AUTOMAP: Inferring Rank-Polymorphic Function Applications with Integer Linear Programming

ROBERT SCHENCK, University of Copenhagen, Denmark
NIKOLAJ HEY HINNERSKOV, University of Copenhagen, Denmark
TROELS HENRIKSEN, University of Copenhagen, Denmark
MAGNUS MADSEN, Aarhus University, Denmark
MARTIN ELSMAN, University of Copenhagen, Denmark

Dynamically typed array languages such as Python, APL, and Matlab lift scalar operations to arrays and replicate scalars to fit applications. We present a mechanism for automatically inferring map and replicate operations in a statically-typed language in a way that resembles the programming experience of a dynamically-typed language while preserving the static typing guarantees. Our type system—which supports parametric polymorphism, higher-order functions, and top-level let-generalization—makes use of integer linear programming in order to find the minimum number of operations needed to elaborate to a well-typed program. We argue that the inference system provides useful and unsurprising guarantees to the programmer. We demonstrate important theoretical properties of the mechanism and report on the implementation of the mechanism in the statically-typed array programming language Futhark.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Parallel programming languages**.

Additional Key Words and Phrases: data parallelism, constraint-based type systems, array programming

ACM Reference Format:
Robert Schenck, Nikolaj Hey Hinnerskov, Troels Henriksen, Magnus Madsen, and Martin Elsman. 2024. AUTOMAP: Inferring Rank-Polymorphic Function Applications with Integer Linear Programming. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 334 (October 2024), 27 pages. https://doi.org/10.1145/3689774

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.

Authors' Contact Information: Robert Schenck, University of Copenhagen, Copenhagen, Denmark, r@bert.lol; Nikolaj Hey Hinnerskov, University of Copenhagen, Copenhagen, Denmark, nihi@di.ku.dk; Troels Henriksen, University of Copenhagen, Copenhagen, Denmark, athas@sigkill.dk; Magnus Madsen, Aarhus University, Aarhus, Denmark, magnusm@cs.au.dk; Martin Elsman, University of Copenhagen, Copenhagen, Denmark, mael@di.ku.dk.

ACM 2475-1421/2024/10-ART334
https://doi.org/10.1145/3689774

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 [Hui and Kromberg 2020]. Most rank polymorphic languages are dynamically typed; while work exists on expressing rank polymorphism in statically typed languages [Gibbons 2017; Slepak et al. 2014; Šinkarovs et al. 2023], 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 (i.e., broadcasting), 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 [Jeffery 2020]. Most notably, implicits have been used to express type classes in the Scala programming language [Odersky et al. 2017; Oliveira et al. 2010]. 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 paper, 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 an 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

[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.

standard dynamic semantics for the target language. We define the meaning of a 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 in which `map` and `rep` constructs are inserted 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 [Henriksen et al. 2017], a statically-typed functional data-parallel array language. 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 [Vytiniotis et al. 2011, 2010]).

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., `maps` 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 $2.5\times$.

In summary, the contributions of the paper 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 `maps` and `reps` 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 `maps` and `reps` 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 (i) Automap is effective (i.e., we can omit `map` and `rep` operations to make programs more concise and readable), (ii) type inference with linear constraint solving is practically feasible, and (iii) Automap is able to unambiguously capture those `map` operations that programmers want to leave implicit.

The paper is organized as follows: Section 2 motivates the need for implicit maps and replicates in Futhark. In Section 3 through Section 7, we present a formalization of the mechanism, in terms of an Automap elaboration into a well-defined target language. Section 8 discusses the implementation of the constraint-based type system, in particular its interaction with other desirable type system features. Section 9 evaluates the usefulness of Automap on a collection of real-world Futhark programs. Finally, Section 10 discusses future work, Section 11 presents related work, and Section 12 concludes.

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 [Hui and Kromberg 2020; Iverson 1962].

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`):

```
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` → `int` → `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.[2] 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 `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,[3] the above expression can be rewritten as

```
[3,4,5] + rep 1
```

and then further as

```
map2 (+) [3,4,5] (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 on readability can be quite dramatic, as we will see in the following section and in Section 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 * c` instead of `map (map (map (*c))) A`.

2.2 Examples

Consider the following definition of linear interpolation:

```
def lerp v w t = v + (w - v) * 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

[2]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`.
[3]The issue of how many elements should be replicated will be discussed briefly in Section 8.

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. 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 by manually inserting maps and reps in the source program. We enshrine this strategy as the first rule of AUTOMAP:

Rule 1: minimize the number of inserted maps and reps.

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 there can be a rank mismatch: either the argument is under-dimensioned (in which case one or more reps are required) or over-dimensioned (in which case one or more maps are required).

Rule 2: a single application may have implicit maps or reps 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 for them to understand—an important quality in any implicit programming system.

Returning to the linear interpolation example, following this strategy, we infer `lerp : float -> float -> float -> float` (i.e., 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**
`lerp vs ws t`	$\longrightarrow$	`map3 lerp vs ws (rep t)`
`lerp vs ws ts`	$\longrightarrow$	`map3 lerp vs ws ts`
`lerp v w ts`	$\longrightarrow$	`map3 lerp (rep v) (rep w) ts`

where `v`, `w`, `t` have type `float` and `vs`, `ws`, `ts` have type `[]float` (i.e., 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 maps, an implementation could be:

