamma) \vdash \mathrm{AM}(s(e_k)) : s(S_1\tau_1)$. Applying S-Array to $\lceil 3 \rceil$ yields $s(\Gamma) \vdash \mathrm{AM}([e_1, \ldots, e_n]) : s([\,]S_1\tau_1)$.

Case C-App ($\Gamma \vdash e_1\ e_2 \triangle (M, R) :_{[\,]^M_{S_1}} \tau_2 \parallel C \cup C_1 \cup C_2$)**:**
We have $\lceil 1 \rceil\ \Gamma \vdash e_1 :_{S_1} \tau_1 \to \tau_2 \parallel C_1$, $\lceil 2 \rceil\ \Gamma \vdash e_2 :_{S_2} \tau_3 \parallel C_2$, and $\lceil 3 \rceil$ $C = \{M \curlyvee R,\ [\,]^M\ S_1\ \tau_1 \doteq [\,]^R\ S_2\ \tau_3\}$. Applying the IH to $\lceil 1 \rceil$ and $\lceil 2 \rceil$, we have $\lceil 4 \rceil$ $s(\Gamma) \vdash \mathrm{AM}(s(e_1)) : s(S_1\tau_1) \to s(S_1\tau_2)$ and $\lceil 5 \rceil\ s(\Gamma) \vdash \mathrm{AM}(s(e_2)) : s(S_2\tau_3)$. Since s satisfies C, we also have $\lceil 6 \rceil\ s(M) = 0$ or $s(R) = 0$ and $\lceil 7 \rceil\ s([\,]^M\ S_1\ \tau_1) = s([\,]^R\ S_2\ \tau_3)$. We now case on $\lceil 6 \rceil$.

Subcase $s(R) = 0$**:**
By $\lceil 7 \rceil$, we have $\lceil 8 \rceil\ s([\,]^M S_1\ \tau_1) = s(S_2\ \tau_3)$. By $\lceil 5 \rceil$ and $\lceil 8 \rceil$ we have $\lceil 9 \rceil\ s(\Gamma) \vdash \mathrm{AM}(s(e_2)) : s([\,]^M S_1\ \tau_1)$. Starting with $\lceil 4 \rceil$ and $\lceil 9 \rceil$ and applying the S-Map rule M times, we have $s(\Gamma) \vdash \mathbf{map}^s(M)\ \mathrm{AM}(s(e_1))\ \mathrm{AM}(s(e_2)) : s([\,]^M S_1\ \tau_2)$ which, by the definition of AM, is the same as $s(\Gamma) \vdash \mathrm{AM}(s(e_1\ e_2 \triangle (M, R))) : s([\,]^M S_1\ \tau_2)$.

Subcase $s(M) = 0$**:**
By $\lceil 7 \rceil$, we have $\lceil 8 \rceil\ s(S_1\ \tau_1) = s([\,]^R\ S_2\ \tau_3)$. By $\lceil 4 \rceil$ and $\lceil 8 \rceil$ we have $\lceil 9 \rceil\ s(\Gamma) \vdash \mathrm{AM}(s(e_1)) : s([\,]^R\ S_2\ \tau_3) \to s(S_1\tau_2)$. Starting with $\lceil 5 \rceil$ and applying the S-Rep rule R times, we have $\lceil 10 \rceil\ s(\Gamma) \vdash \mathbf{rep}^R\ \mathrm{AM}(s(e_2)) : s([\,]^R S_2\tau_3)$. Applying S-App to $\lceil 9 \rceil$ and $\lceil 10 \rceil$ yields $s(\Gamma) \vdash \mathrm{AM}(s(e_1))\ (\mathbf{rep}^R\ \mathrm{AM}(s(e_2))) : s(S_1\tau_2)$ which is the same as $s(\Gamma) \vdash \mathrm{AM}(s(e_1\ e_2 \triangle (M, R))) : s(S_1\tau_2)$ by the definition of AM.

