Towards Size-Dependent Types for Array Programming

Troels Henriksen
DIKU
University of Copenhagen
Copenhagen, Denmark
athas@sigkill.dk

Martin Elsman
DIKU
University of Copenhagen
Copenhagen, Denmark
mael@di.ku.dk

Abstract

We present a type system for expressing size constraints on array types in an ML-style type system. The goal is to detect shape mismatches at compile-time, while being simpler than full dependent types. The main restrictions is that the only terms that can occur in types are array sizes, and syntactically they must be variables or constants. For those programs where this is not sufficient, we support a form of existential types, with the type system automatically managing the requisite book-keeping. We formalise a large subset of the type system in a small core language, which we prove sound. We also present an integration of the type system in the high-performance parallel functional language Futhark, and show on a collection of 44 representative programs that the restrictions in the type system are not too problematic in practice.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**.

Keywords: type systems, parallel programming, functional programming

ACM Reference Format:
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 (ARRAY '21), June 21, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460944.3464310

1 Introduction

Array programming often involves functions whose arguments must satisfy certain *shape constraints*. For example, when multiplying two matrices, the column count of the former must match the row count of the latter. Most of today's programming languages do not support compile-time checking of such constraints, instead deferring them to runtime. As with all dynamic type checking, this can result in frustrating crashes.

Dependently typed languages allow us to encode the sizes of arrays and have the type checker enforce the shape constraints. From the perspective of array programming, current dependently typed languages have two problems:

1. They are typically not very fast, where array programs are often performance-critical.
2. Full dependent types are complicated, and array programmers may not need or want all of their capabilities. In particular, working with *existential sizes*, as produced by operations such as filtering, requires deeper understanding of type theory.

In this paper, we investigate the foundations of a type system that supports *size-dependent types*. Our goal is to construct a programming language in the ML family that allows a "conventional" style of programming, while still verifying shape constraints statically. We have the following concrete design goals for size-dependent types:

1. It must be possible to write functions that operate on multi-dimensional arrays, where we can express equality constraints on the dimensions of these arrays. When a function is applied, the type checker must verify that the arguments satisfy the constraints.
2. Any book-keeping involved in handling existential sizes should be done automatically.
3. Multidimensional arrays should be handled as "arrays of arrays", while still guaranteeing that all arrays are regular.[1]. This permits parametric polymorphism, which allows functions such as `map` to have their conventional behaviour.

[1]A regular multidimensional array is one where all rows have the same shape. These are sometimes also called *rectangular arrays*, and are in opposition to *jagged arrays*, which we do not support.

4. The type system should not impede a high-performance implementation of the language. In particular it must not require complicated value representations or maintenance of complex runtime structures.

We value simplicity over completeness. As we shall see, we place limitations on the form of sizes which means some programs will require dynamic shape checking, although such checks will always be syntactically explicit *size coercions* that are inserted by the programmer when needed. An intentional *non-goal* is that we are not trying to statically eliminate out-of-bounds indexing.

We begin in Section 2 by defining F, a small core language that we use to illustrate the mechanics of the type system. In Section 3 we prove soundness for F, which implies the absence of shape errors at runtime. F is a simplified language, and does not yet fulfill all our goals—in particular, it is monomorphic. However, Section 5 demonstrates how we have added size types to the data parallel programming language Futhark [13], which has full support for parametric polymorphism. This not only shows that the type system, despite its simplicity, is expressive enough to handle real programs, but also that it does not provide obstacles to high-performance implementation. Section 6 discusses our experience with using size types in practice, in particular whether they are flexible enough to handle a wide variety of parallel benchmark problems. Finally, Section 7 discusses how our approach relates to prior work, with a particular focus on how it fits into the array language tradition.

2 A Language with Size-Dependent Types

In this section, we present a language F with support for size types. The language is simplistic in a variety of ways. First of all, it does not support type polymorphism, which we shall treat later in Section 4. Second, the language assumes that expressions are *flattened* so that expressions that allocate existentially sized arrays are bound by explicit `let`-constructs. Thus, if we assume the availability of a type-monomorphic (but size-polymorphic) `map`-function and a function `iota` that takes an integer n and returns an index-array of size n, we also assume that an expression

$$\texttt{map}\ (\lambda x.x + 1)\ (\texttt{iota}\ 5)$$

has been converted to the form

$$\texttt{let}\ a = \texttt{iota}\ 5\ \texttt{in}\ \texttt{map}\ (\lambda x.x + 1)\ a.$$

This conversion simplifies the framework and can easily be implemented in the frontend of a compiler. Although, the type system supports an implicit `let`-binding construct, as used in the `map` example, an explicit version is available, which eases type inference and which brings existential sizes into scope at expression level as `int` variables. In the explicit version of the `let` construct, the expression becomes

$$\texttt{let}\ [n]\ a : [n]\texttt{int} = \texttt{iota}\ 5\ \texttt{in}\ \texttt{map}\ (\lambda x.x + n)\ a.$$