```
def outerprod f x y = map (\x' -> map (f x') y) x
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 maps (and changing to infix notation), we obtain:

```
def outerprod f x y = (\x' -> f x' y) x
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 x y = map (\x' -> f x' y) x
```

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

```
(1) map2 (map2 (==)) (xmat (rep A)) (rep (rep (A!=0))) |> flatten |> and
(2) map2 (map2 (==)) (xmat A) (rep (map2 (!=) 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:[4]

```
def check (A: [][]i32) = xmat A == (A!=0) |> flatten |> and
```

The additional type constraint ensures that only a single AUTOMAP elaboration is well-typed. The inferred reps can be simplified away during elaboration, as we will discuss in Section 8.

2.3 Desired Properties

We consider five requirements that the design of AUTOMAP should satisfy. Additionally, these requirements 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.
- (DETERMINISM) For any program, the elaboration is unambiguous or the program is rejected.[5]
- (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. Bi-directional 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 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 [Milner 1997]. 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]Note that AUTOMAP never requires type annotations and check could also be disambiguated by inserting a map operation.
[5]A naïve elaboration could reject *all* programs. We show that this is not the case for AUTOMAP.

3 Formalization

3.1 Overview

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 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 1, the three languages are identical and will be fully specified in the next section, but we'll first give an overview of each.

Table 1. The differences between the source, internal, and target languages. The "Example" row shows how the source expression `inc` $[1,2,3]$ is transformed into the internal language and then from the internal language into the target language; `inc` is an illustrative scalar function that increments its argument by 1.

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

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 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 1, the source expression `inc` $[1,2,3]$ is transformed to the internal expression `inc` $[1,2,3] \triangle (M,R)$ 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 1, the target expression `map inc` $[1,2,3]$ is obtained from the internal expression `inc` $[1,2,3] \triangle (M,R)$ by using the using the inference algorithm to find the solution $M \mapsto 1$, $R \mapsto 0$ and then transforming into the corresponding target expression `map inc` $[1,2,3]$—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.

3.2 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.

$$
\begin{array}{rcll}
S^{\oslash} & ::= & [\,] & \text{dimension} \\
 & \mid & S^{\oslash}\, S^{\oslash} & \text{concatenation} \\
 & \mid & \bullet & \text{empty shape} \\
 \\
\tau^{\oslash} & ::= & \mathtt{int} & \text{integers} \\
 & \mid & \tau^{\oslash} \rightarrow \tau^{\oslash} & \text{functions} \\
 & \mid & S^{\oslash}\, \tau^{\oslash} & \text{arrays} \\
 & \mid & \alpha \quad (\alpha \in V_\mathrm{t}) & \text{type variable}
\end{array}
\qquad
\begin{array}{rcll}
S & ::= & S^{\oslash} & \text{shapes without rank vars} \\
 & \mid & [\,]^{Q} \quad (Q \in V_\mathrm{r}) & \text{rank power} \\
 & \mid & S\, S & \text{concatenation} \\
 \\
\tau & ::= & \tau^{\oslash} & \text{types without rank vars} \\
 & \mid & \tau \rightarrow \tau & \text{functions} \\
 & \mid & S\, \tau & \text{arrays} \\
 \\
\sigma & ::= & \tau & \text{type} \\
 & \mid & \forall\, \alpha\, .\, \sigma & \text{quantified type}
\end{array}
$$

Fig. 1. Grammar for closed shapes ($S^{\oslash}$), shapes (S), closed types ($\tau^{\oslash}$), types (τ), and type schemes (σ).

Figure 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 $[\,]$s. Correspondingly, when n is a nonnegative integer, $[\,]^{n}$ is sugar for a sequence of n $[\,]$s. 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 $X_1, \ldots, X_n$ and we may sometimes just write $\vec{X}$ if the length of the sequence is clear from the context.

Figure 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 (show in blue in the figure). The application $e\ e \vartriangle (M, R)$ 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 $e\ e \vartriangle (n, n)$.

For values of the form $\lambda x.\, e$ and programs of the form $\mathtt{def}\ f\ x = e\ ;\ p$, 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.

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

Fig. 2. Grammar for values (v), expressions (e), and programs (p). In the internal language, applications are annotated with rank specifications (shown in blue).

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:

$$
\begin{aligned}
\texttt{map}^0\ e &= e, & \texttt{rep}^0\ e &= e, \\
\texttt{map}^{n+1}\ e &= \texttt{map}\ (\texttt{map}^n\ e), & \texttt{rep}^{n+1}\ e &= \texttt{rep}\ (\texttt{rep}^n\ e).
\end{aligned}
$$

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 of 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 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 $\vartriangle (M, R)$ 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 [Bailly et al. 2023; Henriksen and Elsman 2021]) 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 $\tau^{\varnothing}$, as all types are closed with respect to rank variables and environments shall implicitly contain only *closed type schemes* (i.e., 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 $\mathrm{dom}(s_{\mathrm{t}}) = \{\vec{\alpha}\}$ and $s_{\mathrm{t}}(\tau') = \tau$.

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

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