Case C-Fun ($\Gamma \vdash \lambda x.\, e \;:\; \tau_1 \to S\ \tau_2 \parallel C$)**:**
We have $\lceil 1 \rceil\ \Gamma, x : \tau_1 \vdash e \;:_S\; \tau_2 \parallel C$. Applying the IH to $\lceil 1 \rceil$, $\lceil 2 \rceil\ s(\Gamma, x : \tau_1) \vdash \mathrm{AM}(s(e)) \;:\; s(S\tau_2)$ Applying S-Fun to $\lceil 2 \rceil$, $s(\Gamma) \vdash \mathrm{AM}(s(\lambda x.\, e)) \;:\; s(\tau_1) \to s(S\tau_2)$

Case C-Map ($\Gamma \vdash \mathtt{map}\ e_1\ e_2 \;:_{[\,]S_1}\; \tau_2 \parallel \{[\,]S_1\tau_1 \doteq S_2\tau_3\} \cup C_1 \cup C_2$)**:**
We have $\lceil 1 \rceil\ \Gamma \vdash e_1 \;:_{S_1}\; \tau_1 \to \tau_2 \parallel C_1$ and $\lceil 2 \rceil\ \Gamma \vdash e_2 \;:_{S_2}\; \tau_3 \parallel C_2$. Applying the IH to $\lceil 1 \rceil$ and $\lceil 2 \rceil$, we have $\lceil 3 \rceil\ s(\Gamma) \vdash \mathrm{AM}(s(e_1)) \;:\; s(S_1\tau_1) \to s(S_1\tau_2)$ and $\lceil 4 \rceil\ s(\Gamma) \vdash \mathrm{AM}(s(e_2)) \;:\; s(S_2\tau_3)$. Since $s([\,]S_1\tau_1) = s(S_2\tau_3)$, by $\lceil 4 \rceil$ we have $\lceil 5 \rceil\ s(\Gamma) \vdash \mathrm{AM}(s(e_2)) \;:\; s([\,]S_1\tau_1)$. Applying S-Map to $\lceil 3 \rceil$ and $\lceil 5 \rceil$, $s(\Gamma) \vdash \mathtt{map}\ \mathrm{AM}(s(e_1))\ \mathrm{AM}(s(e_2)) \;:\; [\,]s(S_1\tau_2)$, which is the same as $s(\Gamma) \vdash \mathrm{AM}(s(\mathtt{map}\ e_1\ e_2)) \;:\; s([\,]S_1\tau_2)$.

Case C-Rep ($\Gamma \vdash \mathtt{rep}\ e \;:\; [\,]S\tau \parallel C$)**:**
We have $\lceil 1 \rceil\ \Gamma \vdash e \;:_S\; \tau \parallel C$. Applying the IH to $\lceil 1 \rceil$, we have $\lceil 2 \rceil\ s(\Gamma) \vdash \mathrm{AM}(s(e)) \;:\; s(S\tau)$. Applying S-Rep to $\lceil 2 \rceil$, we have $s(\Gamma) \vdash \mathtt{rep}\ \mathrm{AM}(s(e)) \;:\; [\,]s(S\tau)$, which is the same as $s(\Gamma) \vdash \mathrm{AM}(s(\mathtt{rep}\ e)) \;:\; s([\,]S\tau)$.

□

Proposition 9 (Well-Typedness)**.** *If* $\Gamma \vdash p \;:\; \sigma$ *then* $\Gamma \vdash \mathrm{AM}(p) \;:\; \sigma$.

Proof. By induction on $\Gamma \vdash p \;:\; \sigma$.

Case C-Def ($\Gamma \vdash \mathtt{def}\ f\ x = s(e)\ ;\ p \;:\; \sigma'$)**:**
We have $\lceil 1 \rceil\ \Gamma, x : \tau \vdash e \;:\; \tau' \parallel C$, $\lceil 2 \rceil\ s$ satisfies C, $\lceil 3 \rceil\ \{\vec{\alpha}\} \cap \mathrm{ftv}(s(\Gamma), \sigma') = \emptyset$, $\lceil 4 \rceil\ s(\Gamma), f : \forall\vec{\alpha}.s(\tau) \to s(\tau') \vdash p \;:\; \sigma'$. Applying the IH on $\lceil 4 \rceil$ yields $\lceil 5 \rceil\ s(\Gamma), f : \forall\vec{\alpha}.s(\tau) \to s(\tau') \vdash \mathrm{AM}(p) \;:\; \sigma'$ and applying Proposition 8 on $\lceil 1 \rceil$ using $\lceil 2 \rceil$ yields $\lceil 6 \rceil\ s(\Gamma, x : \tau) \vdash \mathrm{AM}(s(e)) \;:\; s(\tau')$. Applying S-Def to $\lceil 6 \rceil$, $\lceil 3 \rceil$, and $\lceil 5 \rceil$ yields $s(\Gamma) \vdash \mathtt{def}\ f\ x = \mathrm{AM}(s(e))\ ;\ \mathrm{AM}(p) \;:\; \sigma'$.