d	::=		**Size sorts**
	|	n	constant
	|	x	variable
τ	::=		**Basic types**
	|	$\texttt{int}$	integer
	|	$\texttt{bool}$	boolean
	|	(τ, τ')	pair
	|	$[d]\tau$	array
	|	$(x : \tau) \rightarrow \mu$	function
μ	::=		**Return types**
	|	$\exists x.\mu$	existential size
	|	τ	basic type
σ	::=		**Type schemes**
	|	τ	basic type
	|	$\star$	abstract size type
e	::=		**Expressions**
	|	n	constant integer
	|	$\texttt{true} \mid \texttt{false}$	constant boolean
	|	x	variable
	|	$\lambda(x : \tau).e$	function
	|	$e\ e$	application
	|	$[e, \cdots, e]$	array
	|	$\texttt{iota}\ e$	index array
	|	$e[e]$	array index
	|	(e, e)	pair
	|	$\texttt{fst}\ e \mid \texttt{snd}\ e$	projection
	|	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	conditional
	|	$e \triangleright \tau$	dynamic type coercion
	|	$\texttt{let}\ p = e\ \texttt{in}\ e$	let
p	::=	x	
	|	$[\bar{x}]\ x : \tau$	

Figure 1. Grammar for F.

Whereas the type of this entire expression will be $\exists x.[x]\texttt{int}$, where x is an existentially bound size variable, inside the scope of the `let`-construct, it will be known that a and the result of the `map` application have the same size.

2.1 Types and Expressions

We assume a denumerably infinite set of program variables, ranged over by x, y, and f. The grammars for size sorts (d), basic types (τ), and return types (μ) are defined in Figure 1.

Size sorts appear in array types of the form $[d]\tau$, which denote arrays of size d containing elements of type τ. A function type $(x : \tau) \rightarrow \mu$ allows for x to appear in μ, which is useful for expressing a dependency between a parameter of type `int` and the size of an array in the result.

For basic types on the form $(x : \tau) \to \mu$, we consider x bound in μ. For return types on the form $\exists x.\mu$, we consider x bound in μ. Respectively, basic types and return types are considered identical up to renaming of bound variables. Return types are also considered identical up to removal of bound existential variables that do not occur free in the body.

Contexts, ranged over by Γ, map program variables to type schemes, which, in the simple setting that lacks parametric size- and type-polymorphism, amounts to being a type τ or a type scheme of the form $\star$, which denote an abstract size.

When $\bar{x} = x_1, \cdots, x_n$ and $\bar{\sigma} = \sigma_1, \cdots, \sigma_n$, we write $\bar{x} : \bar{\sigma}$ for specifying the type assumption $x_1 : \sigma_1, \cdots, x_n : \sigma_n$. We shall also sometimes write $\bar{x} : \bar{c}$, where c is some basic type scheme such as $\star$ or `int`, to denote the type assumption $x_1 : c, \cdots, x_n : c$. We shall use similar notation for size assumptions and dynamic environments (defined later).

Notice that we assume that whenever we have a context on the form $\Gamma, x : \tau$, we have $x \notin \mathrm{fv}(\Gamma)$. In the rules that follow, this property can most often be ensured by renaming of bound variables (i.e., alpha renaming).

Whenever $\bar{x} = x_1, \cdots, x_n$, we often write $\exists \bar{x}.\mu$ to denote the return type $\exists x_1. \cdots \exists x_n.\mu$ (similarly for universal size quantification.)

Type schemes of the form $\star$ play a special role. Variables bound to such type schemes have no dynamic representation. In fact, in the simple setting of F, programmers have no way of referring to such abstract sizes even though they form the basis for establishing static shape properties and static shape equalities of runtime values. For instance, type schemes of the form $\star$ are used for establishing, for instance, that a certain expression returns a value of return type $\exists x.([x]\texttt{int}, [x](\texttt{bool}, \texttt{int}))$ for which inhabitants are pairs of equally sized arrays, where the first array contains integers and the second array contains pairs of a boolean and an integer.

Basic types (τ), return types (μ), and size sorts (d) are *well-formed* under assumptions Γ, written $\Gamma \vdash \tau$ **ok**, $\Gamma \vdash \mu$ **ok**, and $\Gamma \vdash d$ **ok**, respectively, if the particular judgement can be derived using the rules in Figure 2. A context Γ' is well-formed under assumptions Γ, written $\Gamma \vdash \Gamma'$ **ok**, if $\Gamma \vdash \sigma$ **ok** for all σ in the range of Γ'. We shall write $\vdash \Gamma$ **ok** to mean $\Gamma \vdash \Gamma$ **ok**. When μ is some return type (or type), we write $\Gamma \vdash \mu\ \underline{\mathbf{ok}}$ to mean $\Gamma \vdash \mu$ **ok** and $\forall x \in \mathrm{fv}(\mu), \Gamma(x) \neq \star$. Expressions ($e$) and patterns ($p$) are defined according to the grammar in Figure 1.

The expression language has support for basic constants, including integers and booleans, lambda abstraction and function application, pairs and projections, and conditional expressions. The language also has support for immediate arrays and it features an explicit index-space operation `iota`, which, given an integer argument n, creates an array with n integers ranging from 0 to $n - 1$. F also features array indexing, which fails dynamically if an index is out of bounds. Further, the language features two kinds of `let` constructs:

Size sorts $\boxed{\Gamma \vdash d\ \mathbf{ok}}$

$$\frac{}{\Gamma \vdash n\ \mathbf{ok}} \qquad \frac{\Gamma(x) = \texttt{int}}{\Gamma \vdash x\ \mathbf{ok}} \qquad \frac{\Gamma(x) = \star}{\Gamma \vdash x\ \mathbf{ok}}$$