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

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

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ \textsc{S-Var} \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma \geq \tau}{\Gamma \vdash e : \tau}\ \textsc{S-Inst} \qquad \frac{\forall i \in \{1,\ldots,n\}.\, \Gamma \vdash e_i : \tau}{\Gamma \vdash [e_1,\ldots,e_n] : [\,]\tau}\ \textsc{S-Array}$$

$$\frac{\vdash v : \tau}{\Gamma \vdash v : \tau}\ \textsc{S-Val} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \to \tau'}\ \textsc{S-Fun} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ \textsc{S-App}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : [\,]\tau_1}{\Gamma \vdash \texttt{map}\, e_1\, e_2 : [\,]\tau_2}\ \textsc{S-Map} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{rep}\, e : [\,]\tau}\ \textsc{S-Rep}$$

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

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

Fig. 3. Target language typing rules.

Figure 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 4.1 (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 the supplementary material [Schenck 2024] 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 $\{\vec{\alpha'}\} \cap \mathrm{ftv}(s_t(\Gamma), s_t(\sigma')) = \emptyset$. 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 [Morrisett 1995; Wright and Felleisen 1994]. We define the notions of *contexts* (K) and *redexes* (r) according to the grammars in Figure 4 below.

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

Fig. 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:

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

Fig. 5. Reduction rules for the target language.

The proofs for the following propositions are standard, so we relegate full proofs to the supplementary material [Schenck 2024] 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 4.2 (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 4.3 (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 4.4 (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 4.2. □

Proposition 4.5 (Preservation). *If $\vdash p : \sigma$ and $p \rightsquigarrow p'$ then $\vdash p' : \sigma$.*

Proof Sketch. By induction over the typing derivation, using Propositions 4.2 and 4.3. □

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.

5.1 Constraints

A *constraint* is a relation defined by the grammar in Figure 6. Constraints are generated by the type rules in Figure 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 ($M \veebar R$).

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

Fig. 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 $M \veebar R$ 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 2.2—namely that each application can have either implicit `maps` or implicit `reps`, 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'.

5.2 Internal Type System

Figure 7 shows the typing rules for constraint-based judgments; the judgment $\Gamma \vdash e \ :_S \ \sigma \ \| \ C$ 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; $\Gamma \vdash e \ : \ \sigma \ \| \ C$ is syntactic sugar for $\Gamma \vdash e \ :_\bullet \ \sigma \ \| \ C$. Frames are the extra leading dimensions on a result type generated by preceding `maps`; in the relation $\Gamma \vdash e \ :_S \ \sigma \ \| \ C$, 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 of the proofs of the propositions that follow, but is also critical for the implementation (see Section 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 `maps` and `reps` of the application, respectively. The $M \veebar R$ constraint encodes that one of M or R must be zero since a flexible function application allows *either* implicit `reps` or an implicit `maps`, but not both (i.e., it enforces Rule 2 of Section 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 `maps`), and an under-dimensioned argument has $[\,]^R$ extra dimensions

$$\boxed{\Gamma \vdash e \;:_{S}\; \sigma \;\|\; C}$$

$$\frac{}{\vdash n \;:\; \mathtt{int} \;\|\; \emptyset}\;\textsc{C-Int} \qquad \frac{\Gamma \vdash e \;:\; \sigma \;\|\; \emptyset \qquad \sigma \geq \tau}{\Gamma \vdash e \;:\; \tau \;\|\; \emptyset}\;\textsc{C-Inst} \qquad \frac{}{\Gamma, x : \sigma \vdash x \;:\; \sigma \;\|\; \emptyset}\;\textsc{C-Var}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \;:_{S_1}\; \tau_1 \to \tau_2 \;\|\; C_1 \qquad \Gamma \vdash e_2 \;:_{S_2}\; \tau_3 \;\|\; C_2 \\ M, R \text{ fresh} \qquad C = \{M \dot{\vee} R,\; [\,]^M\, S_1\, \tau_1 \doteq [\,]^R\, S_2\, \tau_3\}\end{array}}{\Gamma \vdash e_1\; e_2 \vartriangle (M, R) \;:_{[\,]^M S_1}\; \tau_2 \;\|\; C \cup C_1 \cup C_2}\;\textsc{C-App} \qquad \frac{\Gamma, x : \tau_1 \vdash e \;:_{S}\; \tau_2 \;\|\; C}{\Gamma \vdash \lambda x.\, e \;:\; \tau_1 \to S\, \tau_2 \;\|\; C}\;\textsc{C-Fun}$$

$$\frac{\Gamma \vdash e_1 \;:_{S_1}\; \tau_1 \to \tau_2 \;\|\; C_1 \qquad \Gamma \vdash e_2 \;:_{S_2}\; \tau_3 \;\|\; C_2}{\Gamma \vdash \mathtt{map}\; e_1\; e_2 \;:_{[\,]S_1}\; \tau_2 \;\|\; \{[\,]S_1\tau_1 \doteq S_2\tau_3\} \cup C_1 \cup C_2}\;\textsc{C-Map} \qquad \frac{\Gamma \vdash e \;:_{S}\; \tau \;\|\; C}{\Gamma \vdash \mathtt{rep}\; e \;:\; [\,]S\tau \;\|\; C}\;\textsc{C-Rep}$$

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

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

Fig. 7. Constraint-based typing rules.

tacked on (corresponding to R implicit `reps`). 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 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 each top-level 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 7, consider the the source language expression `inc` $[1, 2, 3]$ introduced in Table 1. Via C-Var, C-Array, and C-App we have

$$\mathtt{inc} : \mathtt{int} \to \mathtt{int} \vdash \mathtt{inc}\; [1, 2, 3] \vartriangle (M, R) \;:_{[\,]^M}\; \mathtt{int} \;\|\; \{M \dot{\vee} R,\; [\,]^M\, \mathtt{int} \doteq [\,]^R\, [\,]\, \mathtt{int}\}$$