□

A.7.2 Backwards Consistency

Lemma 6. *If* $\Gamma \vdash \mathcal{K}\langle e \rangle \;:_S\; \sigma \parallel C$ *then there exists* σ', S', *and* C' *such that* $\Gamma \vdash e \;:_{S'}\; \sigma' \parallel C'$.

Proof. By induction on $\mathcal{K}$. We only show a couple of cases; the remaining cases are analogous and follow by the IH.

Case $\mathcal{K} = \langle\cdot\rangle$**:**
Immediate.

Case $\mathcal{K} = \mathcal{K}'\ e_2 \vartriangle (M, R)$**:**
We have $\lceil 1 \rceil\ \Gamma \vdash \mathcal{K}'\langle e \rangle\ e_2 \vartriangle (M, R) \;:_S\; \sigma \parallel C$. By inversion of C-App, there exists τ_1, τ_2, C_1, and S_1 such that $\lceil 2 \rceil\ \Gamma \vdash \mathcal{K}'\langle e \rangle \;:_{S_1}\; \tau_1 \to \tau_2 \parallel C_1$. Applying the IH to $\lceil 2 \rceil$ yields the required result.

□

Proposition 10 (Removal Well-Typedness)**.** *If* $\mathcal{K}\langle e \rangle \prec_{rem} \mathcal{K}\langle e' \rangle$ *and* $\Gamma \vdash \mathcal{K}\langle e \rangle \;:_S\; \sigma \parallel C$ *then there exists* S', σ', *and* C' *such that*

(a) $\Gamma \vdash \mathcal{K}\langle e' \rangle \;:_{S'}\; \sigma' \parallel C'$.

(b) If s_r *satisfies* $|C|$, *then there exists* $\hat{s}'_r$ *such that* $s_r \circ \hat{s}'_r$ *satisfies* $|C'|$.

(c) $s_{\mathrm{r}}(C) \simeq (s_{\mathrm{r}} \circ \hat{s}'_{\mathrm{r}})(C')$.

Proof. By inversion on the removal relation, we have $\lceil 1 \rceil$ M, R fresh. We proceed by induction on $\mathcal{K}$.

Case $\mathcal{K} = \langle \cdot \rangle$**:**

We proceed by cases on the removal relation.

Subcase Rem-Map (**map** $e_1\ e_2 \prec_{\mathrm{rem}} e_1\ e_2 \vartriangle (M, R)$)**:**

We have $\lceil 2 \rceil$ **map** $e_1\ e_2 \prec_{\mathrm{rem}} e_1\ e_2 \vartriangle (M, R)$ and $\lceil 3 \rceil\ \Gamma \vdash$ **map** $e_1\ e_2 : S\ \sigma \parallel C$. By inversion of $\lceil 3 \rceil$, there exists $\tau_1, \tau_2, C_1, C_2, S_2$ such that $\lceil 4 \rceil\ \Gamma \vdash e_1 : S\ \tau_1 \to \tau_2 \parallel C_1, \lceil 5 \rceil\ \Gamma \vdash e_2 : S_2\ \tau_3 \parallel C_2, \lceil 6 \rceil\ C = \{[\,]S\tau_1 \doteq S_2\tau_3\} \cup C_1 \cup C_2$, and $\lceil 7 \rceil$ $\tau_2 = \sigma$. Define $\lceil 8 \rceil\ C'' = \{M \forall R, [\,]^M\ S\ \tau_1 \doteq [\,]^R\ S_2\ \tau_2\}$.

(a) Applying C-App to $\lceil 1 \rceil$, $\lceil 4 \rceil$, $\lceil 5 \rceil$, and $\lceil 8 \rceil$ yields $\lceil 9 \rceil\ \Gamma \vdash e_1\ e_2 \vartriangle (M, R) : [\,]^M S\ \tau_2 \parallel C'' \cup C_1 \cup C_2$, which is the same as $\Gamma \vdash \mathcal{K}\langle e' \rangle : S'\ \sigma' \parallel C'$ with $e' = e_1\ e_2 \vartriangle (M, R)$, $S' = [\,]^M S$, $\sigma' = \tau_2$, and $C' = C'' \cup C_1 \cup C_2$.