Types $\boxed{\Gamma \vdash \tau\ \mathbf{ok}}$

$$\frac{}{\Gamma \vdash \texttt{int}\ \mathbf{ok}} \qquad \frac{}{\Gamma \vdash \texttt{bool}\ \mathbf{ok}}$$

$$\frac{\Gamma \vdash d\ \mathbf{ok} \quad \Gamma \vdash \tau\ \mathbf{ok}}{\Gamma \vdash [d]\tau\ \mathbf{ok}} \qquad \frac{\Gamma \vdash \tau\ \mathbf{ok} \quad \Gamma \vdash \tau'\ \mathbf{ok}}{\Gamma \vdash (\tau, \tau')\ \mathbf{ok}}$$

$$\frac{\Gamma \vdash \tau\ \mathbf{ok} \quad \Gamma \vdash \mu\ \mathbf{ok}}{\Gamma \vdash (x : \tau) \to \mu\ \mathbf{ok}} \qquad \frac{\Gamma, x : \star \vdash \mu\ \mathbf{ok}}{\Gamma \vdash (x : \texttt{int}) \to \mu\ \mathbf{ok}}$$

Return types $\boxed{\Gamma \vdash \mu\ \mathbf{ok}}$

$$\frac{\Gamma, x : \star \vdash \mu\ \mathbf{ok}}{\Gamma \vdash \exists x.\mu\ \mathbf{ok}} \qquad \frac{\Gamma \vdash \tau\ \mathbf{ok} \quad \mu = \tau}{\Gamma \vdash \mu\ \mathbf{ok}}$$

Type schemes $\boxed{\Gamma \vdash \sigma\ \mathbf{ok}}$

$$\frac{}{\Gamma \vdash \star\ \mathbf{ok}} \qquad \frac{\Gamma \vdash \tau\ \mathbf{ok} \quad \sigma = \tau}{\Gamma \vdash \sigma\ \mathbf{ok}}$$

Figure 2. Well-formedness of size sorts, basic types, return types, and type schemes.

one that implicitly "opens" existentially bound size variables (making them available only to meta-level reasoning), and one that explicitly extracts sizes for existentially bound size variables (bringing them into scope as variables of type `int`).

Finally, the language features a type coercion construct of the form $e \triangleright \tau$, which allows the programmer to coerce an array value to be of a specific shape, checked dynamically.

For simplicity, the rules for the dynamic semantics shall deal only with succeeding evaluations, which means that the type safety property that we shall prove only relates to programs for which a dynamic derivation exists. We consider it future work to further distinguish between the possible dynamic effects and to establish a guarantee for termination.

2.2 Type System

Typing rules for expressions allow inferences among sentences on the form $\Gamma \vdash e : \mu$, which are read "under assumptions Γ, the expression e has return type μ."

The typing rules are shown in Figure 3. In general, existential quantification in return types is used for modeling existential sizes. In rule T-LET, existentially bound sizes are implicitly opened in the scope of the `let` construct and closed again when forming the return type of the entire expression. In rule T-IOTAD and rule T-APPD, the type system supports

Expressions $\boxed{\Gamma \vdash e : \mu}$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ [\textsc{t-true}] \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ [\textsc{t-false}] \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ [\textsc{t-int}] \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ [\textsc{t-var}]$$

$$\frac{\Gamma \vdash e_1 : \exists \bar{x}_1.\tau_1 \quad \Gamma \vdash e_2 : \exists \bar{x}_2.\tau_2}{\Gamma \vdash (e_1, e_2) : \exists \bar{x}_1\bar{x}_2.(\tau_1, \tau_2)}\ [\textsc{t-pair}] \qquad \frac{\Gamma \vdash e_1 : \tau \quad \cdots \quad \Gamma \vdash e_n : \tau}{\Gamma \vdash [e_1, \cdots, e_n] : [n]\tau}\ [\textsc{t-array}]$$

$$\frac{\Gamma \vdash e : \exists \bar{x}.(\tau_1, \tau_2)}{\Gamma \vdash \texttt{fst}\ e : \exists \bar{x}.\tau_1}\ [\textsc{t-fst}] \qquad \frac{\Gamma \vdash e : \exists \bar{x}.(\tau_1, \tau_2)}{\Gamma \vdash \texttt{snd}\ e : \exists \bar{x}.\tau_2}\ [\textsc{t-snd}] \qquad \frac{\Gamma \vdash e_1 : \exists \bar{x}.[d]\tau \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1[e_2] : \exists \bar{x}.\tau}\ [\textsc{t-index}]$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \mu \quad \Gamma \vdash e_2 : \mu}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \mu}\ [\textsc{t-if}] \qquad \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{iota}\ e : \exists x.[x]\texttt{int}}\ [\textsc{t-iota}] \qquad \frac{\Gamma \vdash d : \texttt{int}}{\Gamma \vdash \texttt{iota}\ d : [d]\texttt{int}}\ [\textsc{t-iotad}]$$