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 $s = s_{\mathrm{t}} \circ s_{\mathrm{r}}$.

6.1 Rank

The *rank* of a shape or of a type, written $|\cdot|$, is defined in Figure 8 and denotes the dimensionality of the shape or type. Because a shape S or a type τ may contain (rank and type) variables, $|S|$ and

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

Fig. 8. Definition of rank; $\overline{\alpha}$ is the associated rank variable of α.

$|\tau|$ 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 $[\,]^n\mathtt{int}$ is given by $|[\,]^n\mathtt{int}| = |[\,]^n| + |\mathtt{int}| = n + 0 = n$. Another example—this time including a rank and a type variable—is $|[\,]^M\alpha| = M + \overline{\alpha}$.

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 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 9 shows the constraint grammar extended with rank equality constraints.

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

Fig. 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:

$$|\tau_1 \doteq \tau_2| = |\tau_1| \doteq |\tau_2|,$$
$$|M \mathrel{\dot\vee} R| = M \mathrel{\dot\vee} R.$$

Intuitively, $|\cdot|$ is the identity operation on zero-rank disjunction constraints ($M \mathrel{\dot\vee} R$) 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 $C = \{M \mathrel{\dot\vee} R,\ [\,]^M\ \mathtt{int} \doteq [\,]^R\,[\,]\mathtt{int}\}$ from the typing of $\mathtt{inc}\ [1,2,3] \triangle (M,R)$ just before Section 6 on the previous page. The corresponding rank constraint set is given by $|C| = \{M \mathrel{\dot\vee} R, |[\,]^M\ \mathtt{int} \doteq [\,]^R\,[\,]\mathtt{int}|\} = \{M \mathrel{\dot\vee} R, M \doteq R + 1\}$, 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 1.

6.3 Size and Ambiguity

There may be many different rank substitutions that can satisfy a 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 maps and reps 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 $\{M \veebar R, [\,]^M\alpha \doteq [\,]^R[\,][\,][\,]\mathtt{int}\}$. The corresponding rank constraint set is given by $\{M \veebar R, M + \overline{\alpha} \doteq R + 2\}$; one possible satisfier with size 1 is $s_\mathrm{r} = [M \mapsto 1, R \mapsto 0, \overline{\alpha} \mapsto 1]$, but another is $s_\mathrm{r} = [M \mapsto 0, R \mapsto 1, \overline{\alpha} \mapsto 3]$ and hence $|C|$ is ambiguous at size 1. On the other hand, $|C|$ is unambiguous at size 0 and $s_\mathrm{r} = [M \mapsto 0, R \mapsto 0, \overline{\alpha} \mapsto 2]$ is a satisfier. (All size 0 solutions are always unique since they correspond to no implicit maps and reps).

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 2 shows how rank constraints are converted into constraints of the ILP.

Table 2. Mapping of rank constraints to ILP constraints. U is a constant upper bound on M and R.

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

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 ($M \veebar R$) are satisfied by a rank substiution s_r when $s_\mathrm{r}(M) = 0$ or $s_\mathrm{r}(R) = 0$. 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 maps and reps 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 10 shows how the rank ILP is generated for the expression `sum` (`length` xss), where xss is a matrix.

$$\dfrac{\Gamma \vdash \mathtt{sum}\ :\ [\,]\mathtt{int} \to \mathtt{int}\ \|\ \emptyset \qquad C_2 = \{M_2 \mathrel{\dot\vee} R_2,\ [\,]^{M_2}[\,]\mathtt{int} \doteq [\,]^{R_2}[\,]^{M_1}\mathtt{int}\} \qquad \dfrac{\begin{array}{c}\Gamma \vdash \mathtt{length}\ :\ [\,]\alpha \to \mathtt{int}\ \|\ \emptyset \\ \Gamma \vdash xss\ :\ [\,][\,]\mathtt{int}\ \|\ \emptyset \qquad C_1 = \{M_1 \mathrel{\dot\vee} R_1,\ [\,]^{M_1}[\,]\alpha \doteq [\,]^{R_1}[\,][\,]\mathtt{int}\}\end{array}}{\Gamma \vdash \mathtt{length}\ xss \triangle (M_1, R_1)\ :_{[\,]^{M_1}}\ \mathtt{int}\ \|\ C_1}\ \text{C-App}}{\Gamma \vdash \mathtt{sum}\ (\mathtt{length}\ xss) \triangle (M_2, R_2)\ :_{[\,]^{M_2}[\,]^{M_1}}\ \mathtt{int}\ \|\ C_2 \cup C_1}\ \text{C-App}$$

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

$$|C_2 \cup C_1| = \left\{ \begin{array}{l} M_1 \mathrel{\dot\vee} R_1, \\ M_1 + 1 + \overline{\alpha} \doteq R_1 + 2, \\ M_2 \mathrel{\dot\vee} R_2, \\ M_2 + 1 \doteq R_2 + M_1 \end{array} \right\}$$

(b) The associated rank constraint set of $C_2 \cup C_1$ from Figure 10a.

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

(c) The ILP corresponding to $|C_2 \cup C_1|$ in Figure 10b. All variables are additionally constrained to be non-negative; these constraints are omitted for brevity.

Fig. 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 expression 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 $s_r(C)$ must assign to α; in the first elaboration, $s_t(\alpha) = \mathtt{int}$, in the second, $s_t(\alpha) = [\,]\mathtt{int}$.

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 the 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.

6.5 Constraint Set Solving

Our constraint solving algorithm Solve is shown in Figure 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 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 [Milner 1978], 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 $s_r(C)$ is a closure of C; that is, all of its rank variables have been instantiated with integral ranks. Hence, the constraint set $s_r(C)$ on line 10 of the Solve procedure is closed.

If C is satisfiable, so must be $s_r(C)$ and a satisfying type substitution s_t can be found via standard syntactic type unification. This means that the substitution $s_t \circ s_r$—where s_t is returned by Unify—is a satisfier of the original constraint set C. Propositions 6.1 and 6.2 show that every satisfier of C can be found in this manner; their full proofs can be found in the supplementary material [Schenck 2024].