(b) Suppose s_{r} satisfies $|C|$. The only difference between C and C' is that C contains $\{[\,]S\tau_1 \doteq S_2\tau_3\}$ whereas C' contains $\{M \forall R, [\,]^M\ S\ \tau_1 \doteq [\,]^R\ S_2\ \tau_2\}$. Define $\hat{s}'_{\mathrm{r}} = [M \mapsto 1, R \mapsto 0]$. Then, $\lceil 10 \rceil$ $\hat{s}'_{\mathrm{r}}(|\{M \forall R, [\,]^M\ S\ \tau_1 \doteq [\,]^R\ S_2\ \tau_2\}|) = \{1 \forall 0, |[\,]^M\ S\ \tau_1 \doteq [\,]^R\ S_2\ \tau_2|\}$, which is clearly satisfied by s_{r}.

(c) By $\lceil 10 \rceil$, the only difference between $s_{\mathrm{r}}(C)$ and $(s_{\mathrm{r}} \circ \hat{s}'_{\mathrm{r}})(C')$ is that the latter contains the additional variable-free constraint $1 \forall 0$, which is trivially satisfied by any substitution.

Subcase Rem-Rep (**rep** $e \prec_{\mathrm{rem}} (\lambda x.\, x)\ e \vartriangle (M, R)$)**:**

We have $\lceil 2 \rceil$ **rep** $e \prec_{\mathrm{rem}} (\lambda x.\, x)\ e \vartriangle (M, R)$ and $\lceil 3 \rceil\ \Gamma \vdash$ **rep** $e : \sigma \parallel C$. By inversion of $\lceil 3 \rceil$, there exists τ and S_1 with $\sigma = [\,]S_1\tau$ such that $\lceil 4 \rceil$ $\Gamma \vdash e : S_1\ \tau \parallel C$. By C-Fun, we have $\lceil 5 \rceil\ \Gamma \vdash \lambda x.\, x : \alpha \to \alpha \parallel \emptyset$, where α can be chosen fresh. Define $\lceil 6 \rceil\ C'' = \{M \forall R, [\,]^M\ \alpha \doteq [\,]^R\ S_1\ \tau\}$.

(a) Applying C-App to $\lceil 1 \rceil$, $\lceil 4 \rceil$, $\lceil 5 \rceil$, and $\lceil 6 \rceil$ yields $\lceil 9 \rceil\ \Gamma \vdash \mathtt{id}\, e \vartriangle (M, R) : [\,]^M\ \alpha \parallel C'' \cup C$, as required.

(b) Suppose s_{r} satisfies $|C|$. The only difference between C and C' is that C' additionally contains the constraints of $\lceil 6 \rceil$ (i.e., $\{M \forall R, [\,]^M\ \alpha \doteq [\,]^R\ S_1\ \tau\}$). Define $\hat{s}'_{\mathrm{r}} = [M \mapsto 0, R \mapsto 1, \overline{\alpha} \mapsto |[\,]\ S_1\ \tau|]$. Then, $\lceil 10 \rceil$ $\hat{s}'_{\mathrm{r}}(|\{M \forall R, [\,]^M\ \alpha \doteq [\,]^R\ S_1\ \tau\})| = \{0 \forall 1, \overline{\alpha} \doteq |[\,]\ S_1\ \tau|\}$, which is obviously satisfied by s_{r}.

(c) Since α can always be chosen uniquely fresh, we assume that it only appears in the constraint $\alpha \doteq [\,]\ S_1\ \tau$ (and hence can be trivially satisfied by the mapping $\alpha \mapsto [\,]\ S_1\ \tau$). Since both constraints of $\lceil 10 \rceil$ are always satisfiable, we conclude that C and C' are equivalent.

Case $\mathcal{K} =$ **rep** $\mathcal{K}'$**:**

We have $\lceil 2 \rceil$ **rep** $\mathcal{K}'\langle e \rangle \prec_{\mathrm{rem}}$ **rep** $\mathcal{K}'\langle e' \rangle$ and $\lceil 3 \rceil\ \Gamma \vdash$ **rep** $\mathcal{K}'\langle e \rangle : S\ \sigma \parallel C$. $\lceil 2 \rceil$ implies $\lceil 4 \rceil$ $\mathcal{K}'\langle e \rangle \prec_{\mathrm{rem}} \mathcal{K}'\langle e' \rangle$ by the definition of the removal relation. By Lemma 6, there exists S', σ', and C' such that $\lceil 5 \rceil\ \Gamma \vdash e : S'\ \sigma' \parallel C'$.