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash \tau\ \underline{\mathbf{ok}} \quad \tau = \tau'\{\bar{d}/\bar{x}\}}{\Gamma \vdash e \triangleright \tau : \tau}\ [\textsc{t-coerce}] \qquad \frac{\Gamma, x : \tau \vdash e : \mu \quad \Gamma \vdash \tau\ \underline{\mathbf{ok}}}{\Gamma \vdash \lambda(x : \tau).e : (x : \tau) \rightarrow \mu}\ [\textsc{t-lam}]$$

$$\frac{\Gamma \vdash e : (x : \tau) \rightarrow \mu \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \exists x.\mu}\ [\textsc{t-app}] \qquad \frac{\Gamma \vdash e : (x : \texttt{int}) \rightarrow \mu \quad \Gamma \vdash d : \texttt{int}}{\Gamma \vdash e\ d : \mu\{d/x\}}\ [\textsc{t-appd}]$$

$$\frac{\Gamma \vdash e : \exists \bar{x}.\tau \quad \Gamma, \bar{x} : \bar{\star}, x : \tau \vdash e' : \mu}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \exists \bar{x}x.\mu}\ [\textsc{t-let}] \qquad \frac{\Gamma \vdash e : \exists \bar{x}.\tau \quad \Gamma \vdash \exists \bar{x}.\tau\ \underline{\mathbf{ok}} \quad \Gamma, \bar{x} : \overline{\texttt{int}}, x : \tau \vdash e' : \mu}{\Gamma \vdash \texttt{let}\ [\bar{x}]\ x : \tau = e\ \texttt{in}\ e' : \exists \bar{x}x.\mu}\ [\textsc{t-let-sz}]$$

Figure 3. Typing rules for expressions.

explicit size applications, which allows the programmer to establish a connection between a computed value and the size of an array. A dual feature of the type system is provided by rule T-LET-SZ, which allows the programmer to extract the size of an existentially sized array and use the size in an expression. Finally, rule T-COERCE allows the programmer to specify a type for an expression, which will be checked dynamically against the actual value. Notice that we allow only type coercions that change the static dimensions of an otherwise unchanged array type. Statically, this restriction is modelled by the existence of a substitution that maps size variables $\bar{x}$ in the type τ' into type descriptors $\bar{d}$ in the type τ. A set of dynamic rules (as we shall see in the next section) ensures that the involved sizes are identical at runtime.

2.3 Dynamic Semantics

Values (v) are defined according to Figure 4, where dynamic environments (ρ) are maps from program variables to values:

Notice here that the goal of our type system is not to rule out index and type coercion errors which means that even well-typed expressions, may fail to evaluate to a value.

The dynamic semantics for the language evaluates an expression e to a result value v in a dynamic environment ρ. The dynamic semantics is specified by a set of inference rules, which allow inferences among sentences of the form

v	$::=$		Values
	$\mid$	$n \mid \texttt{true} \mid \texttt{false}$	basic constant
	$\mid$	$\langle x, e, \rho \rangle$	closure
	$\mid$	$[v, \cdots, v]$	array
	$\mid$	(v, v)	pair

Figure 4. Grammar for values in F.

$\rho \vdash e \rightsquigarrow v$, which are read "The expression e evaluates to v in the dynamic environment ρ".

The rules make use of a set of dynamic *size-matching rules* that allow for extracting sizes from array values at runtime. These rules take the form $\tau \vdash_x v \rightsquigarrow r$, where τ is a type, x is a variable, v is a value, and r is either $\bullet$ (denoting that x does not appear appropriately in τ) or a size n extracted from an array value in v, guided by the first preorder location of x in τ.[2] A rule for extracting multiple sizes takes the form $\tau \vdash_{\bar{x}} v \rightsquigarrow \bar{n}$. The size-matching rules are given in Figure 5.

The rules for the dynamic semantics also make use of a set of dynamic *size-checking rules* that allow for dynamic checking of array sizes according to a type. These rules take

[2] All found occurrences of a size variable should result in the same value, so we just use the first one we find.

Single variable $\boxed{\tau \vdash_x v \rightsquigarrow n/\bullet}$

$$\frac{}{\texttt{int} \vdash_x v \rightsquigarrow \bullet}\ [\textsc{m-int}] \qquad \frac{}{\texttt{bool} \vdash_x v \rightsquigarrow \bullet}\ [\textsc{m-bool}] \qquad \frac{}{(x : \tau) \to \mu \vdash_x v \rightsquigarrow \bullet}\ [\textsc{m-fun}]$$

$$\frac{\tau_1 \vdash_x v_1 \rightsquigarrow n}{(\tau_1, \tau_2) \vdash_x (v_1, v_2) \rightsquigarrow n}\ [\textsc{m-pair1}] \qquad \frac{\tau_1 \vdash_x v_1 \rightsquigarrow \bullet \quad \tau_2 \vdash_x v_2 \rightsquigarrow r}{(\tau_1, \tau_2) \vdash_x (v_1, v_2) \rightsquigarrow r}\ [\textsc{m-pair2}]$$

$$\frac{}{[x]\tau \vdash_x [v_1, \cdots, v_n] \rightsquigarrow n}\ [\textsc{m-arr1}] \qquad \frac{x \neq y \quad \tau \vdash_x v_1 \rightsquigarrow r}{[y]\tau \vdash_x [v_1, \cdots, v_n] \rightsquigarrow r}\ [\textsc{m-arr2}]$$

Multiple variables $\boxed{\tau \vdash_{\bar{x}} v \rightsquigarrow \bar{n}}$

$$\frac{\tau \vdash_{x_1} v \rightsquigarrow n_1 \quad \cdots \quad \tau \vdash_{x_m} v \rightsquigarrow n_m}{\tau \vdash_{(x_1 \cdots x_m)} v \rightsquigarrow (n_1 \cdots n_m)}\ [\textsc{m-multi}]$$

Figure 5. Dynamic size-matching for the language.