```
Procedure Solve
input  : A constraint set $C$.
output : A satisfying substitution $s$.
1  $|C| \leftarrow$ construct the associated rank constraint set from $C$;            // Section 6.2
2  $I \leftarrow$ construct the corresponding ILP from $|C|$;                          // Section 6.4
3  $s_\mathrm{r} \leftarrow$ solve $I$ using an ILP solver;
4  if $s_\mathrm{r}$ then
5      $I' \leftarrow$ add constraints to $I$ to ban solution $s_\mathrm{r}$ and enforce a size of size$(s_\mathrm{r}, |C|)$;
6      $s'_\mathrm{r} \leftarrow$ solve $I'$ using an ILP solver;
7      if $s'_\mathrm{r}$ then
8          return $\perp$;                                              // |C| is ambiguous; fail
9      else
10         return Unify$(s_\mathrm{r}(C)) \circ s_\mathrm{r}$
11 else
12     return $\perp$
```

Fig. 11. Our algorithm for constraint solving.

Proposition 6.1. *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 Sketch. Follows by constructing a rank substitution s_r defined by $s_\mathrm{r}(Q) = s(Q)$ and $s_\mathrm{r}(\overline{\alpha}) = |s(\alpha)|^{\varnothing}$ (where $|\cdot|^{\varnothing}$ works like $|\cdot|$ except it assigns type variables the rank 0 and expects a closed argument) and a type substitution $s_\mathrm{t} = s|_{\mathrm{ftv}(C)}$. □

Proposition 6.2. *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 Sketch. Since C is satisfiable, a satisfier s for it exists; s_t is constructed using both s_r and s and is defined as $s_\mathrm{t}(\alpha) = [\,]^{s_\mathrm{r}(\overline{\alpha})}\mathrm{basetype}(s(\alpha))$ where $\mathrm{basetype}(S\tau) = \mathrm{basetype}(\tau)$ and is the identity otherwise (that is, basetype strips all array dimensions). □

Note that Propositions 6.1 and 6.2 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 $|C|$, $s_\mathrm{r}(C)$ may not be satisfiable due to size mismatches.

7 Transformation to the Target Language

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 ($\triangle\ (n_\mathrm{M}, n_\mathrm{R})$)—corresponding to implicit numbers of `maps` and `reps`—into explicit `maps` and `reps`. That is, AM converts programs from the internal language into the target language. It is defined in Figure 12.

Recall that the $R \veebar M$ constraint in C-App forces one of M, R to be 0; hence at least n_M or n_R in the above must be 0 and the expansion of $e_1\ e_2 \triangle (n_\mathrm{M}, n_\mathrm{R})$ will result in either a `map` or a `rep` (or neither) but never both (in accordance with Rule 2 of Section 2.2).

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

Fig. 12. The AM transformation from the internal language to the target language; note that $n_\mathrm{M}, n_\mathrm{R} \in \mathbb{N}$.

Returning to the example from Table 1 ($\mathtt{inc}\ [1,2,3] \vartriangle (M,R)$), in Section 6.2 we found that the rank substitution $s_\mathrm{r} = [M \mapsto 1, R \mapsto 0]$ satisfied its rank constraint set; this solution is also minimal and would be the same one returned by the Solve algorithm.[6] The substitution is applied to the expression to obtain $\mathtt{inc}\ [1,2,3] \vartriangle (1,0)$, which is the form the expression would take in the conclusion of the C-Def rule. To now convert the internal expression $\mathtt{inc}\ [1,2,3] \vartriangle (1,0)$ to the target language, we simply apply the AM transformation:

$$\begin{aligned}
\mathrm{AM}(\mathtt{inc}\ [1,2,3] \vartriangle (1,0)) &= \mathtt{map}^1\ \mathrm{AM}(\mathtt{inc})\ (\mathtt{rep}^0\ \mathrm{AM}([1,2,3])) \\
&= \mathtt{map}\ \mathtt{inc}\ [1,2,3].
\end{aligned}$$

7.1 Well-Typedness

Proposition 7.2 shows the Well-Typedness property from Section 2.3. That is, if p is a well-typed program in the internal language then $\mathrm{AM}(p)$ is a well-typed program in the target language. First, an equivalent property is needed for expressions.

Proposition 7.1 (Well-Typedness for Expressions). *If* $\Gamma \vdash e\ :_S\ \sigma \ \|\ C$ *and* s *is a satisfier of* C*, then* $s(\Gamma) \vdash \mathrm{AM}(s(e))\ :\ s(S\ \sigma)$

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 7.2 (Well-Typedness). *If* $\Gamma \vdash p\ :\ \sigma$ *then* $\Gamma \vdash \mathrm{AM}(p)\ :\ \sigma$.

Proof Sketch. Straightforward induction over the typing derivation, using Proposition 7.1. □

The supplementary material [Schenck 2024] features full proofs for the above propositions.

7.2 Backwards Consistency

In this section, we show that rank analysis correctly reconstructs a removed explicit map or rep from an expression.[7] To talk about specific maps or rep in a program, we introduce internal language contexts with the following grammar:

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

Fig. 13. Internal language contexts grammar.

[6]Since the constraint set doesn't feature any type variables, the Unify call in Figure 11 will return an empty substitution and hence the satisfier for the constraint set is simply the substitution $s = s_\mathrm{r}$.
[7]The Forwards Consistency property is shown similarly.

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 a semantically equivalent expression to $\mathcal{K}\langle e\rangle$ with the `map` or `rep` removed and replaced by a flexible function application, shown in Figure 14 below.

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

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

Fig. 14. The removal relation.

The following proposition says that we can remove a `map` or a `rep` from a well-typed expression, replace it with a flexible function application, and obtain a well-typed expression. Additionally, it says that the constraint sets returned in typing the two expressions 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 7.3 (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) $C \simeq \hat{s}'_r(C')$.