(a) Applying the IH to $\lceil 4 \rceil$ and $\lceil 5 \rceil$, there exists S'', σ'', and C'' such that $\lceil 5 \rceil\ \Gamma \vdash \mathcal{K}'\langle e' \rangle : S''\ \sigma'' \parallel C''$. Applying C-Rep to $\lceil 5 \rceil$ yields $\Gamma \vdash$ **rep** $\mathcal{K}'\langle e' \rangle$ **:** $[\,]\ S''\ \sigma'' \parallel C''$.

(b) Suppose s_r satisfies $|C|$. Applying the IH to $\lceil 5 \rceil$ says that there exists $\hat{s}'_r$ such that $s_r \circ \hat{s}'_r$ satisfies $|C''|$.

(c) Immediate by the inductive hypothesis since the **rep** constructor does not augment the constraint set.

The remaining context cases are analogous to the $\mathcal{K} = \mathbf{rep}\ \mathcal{K}'$ case and are proved by invoking the inductive hypothesis. □

Proposition 11 (Backwards Consistency)**.** *If* $\mathcal{K}\langle e \rangle \prec_{rem} \mathcal{K}\langle e' \rangle$, $\Gamma \vdash \mathcal{K}\langle e \rangle :_S \sigma \parallel C$, *and* s_r *is unambiguous at size* k *for* $|C|$*, then there exists* S', σ', C', s'_r *such that* $\Gamma \vdash \mathcal{K}\langle e' \rangle :_{S'} \sigma' \parallel C'$ *and* s'_r *is unambiguous at size* $k+1$ *for* $|C'|$ *with* $\mathrm{AM}(s_r(\mathcal{K}\langle e \rangle)) = \mathrm{AM}(s'_r(\mathcal{K}\langle e' \rangle))$.

Proof. By Proposition 10, there exists by S', σ', C', s'_r such that $\Gamma \vdash \mathcal{K}\langle e' \rangle :_{S'} \sigma' \parallel C'$. We proceed by induction on $\mathcal{K}$.

Case $\mathcal{K} = \langle \cdot \rangle$**:**

We proceed by cases on the removal relation.

Subcase Rem-Map ($\mathbf{map}\ e_1\ e_2 \prec_{\mathrm{rem}} e_1\ e_2 \triangle (M, R)$)**:**

By the definition of AM, we have $\lceil 1 \rceil$ $\mathrm{AM}(s_r(\mathcal{K}\langle e \rangle)) = \mathrm{AM}(s_r(\mathbf{map}\ e_1\ e_2)) = \mathbf{map}\ \mathrm{AM}(s_r(e_1))\ \mathrm{AM}(s_r(e_2))$ and $\lceil 2 \rceil$ $\mathrm{AM}(s'_r(\mathcal{K}\langle e' \rangle)) = \mathrm{AM}(s'_r(e_1\ e_2 \triangle (M, R))) = \mathbf{map}^{s'_r(M)}\ \mathrm{AM}(s'_r(e_1))\ (\mathbf{rep}^{s'_r(R)}\ \mathrm{AM}(s'_r(e_2)))$. By Proposition 10, there exists $\hat{s}'_r$ such that $s_r \circ \hat{s}'_r$ satisfies $|C'|$ and by the construction in the proof, one such possibility is $\hat{s}'_r = [M \mapsto 1, R \mapsto 0]$. But then clearly $\mathrm{size}(s_r \circ \hat{s}_r, |C'|) = k+1$ (because $\mathrm{size}(s_r, |C|) = 1$ and the constraint sets only differ on the constraints introduced by the **map** removal), which requires that $\lceil 3 \rceil$ $s'_r = s_r \circ \hat{s}_r$ because s'_r is unambiguous at size $k+1$ for $|C'|$. Hence, by $\lceil 1 \rceil$ and $\lceil 2 \rceil$, $\mathrm{AM}(s_r(\mathcal{K}\langle e \rangle)) = \mathbf{map}\ \mathrm{AM}(s_r(e_1))\ \mathrm{AM}(s_r(e_2)) = \mathbf{map}^{s'_r(M)}\ \mathrm{AM}(s'_r(e_1))\ (\mathbf{rep}^{s'_r(R)}\ \mathrm{AM}(s'_r(e_2))) = \mathrm{AM}(s'_r(\mathcal{K}\langle e' \rangle))$.