Values $\boxed{\rho \vdash v \triangleright \tau}$

$$\frac{}{\rho \vdash n \triangleright \texttt{int}}\ [\textsc{c-int}] \qquad \frac{v = \texttt{true} \vee v = \texttt{false}}{\rho \vdash v \triangleright \texttt{bool}}\ [\textsc{c-bool}] \qquad \frac{\rho \vdash v_1 \triangleright \tau_1 \quad \rho \vdash v_2 \triangleright \tau_2}{\rho \vdash (v_1, v_2) \triangleright (\tau_1, \tau_2)}\ [\textsc{c-pair}]$$

$$\frac{\rho(x) = n \quad \rho \vdash v_1 \triangleright \tau \quad \cdots \quad \rho \vdash v_n \triangleright \tau}{\rho \vdash [v_1, \cdots, v_n] \triangleright [x]\tau}\ [\textsc{m-arr-x}] \qquad \frac{\rho \vdash v_1 \triangleright \tau \quad \cdots \quad \rho \vdash v_n \triangleright \tau}{\rho \vdash [v_1, \cdots, v_n] \triangleright [n]\tau}\ [\textsc{m-arr-n}]$$

Figure 6. Dynamic size-checking for the language.

the form $\rho \vdash v \triangleright \tau$, where ρ is a dynamic environment, v is a value, and τ is a type. The size-checking rules are given in Figure 6. The dynamic semantics rules are given in Figure 7.

3 Metatheory for F

This section builds up a metatheory for F, culminating in a soundness proof.

As mentioned earlier, for simplicity, the soundness result concerns well-typed and well-terminating programs. By restricting the result to well-terminating programs, we have pushed aside the technicalities involved in propagating errors in the rules for the dynamic semantics. Notice that F does not contain any form of unbounded recursion (all the second-order array combinators can be defined inductively), thus, in priciple, by propagating out-of-bound array-index errors and failed type-matching errors, explicitly, in the rules for the dynamic semantics, termination can be established formally; we will leave it up to future work to consider this possibility.

The soundness result that we establish here gives a guarantee that when an expression evaluates to a value containing arrays, the arrays are described by the type of the expression. As a consequence, because size quantifiers cannot appear underneath array type constructors, all multi-dimensional arrays that are constructed during evaluation are regular (all rows have the same shape). Moreover, the soundness result allows the compiler to omit a large number of size compatibility checks (which are established statically by the type system).

The remainder of this section is quite dense, and readers are free to skip the detailed proofs, as later sections do not directly build on the actual proofs (only on the properties they concern).

3.1 Properties of the Type System

The type system possesses a number of properties that are important for ensuring well-formedness of types. The following two propositions are established by straightforward induction on the structure of the type-well-formedness derivation and the typing derivation, respectively.

Proposition 3.1 (Type well-formedness preserved under size substitution). *Let* $\tau = \star$ *or* $\tau = \texttt{int}$. *If* $\Gamma, x : \tau \vdash \mu$ **ok** *and* $\Gamma \vdash d$ **ok** *then* $\Gamma \vdash \mu\{d/x\}$ **ok**.

Proposition 3.2 (Typing yields well-formed types). *If* $\Gamma \vdash \Gamma$ **ok** *and* $\Gamma \vdash e : \mu$ *then* $\Gamma \vdash \mu$ **ok**.

Expressions $\boxed{\rho \vdash e \rightsquigarrow v}$

$$\frac{}{\rho \vdash n \rightsquigarrow n}\;[\textsc{d-int}] \qquad \frac{}{\rho \vdash \texttt{true} \rightsquigarrow \texttt{true}}\;[\textsc{d-true}] \qquad \frac{}{\rho \vdash \texttt{false} \rightsquigarrow \texttt{false}}\;[\textsc{d-false}]$$

$$\frac{\rho(x) = v}{\rho \vdash x \rightsquigarrow v}\;[\textsc{d-var}] \qquad \frac{\rho \vdash e_1 \rightsquigarrow v_1 \quad \cdots \quad \rho \vdash e_n \rightsquigarrow v_n}{\rho \vdash [e_1, \cdots, e_n] \rightsquigarrow [v_1, \cdots, v_n]}\;[\textsc{d-array}] \qquad \frac{\rho \vdash e_1 \rightsquigarrow v_1 \quad \rho \vdash e_2 \rightsquigarrow v_2}{\rho \vdash (e_1, e_2) \rightsquigarrow (v_1, v_2)}\;[\textsc{d-pair}]$$

$$\frac{\rho \vdash e \rightsquigarrow (v_1, v_2)}{\rho \vdash \texttt{fst}\; e \rightsquigarrow v_1}\;[\textsc{d-fst}] \qquad \frac{\rho \vdash e \rightsquigarrow (v_1, v_2)}{\rho \vdash \texttt{snd}\; e \rightsquigarrow v_2}\;[\textsc{d-snd}] \qquad \frac{\begin{array}{c}\rho \vdash e_1 \rightsquigarrow [v_0, \ldots, v_{m-1}] \\ \rho \vdash e_2 \rightsquigarrow n \quad 0 \le n < m\end{array}}{\rho \vdash e_1[e_2] \rightsquigarrow v_n}\;[\textsc{d-index}]$$