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 $\hat{s}'_r$ 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 $\hat{s}'_r$ (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 7.4 formally states the Backwards Consistency property of Section 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 7.4 (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, if s'_r is unambiguous at size $k+1$ for $|C'|$, then* $\mathrm{AM}(s_r(\mathcal{K}\langle e\rangle)) = \mathrm{AM}(s'_r(\mathcal{K}\langle e'\rangle))$.

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 $s'_r = s_r \circ \hat{s}'_r$ where $\hat{s}_r$ is as constructed in the relevant cases of the proof of Proposition 7.3, from which we obtain the required equality by the definition of AM. □

8 Implementation

We have implemented Automap in a compiler for the functional array language Futhark. The implementation consists of four phases, shown in Figure 15 below.

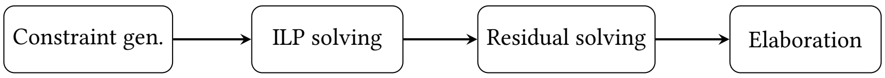

Fig. 15. The four phases of AUTOMAP in the implementation.

8.1 Constraint Generation

This part of the implementation largely follows the structure given in Section 5: each top-level definition is type checked individually, application nodes are annotated with the equivalent of the $\triangle\ (M, R,)$ annotations in Section 5,[8] and constraints are collected.

8.2 ILP Solving

The implementation mostly follows Section 6.5, utilizing the GNU Linear Programming Kit[9] to solve the ILPs. One notable difference from Section 6.5 is that the ILP is modified to avoid counting *induced* `reps`, which are `reps` that are required as a consequence of previous `maps`. For example, `map` $(\lambda y.\, xs * y)\ ys$ (noting the existent `map` is an explicit one) can be elaborated as

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

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` $(\lambda y.\, xs * y)\ ys$ 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

$$\texttt{map}\ (\lambda y.\, \texttt{map}\ (\lambda x.\, x\ *\ y)\ xs)\ ys$$

which has size 1 and is the only elaboration of `map` $(\lambda y.\, xs * y)\ ys$ 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 `reps` can be recovered by subtracting the rank of the current frame from the total number of `reps`. More precisely, if e_1 is a function with frame S_1, then the non-induced `reps` 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 `reps` in the ILP objective to greatly diminish the frequency of ambiguity in practice (although it raises issues in other cases, see Section 10).

To check for ambiguity, we discriminate on the binary variables used to enforce the $M \veebar R$ constraints; see section 10.3 for more details.

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 [Bailly et al. 2023], 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 9 contains an example where the elaborated program is ill-typed when considering sizes.

[8]Actually, in the implementation, applications are annotated with triples of the form $\triangle(M, R, S)$ 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.
[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 (\r i -> r*r + i*i) phiR phiI
  let as = map3 (\x_e y_e z_e ->
            map (2*pi*)
              (map3 (\kx_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 (\a -> sum(map2 (*) phiM (map cos a))) as
  let qi = map (\a -> 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.

Fig. 16. 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.

Concretely, we first infer ground types via the first two phases of Figure 15, 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 [Bailly et al. 2023], 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 ranks are known. It is perhaps possible to encode this situation as a form of conditional rule in the vein of [Vytiniotis et al. 2011], but we did not find it necessary in order to obtain a useful system.

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 7 except only non-induced `rep`s are elaborated due to rep fusion.

9 Evaluation

We evaluate the practical merits of AUTOMAP on a collection of Futhark programs ported from the benchmark suites Parboil [Stratton et al. 2012], PBBS [Anderson et al. 2022], FinPar [Andreetta et al. 2016], Accelerate [Chakravarty et al. 2011], and Rodinia [Che et al. 2009]. 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 18.

In principle, we could expect that any expression `map` $f\ x$ could be replaced with $f\ x$, as AUTOMAP allows the `map` to be 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

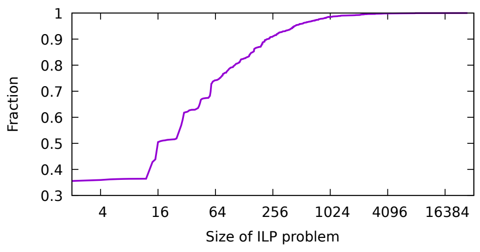

Fig. 17. Proportion of ILP problems that have less than some given number of constraints. Note that almost all problems contain at most 1024 constraints.

Number of programs: 67
Change in lines of code: 8623 ⇒ 8517
Change in `maps`: 467 ⇒ 213
Largest ILP size: 28104 constraints
Median ILP size: 18 constraints
Mean ILP size: 121 constraints
Mean type checking slowdown: 2.50×

Fig. 18. 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.

such as

$$\texttt{map}\ (\lambda x.\texttt{map}\ (*x)\ ys)\ xs$$

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

$$\texttt{map}\ (\lambda x.\, x * ys)\ xs$$

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

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

which is elaborated by Automap to `map` $(*)\ xs\ ys$, 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 8.3).

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

Figure 16 shows an example based on the `mri-q` benchmark from Parboil. The original program is shown in Figure 16a, with a total of ten instances of `map`, somewhat cluttering the program. The modified program on Figure 16b contains no explicit `maps`, 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.

9.1 Quantifying `maps`

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.[10]

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