Subcase Rem-Rep ($\mathbf{rep}\ e \prec_{\mathrm{rem}} (\lambda x.\, x)\ e \triangle (M, R)$)**:**

By the definition of AM, we have $\lceil 1 \rceil$ $\mathrm{AM}(s_r(\mathcal{K}\langle e \rangle)) = \mathrm{AM}(s_r(\mathbf{rep}\ e)) = \mathbf{rep}\ \mathrm{AM}(s_r(e))$ and $\lceil 2 \rceil$ $\mathrm{AM}(s'_r(\mathcal{K}\langle e' \rangle)) = \mathrm{AM}(s'_r((\lambda x.\, x)\ e \triangle (M, R))) = \mathbf{map}^{s'_r(M)}\ \lambda x.\, x\ (\mathbf{rep}^{s'_r(R)}\ \mathrm{AM}(s'_r(e)))$. As with the Rem-Map case, we use the $\hat{s}'_r$ construction of Proposition 10 and the unambiguity of s'_r to argue that $s'_r = s_r \circ \hat{s}_r$. Hence, by $\lceil 1 \rceil$ and $\lceil 2 \rceil$, $\mathrm{AM}(s_r(\mathcal{K}\langle e \rangle)) = \mathbf{rep}\ \mathrm{AM}(s_r(e)) = \mathbf{rep}^{s'_r(R)}\ \mathrm{AM}(()s'_r(e)) = \mathbf{rep}^{s'_r(R)}\ \mathrm{AM}(()s'_r(e)) = \mathbf{map}^{s'_r(M)}\ \lambda x.\, x\ (\mathbf{rep}^{s'_r(R)}\ \mathrm{AM}(s'_r(e))) = \mathrm{AM}(s'_r(\mathcal{K}\langle e' \rangle))$.

Case $\mathcal{K} = \mathbf{rep}\ \mathcal{K}'$**:**

By the definition of AM, we have $\lceil 1 \rceil$ $\mathrm{AM}(s_r(\mathbf{rep}\ \mathcal{K}'\langle e \rangle)) = \mathbf{rep}\ \mathrm{AM}(s_r(\mathcal{K}'\langle e \rangle))$. Since $\mathbf{rep}\ \mathcal{K}'\langle e \rangle \prec_{\mathrm{rem}} \mathbf{rep}\ \mathcal{K}'\langle e' \rangle$, $\lceil 2 \rceil$ $\mathcal{K}'\langle e \rangle \prec_{\mathrm{rem}} \mathcal{K}'\langle e' \rangle$ by the definition of the removal relation. Additionally, by inversion on C-Rep, there exists τ and S_1 with $\sigma = [\,]S_1\tau$ such that $\lceil 3 \rceil$ $\Gamma \vdash \mathcal{K}'\langle e \rangle :_{S_1} \tau \parallel C$ and, similarly, there exists τ' and S'_1 with $\sigma' = [\,]S'_1\tau'$ such that $\lceil 4 \rceil$ $\Gamma \vdash \mathcal{K}'\langle e' \rangle :_{S'_1} \tau' \parallel C'$. Hence, applying the IH to $\lceil 2 \rceil$, $\lceil 3 \rceil$, and $\lceil 4 \rceil$, we have that $\lceil 5 \rceil$ $\mathrm{AM}(s_r(\mathcal{K}'\langle e \rangle)) = \mathrm{AM}(s'_r(\mathcal{K}'\langle e' \rangle))$. Combining $\lceil 1 \rceil$ and $\lceil 5 \rceil$ yields $\mathrm{AM}(s_r(\mathcal{K}\langle e \rangle)) = \mathbf{rep}\ \mathrm{AM}(s_r(\mathcal{K}'\langle e \rangle)) = \mathbf{rep}\ \mathrm{AM}(s'_r(\mathcal{K}'\langle e' \rangle)) = \mathrm{AM}(s'_r(\mathcal{K}\langle e' \rangle))$, as required.