$$\frac{}{\rho \vdash \lambda x.e \rightsquigarrow \langle x, e, \rho \rangle}\;[\textsc{d-lam}] \qquad \frac{\begin{array}{c}\rho \vdash e_1 \rightsquigarrow \langle x, e_0, \rho' \rangle \\ \rho \vdash e_2 \rightsquigarrow v' \quad \rho', x : v' \vdash e_0 \rightsquigarrow v\end{array}}{\rho \vdash e_1\; e_2 \rightsquigarrow v}\;[\textsc{d-app}] \qquad \frac{\begin{array}{c}\rho \vdash e \rightsquigarrow n \\ v = [0, \cdots, n-1]\end{array}}{\rho \vdash \texttt{iota}\; e \rightsquigarrow v}\;[\textsc{d-iota}]$$

$$\frac{\rho \vdash e \rightsquigarrow \texttt{true} \quad \rho \vdash e_1 \rightsquigarrow v}{\rho \vdash \texttt{if}\; e\; \texttt{then}\; e_1\; \texttt{else}\; e_2 \rightsquigarrow v}\;[\textsc{d-if-t}] \qquad \frac{\rho \vdash e \rightsquigarrow \texttt{false} \quad \rho \vdash e_2 \rightsquigarrow v}{\rho \vdash \texttt{if}\; e\; \texttt{then}\; e_1\; \texttt{else}\; e_2 \rightsquigarrow v}\;[\textsc{d-if-f}]$$

$$\frac{\begin{array}{c}\rho \vdash e \rightsquigarrow v \\ \rho, x : v \vdash e' \rightsquigarrow v'\end{array}}{\rho \vdash \texttt{let}\; x = e\; \texttt{in}\; e' \rightsquigarrow v'}\;[\textsc{d-let}] \qquad \frac{\begin{array}{c}\rho \vdash e \rightsquigarrow v \quad \tau \vdash_{\bar{x}} v \rightsquigarrow \bar{n} \\ \rho, \bar{x} : \bar{n}, x : v \vdash e' \rightsquigarrow v'\end{array}}{\rho \vdash \texttt{let}\; [\bar{x}]\; x : \tau = e\; \texttt{in}\; e' \rightsquigarrow v'}\;[\textsc{d-let-m}] \qquad \frac{\begin{array}{c}\rho \vdash e \rightsquigarrow v \\ \rho \vdash v \triangleright \tau\end{array}}{\rho \vdash e \triangleright \tau \rightsquigarrow v}\;[\textsc{d-coerce}]$$

Figure 7. Dynamic semantics for the language.

3.2 Relating Values and Types

In this section, we define a logical relation between values and types. To do so, we use δ to range over *size assignments*, which are mappings from program variables to sizes (i.e., integers). The logical relation is specified using a series of inductive definitions on the forms $\delta \models v : \sigma$ and $\delta \models v : \mu$, which are read "the value v has type scheme σ (or return type μ) under the size assignment δ."

We define a notion of *agreement* between size assignments on a set of size variables. When X is a set of size variables and δ and δ' are size assignments, we say that δ agrees with δ' on the set X, written $\delta' \approx_X \delta$, if $X \subseteq \mathrm{Dom}(\delta)$ and $X \subseteq \mathrm{Dom}(\delta')$ and $\delta(x) = \delta'(x)$, for all $x \in X$.

The notion of agreement, on a given set, is reflexive, transitive, and symmetric. Further, if $\delta' \approx_X \delta$ and $Y \subseteq X$ then $\delta' \approx_Y \delta$.

The logical relation is first defined inductively on types and return types:

Types and Return Types $\boxed{\delta \models v : \mu}$.

L-INT	$\delta \models n : \texttt{int}$
L-TRUE	$\delta \models \texttt{true} : \texttt{bool}$
L-FALSE	$\delta \models \texttt{false} : \texttt{bool}$
L-PAIR	$\delta \models (v_1, v_2) : (\tau_1, \tau_2)$ iff $\delta \models v_1 : \tau_1$ and $\delta \models v_2 : \tau_2$
L-ARR-N	$\delta \models [v_1, \cdots, v_n] : [n]\tau$ iff $\delta \models v_i : \tau, \forall i \in \{1..n\}$
L-ARR-X	$\delta \models [v_1, \cdots, v_n] : [x]\tau$ iff $\delta(x) = n$ and $\delta \models v_i : \tau, \forall i \in \{1..n\}$
L-CLOS	$\delta \models \langle x, e', \rho \rangle : (x : \tau') \to \mu$ iff $\forall v_1, v_2,\ (\delta \models v_1 : \tau'$ and $\rho, x : v_1 \vdash e' \rightsquigarrow v_2) \implies$ if $\tau' = \texttt{int}$ then $\delta, x : v_1 \models v_2 : \mu$ else $\delta \models v_2 : \mu$
L-RT	$\delta \models v : \exists x.\mu'$ iff $\exists n$ s.t. $\delta, x : n \models v : \mu'$

The logical relation extends to type schemes as follows:

Type Schemes $\boxed{\delta \models v : \sigma}$.

L-TS-TYPE	$\delta \models v : \sigma$ iff $\delta \models v : \tau$	$(\sigma = \tau)$
L-TS-★	$\delta \models n : \star$	

The logical relation extends to environments point-wise, but we take care to ensure that abstract size variables (type scheme $\star$) or variables of type `int`, appear as size assumptions in δ and appear, identically, as values in ρ.