[10]Type annotations refine the constraints in the ILP and can thereby eliminate additional solutions/disambiguate expressions.

analysis; specifically on constructing and solving ILP problems. Compared to the unmodified type checker, we see less than a 3× slowdown on most programs. The outlier is `myocyte`, which is about 13× 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 produces an ILP program with 28104 constraints, which is slow to solve. This could be optimized by not generating constraints for applications whose implicit `maps` and `reps` 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 17.

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 `maps` and `reps` 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 `maps`/`reps` (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, `maps` and `reps` are also inserted to make these constructs explicit as well.

10 Future Work

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 `|>`) 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)`

These elaborations have the same size, due to not counting induced `reps` (Section 8.2). Piping is a common pattern, so better interaction with AUTOMAP is desirable—possibly by refining the cost function to count induced `reps` in those cases where the result is otherwise ambiguous.

10.2 Solving Constraints Locally

As mentioned in Section 9.2, there are also opportunities to reduce the size of the ILP program by eliding constraints for applications that can be locally solved.

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 $\sum_{Q \in \{M_0,R_0,\ldots,M_n,R_n\}} \left| s_\mathrm{r}(Q) - s'_\mathrm{r}(Q) \right| \geq 1$. This constraint can be linearlized [Tsai et al. 2008], 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 $M \veebar R$ constraint (see section 6.4) may be discriminated on instead: $\sum_{Q \in \{M_0,R_0,\ldots,M_n,R_n\}} \left| s_\mathrm{r}(b_Q) - s'_\mathrm{r}(b_Q) \right| \geq 1$. 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

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

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 $\sum_{Q \in \{M_0, R_0, \ldots, M_n, R_n\}} \left| s_r(b_Q) - s'_r(b_Q) \right| \geq 1$ 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 [Danna et al. 2007; Danna and Woodruff 2009].

11 Related Work

11.1 Data Parallelism

NumPy [Harris et al. 2020] 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,[11] 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 can always be rewritten with explicit `maps` and `reps` 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* [Iverson 1962], APL only allows broadcasting (or more properly, scalar extension [Hui and Kromberg 2020]) 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 [Elsman and Dybdal 2014; Guibas and Wyatt 1978]. The limitation to scalars also exist in work by Thatte [Thatte 1991], 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 [Šinkarovs et al. 2023; Šinkarovs and Scholz 2022]. Also related to this work is the work on Remora [Slepak et al. 2014], 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 [Slepak et al. 2018]. Gibbons has shown how to encode rank polymorphism in Glasgow Haskell through the use of Naperian functors [Gibbons 2017], 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.

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

11.2 Type Systems and Type Inference

The Hindley-Damas-Milner type system [Damas 1984; Hindley 1969; Milner 1978] 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 [Odersky et al. 1999]. 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 [Garrigue 2004]. The trouble with local let-generalization has led a number of papers to propose that "let should not be generalized" [Vytiniotis et al. 2011, 2010]. 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 paper, 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 [Foster et al. 2002; Jones 1994; Sulzmann and Hudak 2000], including work on implementing type classes [Peterson and Jones 1993]. Whereas AUTOMAP is based on a constraint-based type system, constraints are local to function definitions, which simplifies the type system significantly compared to other work.

11.3 Implicit Program Constructs

Implicit program constructs, like AUTOMAP, are found in several programming languages. Implicit parameters are known from Haskell [Lewis et al. 2000] and Scala [Odersky et al. 2017] where in the latter they are used to support type classes [Odersky et al. 2017; Oliveira et al. 2010]. 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 [Křikava et al. 2019]. An overview of implicit programming constructs is provided by Jeffery's PhD thesis.

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 shown 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 `maps`, 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 9 is available on Zenodo [Schenck et al. 2024].

References

Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), V2. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 445–447. https://doi.org/10.1145/3503221.3508422

Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. 2016. FinPar: A Parallel Financial Benchmark. *ACM Trans. Archit. Code Optim.* 13, 2, Article 18 (June 2016), 27 pages. https://doi.org/10.1145/2898354

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

Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. 3–14. https://doi.org/10.1145/1926354.1926358

S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

Luis Damas. 1984. *Type assignment in programming languages.* Ph. D. Dissertation. The University of Edinburgh.

Emilie Danna, Mary Fenelon, Zonghao Gu, and Roland Wunderling. 2007. Generating multiple solutions for mixed integer programming problems. In *International Conference on Integer Programming and Combinatorial Optimization*. Springer, 280–294. https://doi.org/10.1007/978-3-540-72792-7_22