The remaining cases are all similar to the $\mathcal{K} = \mathbf{rep}\ \mathcal{K}'$ case and involve applying the IH appropriately. □

A.7.3 Forwards Consistency

Proposition 12 (Forwards Consistency)**.** *If* $\mathcal{K}\langle e\rangle \succ_{add} \mathcal{K}\langle e'\rangle$ *then* $\mathrm{AM}(\mathcal{K}\langle e\rangle) = \mathrm{AM}(\mathcal{K}\langle e'\rangle)$.

Proof. We proceed by induction on $\mathcal{K}$.

Case $\mathcal{K} = \langle\cdot\rangle$**:**

We proceed by cases on the add relation.

Subcase Add-Map $(e_1\ e_2 \triangle (n_\mathrm{M}+1, n_\mathrm{R}) \succ_\mathrm{add} (\mathbf{map}\ e_1)\ e_2 \triangle (n_\mathrm{M}, n_\mathrm{R}))$**:**

By the definition of AM, we have $\lceil 1\rceil\ \mathrm{AM}(\mathcal{K}\langle e\rangle) = \mathrm{AM}(e_1\ e_2 \triangle (n_M+1, n_R)) = \mathbf{map}^{n_\mathrm{M}+1}\ \mathrm{AM}(e_1)\ (\mathbf{rep}^{n_\mathrm{R}}\ \mathrm{AM}(e_2))$ and $\lceil 2\rceil\ \mathrm{AM}(\mathcal{K}\langle e'\rangle) = \mathbf{map}^{n_\mathrm{M}}\ \mathrm{AM}(\mathbf{map}\ e_1)\ (\mathbf{rep}^{n_\mathrm{R}}\ \mathrm{AM}(e_2)) = \mathbf{map}^{n_\mathrm{M}+1}\ \mathrm{AM}(e_1)\ (\mathbf{rep}^{n_\mathrm{R}}\ \mathrm{AM}(e_2))$

Subcase Add-Rep $(e_1\ e_2 \triangle (n_\mathrm{M}, n_\mathrm{R}+1) \succ_\mathrm{add} e_1\ (\mathbf{rep}\ e_2) \triangle (n_\mathrm{M}, n_\mathrm{R}))$**:**

By the definition of AM, we have $\lceil 1\rceil\ \mathrm{AM}(\mathcal{K}\langle e\rangle) = \mathrm{AM}(e_1\ e_2 \triangle (n_M, n_R+1)) = \mathbf{map}^{n_\mathrm{M}}\ \mathrm{AM}(e_1)\ (\mathbf{rep}^{n_\mathrm{R}+1}\ \mathrm{AM}(e_2))$ and $\lceil 2\rceil\ \mathrm{AM}(\mathcal{K}\langle e'\rangle) = \mathbf{map}^{n_\mathrm{M}}\ \mathrm{AM}(e_1)\ (\mathbf{rep}^{n_\mathrm{R}}\ \mathrm{AM}(\mathbf{rep}\ e_2)) = \mathbf{map}^{n_\mathrm{M}}\ \mathrm{AM}(e_1)\ (\mathbf{rep}^{n_\mathrm{R}+1}\ \mathrm{AM}(e_2))$

Case $\mathcal{K} = \mathbf{rep}\ \mathcal{K}'$**:**

By the definition of AM, we have $\lceil 1\rceil\ \mathrm{AM}(\mathbf{rep}\ \mathcal{K}'\langle e\rangle) = \mathbf{rep}\ \mathrm{AM}(\mathcal{K}'\langle e\rangle)$.

Since $\mathbf{rep}\ \mathcal{K}'\langle e\rangle \succ_\mathrm{add} \mathbf{rep}\ \mathcal{K}'\langle e'\rangle$, $\lceil 2\rceil\ \mathcal{K}'\langle e\rangle \succ_\mathrm{add} \mathcal{K}'\langle e'\rangle$ by the definition of the add relation. By the IH, $\lceil 3\rceil\ \mathrm{AM}(\mathcal{K}'\langle e\rangle) = \mathrm{AM}(\mathcal{K}'\langle e'\rangle)$ and hence by the definition of AM, we have $\mathrm{AM}(\mathcal{K}\langle e\rangle) = \mathrm{AM}(\mathcal{K}\langle e'\rangle)$ as required.

The remaining inductive cases are analogous. □