We shall also make use of the following definitions of TyDom(Γ) and StarDom(Γ), which capture the set of variables in Γ that are associated with proper type, and the set of variables that are associated with abstract sizes (★):

$$\begin{array}{rcl} \mathrm{TyDom}(\Gamma) & = & \{x \in \mathrm{Dom}(\Gamma) \mid \Gamma(x) \neq \star\} \\ \mathrm{StarDom}(\Gamma) & = & \{x \in \mathrm{Dom}(\Gamma) \mid \Gamma(x) = \star\} \end{array}$$

Thus, for any Γ, we have $\mathrm{Dom}(\Gamma) = \mathrm{TyDom}(\Gamma) \cup \mathrm{StarDom}(\Gamma)$ and $\mathrm{TyDom}(\Gamma) \cap \mathrm{StarDom}(\Gamma) = \emptyset$.

Environments $\boxed{\delta \models \rho : \Gamma}$.

L-ENV $\delta \models \rho : \Gamma$ iff

1. $\mathrm{Dom}(\rho) = \mathrm{TyDom}(\Gamma)$ and $\vdash \Gamma$ **ok**
2. $\forall x \in \mathrm{TyDom}(\Gamma), \delta \models \rho(x) : \Gamma(x)$
3. $\forall x \in \mathrm{TyDom}(\Gamma), (\Gamma(x) = \mathtt{int} \implies \delta(x) = \rho(x))$
4. $\mathrm{StarDom}(\Gamma) \subseteq \mathrm{Dom}(\delta)$

The *free size variables* of an environment Γ, written fsv(Γ) is defined as follows:

$$\begin{array}{rcl} \mathrm{fsv}(\Gamma) & = & \{x \in \mathrm{Dom}(\Gamma) \mid \Gamma(x) = \mathtt{int} \vee \Gamma(x) = \star\} \\ & & \cup\ \{x \in \mathrm{fv}(\Gamma(y)) \mid y \in \mathrm{Dom}(\Gamma)\} \end{array}$$

Notice also that $\delta \models \rho : \Gamma$ entails $\vdash \Gamma$ **ok**, which is useful for ensuring well-formedness of type environments throughout the inductive proofs. Well-formed type environments are important for establishing, for instance, well-formedness of types, as we have seen earlier, which, allows us to reason about the identity of types. For instance, we can establish that if $x \notin \mathrm{Dom}(\Gamma)$ and $\Gamma \vdash \mu$ **ok** then $\mu = \exists x.\mu$ (because return types are considered identical up to removal of bound variables that do not occur in the body).

3.3 Logical Relation Extensibility

Before we state a soundness property for the type system, we first demonstrate an important property of the logical relation, namely that it supports an extension property saying that if $\delta \models \rho : \Gamma$ and $\delta' \approx_{\mathrm{fsv}(\Gamma)} \delta$ then $\delta' \models \rho : \Gamma$. This property is important for giving a soundness proof for the type system, based on simple induction principles.[3]

Proposition 3.3 (Logical Relation Extensibility)**.**

1. *If* $\delta \models v : \tau$ *and* $\delta' \approx_{\mathrm{fv}(\tau)} \delta$ *then* $\delta' \models v : \tau$.
2. *If* $\delta \models v : \mu$ *and* $\delta' \approx_{\mathrm{fv}(\mu)} \delta$ *then* $\delta' \models v : \mu$.
3. *If* $\delta \models v : \sigma$ *and* $\delta' \approx_{\mathrm{fv}(\sigma)} \delta$ *then* $\delta' \models v : \sigma$.
4. *If* $\delta \models \rho : \Gamma$ *and* $\delta' \approx_{\mathrm{fsv}(\Gamma)} \delta$ *then* $\delta' \models \rho : \Gamma$.

Proof. The main interesting case is the case for function types. The remaining cases either follow immediately or can be shown by straightforward induction. For details, see Appendix A. ◻

[3]In particular, the property is used in the proof cases for lambda abstraction and let-binding.

3.4 Size Matching and Size Checking Properties

In the proof of soundness of the size-typing rules, we will make use of a property of the dynamic size-matching rules. The following proposition holds:

Proposition 3.4 (Dynamic Size Matching)**.**

1. *If* $\delta \models v : \exists x.\tau$ *and* $\tau \vdash_x v \rightsquigarrow n$ *then* $\delta, x : n \models v : \tau$.
2. *If* $\delta \models v : \exists \bar{x}.\tau$ *and* $\tau \vdash_{\bar{x}} v \rightsquigarrow \bar{n}$ *then* $\delta, \bar{x} : \bar{n} \models v : \tau$.

Proof. Property 1 follows from simple induction over the structure of τ. Property 2 follows by repeated application of property 1. ◻

We will make use also of the following property of the size checking rules:

Proposition 3.5 (Dynamic Size Checking)**.** *If* $\delta \models v : \tau_1$ *and* $\tau = \tau_1\{\bar{d}/\bar{x}\}$ *and* $\rho \vdash v \triangleright \tau$ *and* $\rho(x) = \delta(x), \forall x \in \mathrm{fv}(\tau)$, *then* $\delta \models v : \tau$.