Emilie Danna and David L Woodruff. 2009. How to select a small set of diverse solutions to mixed integer programming problems. *Operations Research Letters* 37, 4 (2009), 255–260. https://doi.org/10.1016/j.orl.2009.03.004

Martin Elsman and Martin Dybdal. 2014. 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* (Edinburgh, United Kingdom) *(ARRAY'14)*. Association for Computing Machinery, New York, NY, USA, 101–106. https://doi.org/10.1145/2627373.2627390

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-Sensitive Type Qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/512529.512531

Jacques Garrigue. 2004. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming*. Springer, 196–213. https://doi.org/10.1007/978-3-540-24754-8_15

Jeremy Gibbons. 2017. 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, Berlin, Heidelberg, 556–583. https://doi.org/10.1007/978-3-662-54434-1_21

Leo J. Guibas and Douglas K. Wyatt. 1978. Compilation and delayed evaluation in APL. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) *(POPL '78)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/512760.512761

Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Troels Henriksen and Martin Elsman. 2021. Towards Size-Dependent Types for Array Programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Virtual, Canada) *(ARRAY 2021)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3460944.3464310

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. 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* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society (AMS)* (1969). https://doi.org/10.2307/1995158

Roger K. W. Hui and Morten J. Kromberg. 2020. APL since 1978. *Proc. ACM Program. Lang.* 4, HOPL, Article 69 (jun 2020), 108 pages. https://doi.org/10.1145/3386319

Kenneth E. Iverson. 1962. *A programming language.* John Wiley & Sons, Inc., USA. https://doi.org/10.1145/1460833.1460872

Alexander Jeffery. 2020. *On Implicit Program Constructs.* Ph. D. Dissertation. University of Sussex.

Mark P. Jones. 1994. A theory of qualified types. *Science of Computer Programming* 22, 3 (1994), 231–256. https://doi.org/10.1016/0167-6423(94)00005-0

Filip Křikava, Heather Miller, and Jan Vitek. 2019. Scala implicits are everywhere: A large-scale study of the use of scala implicits in the wild. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28. https://doi.org/10.1145/3360589

Jeffrey R Lewis, John Launchbury, Erik Meijer, and Mark B Shields. 2000. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 108–118. https://doi.org/10.1145/325694.325708

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* (1978). https://doi.org/10.1016/0022-0000(78)90014-4

Robin Milner. 1997. *The definition of standard ML: revised.* MIT press. https://doi.org/10.7551/mitpress/2319.001.0001

Greg Morrisett. 1995. *Compiling with Types.* Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicitly: Foundations and applications of implicit function types. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29. https://doi.org/10.1145/3158130

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

Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type classes as objects and implicits. *ACM Sigplan Notices* 45, 10 (2010), 341–360. https://doi.org/10.1145/1932682.1869489

John Peterson and Mark Jones. 1993. Implementing Type Classes. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 227–236. https://doi.org/10.1145/155090.155112

Robert Schenck. 2024. AUTOMAP Supplementary Material. https://doi.org/10.5281/zenodo.13628289

Robert Schenck, Nikolaj Hey Hinnerskov, Troels Henriksen, Magnus Madsen, and Martin Elsman. 2024. *futhark-oopsla24*. https://doi.org/10.5281/zenodo.12775308

Justin Slepak, Panagiotis Manolios, and Olin Shivers. 2018. Rank polymorphism viewed as a constraint problem. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (Philadelphia, PA, USA) *(ARRAY 2018)*. Association for Computing Machinery, New York, NY, USA, 34–41. https://doi.org/10.1145/3219753.3219758

Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. 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, 27–46. https://doi.org/10.1007/978-3-642-54833-8_3

John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

Martin Franz Sulzmann and Paul Hudak. 2000. *A General Framework for Hindley/Milner Type Systems with Constraints.* Ph. D. Dissertation. USA. AAI9973781.

Satish Thatte. 1991. A type system for implicit scaling. *Science of Computer Programming* 17, 1 (1991), 217–245. https://doi.org/10.1016/0167-6423(91)90040-5

Jung-Fa Tsai, Ming-Hua Lin, and Yi-Chung Hu. 2008. Finding multiple solutions to general integer linear programs. *European Journal of Operational Research* 184, 2 (2008), 802–809. https://doi.org/10.1016/j.ejor.2006.11.024

Artjoms Šinkarovs, Thomas Koopman, and Sven-Bodo Scholz. 2023. Rank-Polymorphism for Shape-Guided Blocking. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing* (Seattle, WA, USA) *(FHPNC 2023)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3609024.3609410

Artjoms Šinkarovs and Sven-Bodo Scholz. 2022. Parallel Scan as a Multidimensional Array Problem. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (San Diego, CA, USA) *(ARRAY 2022)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3520306.3534500

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn (X) Modular type inference with local assumptions. *Journal of functional programming* 21, 4-5 (2011), 333–412. https://doi.org/10.1017/S0956796811000098

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Madrid, Spain) *(TLDI '10)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/1708016.1708023

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (nov 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Received 2024-04-06; accepted 2024-08-18