Proof. By induction on the structure of τ_1. ◻

3.5 Soundness of Size Typing

We can now state and prove a soundness property for the type system. The property gives us no guarantee that evaluation is strongly normalising. We shall return to this issue later in Section 3.6.

Proposition 3.6 (Soundness)**.** *If* $\Gamma \vdash e : \mu$ *and* $\delta \models \rho : \Gamma$ *and* $\rho \vdash e \rightsquigarrow v$ *then* $\delta \models v : \mu$.

Proof. By induction on the structure of the typing derivation. See Appendix A for details. ◻

3.6 Distinguishing Effects

By considering only value-terminating expressions, we have avoided specifying dynamic evaluation rules for propagating dynamic errors. There are a number of ways in which well-typed expressions may fail to evaluate. First, rule D-INDEX requires the index to be in-bound. Second, rule D-COERCE may hit an array size mismatch. Third, rule D-COERCE assumes that the result type does not contain any function types. Fourth, rule D-LET-SZ may fail either due to the presence of an empty array, due to the lack of a size name in the type, or due to a size name being present only below a function type.

Dealing properly with inner sizes of empty arrays requires that shape information is available dynamically in cases where an array can be empty. Whereas our Futhark implementation treats empty arrays properly (by annotating array values with shape information at runtime), we leave it to future work to provide a formal treatment of empty arrays.

Technically, it would not be difficult to establish a termination result based on a logical relation argument similar to the one we have presented here but changed to demand that there exists an evaluation derivation resulting in a value (or a dynamic error) that relates to the result type of the typing derivation. Such an argument would follow closely Tait's

τ	::=	$\cdots$	**Basic types**
	$\mid$	α	type variable
σ	::=	$\cdots$	**Type schemes**
	$\mid$	$\forall x.\sigma$	universal size quantification
	$\mid$	$\forall \alpha.\sigma$	universal type quantification
e	::=		**Expressions**
	$\mid$	$\texttt{let}\ f\ [\bar{x}]\ \bar{\alpha} = e\ \texttt{in}\ e$	polymorphic let

Figure 8. Grammar components for parametric polymorphism, extending those of Figure 1.

strong-normalisation argument for the simply-typed lambda calculus, which dates back to 1967 [25] and is later established as a fundamental proof technique [10] and adapted for a variety of uses [5].

For proving strong-normalisation in cases of inductively defined data-structures, logical relation techniques have been augmented with step-indexing techniques [2, 6], which serve to establish a well-defined logical relation. Such techniques could be useful also in the setting of array programming, where array sizes could serve as a means for controlling array slicing and looping in an inductively defined way, which could be used for providing strong-normalisation guarantees for a large class of array programs.

4 Universal Size- and Type-Quantification

For supporting universal size- and type-quantification (parametric polymorphism), the grammar for types, type schemes, and expressions is extended as shown in Figure 8. Note that the polymorphic let binding binds only a single name, and requires explicit quantification of both type- and size parameters. The type rules are given in Figure 9, where $\mathrm{ftv}(\Gamma, \tau)$ denotes the free type variables in Γ and τ.

With this extension, we can express the types of common parallel building blocks:

$$
\begin{array}{lcl}
\textbf{val}\ \texttt{length} & : & \forall x.[x]\alpha \rightarrow \texttt{int} \\
\textbf{val}\ \texttt{map} & : & \forall x.(\alpha \rightarrow \beta) \rightarrow [x]\alpha \rightarrow [x]\beta \\
\textbf{val}\ \texttt{zip} & : & \forall x.[x]\alpha \rightarrow [x]\beta \rightarrow [x](\alpha, \beta) \\
\textbf{val}\ \texttt{reduce} & : & \forall x.(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [x]\alpha \rightarrow \alpha \\
\textbf{val}\ \texttt{scan} & : & \forall x.(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [x]\alpha \rightarrow [x]\alpha \\
\textbf{val}\ \texttt{filter} & : & \forall x.(\alpha \rightarrow \texttt{bool}) \rightarrow [x]\alpha \rightarrow \exists y.[y]\alpha
\end{array}
$$

Notice, however, that F is not sufficiently expressive to actually *define* all of these functions, as `reduce`, `scan`, and `filter` require recursion or some other flexible construct for expressing recurrences.

It is critical that parametric polymorphism does not allow us to construct irregular arrays. To see how the type rules prevent irregular arrays, consider the expression

$$\texttt{map}\ (\lambda x.\texttt{iota}\ x)\ y$$

which is ill-typed because

$$\Gamma \vdash (\lambda x.\texttt{iota}\ x) : (x : int) \rightarrow [x]\texttt{int}$$

and we cannot instantiate β in the type scheme for `map` with $[x]\texttt{int}$ because $\Gamma \vdash [x]\texttt{int}\ \textbf{ok}$ does not hold ($x \notin \mathrm{Dom}(\Gamma)$). Similarly, the expression

$$\texttt{map}\ (\lambda x.\texttt{filter}\ f\ x)\ y$$

is ill-typed because T-INST only allows instantiation of type variables with basic types (τ).

We postpone the development of a soundness result for the language featuring size- and type-polymorphism. However, for the Hindley-Damas-Milner style `let`-polymorphism we support, extending the proof to cover these features, we believe can be built on standard techniques [18].

5 Size Types in Futhark

Apart from the formal treatment of F, we have also added size types to the Futhark programming language. The core design is the same—in particular, sizes must still be constants or variable names—but the system has been extended and augmented to handle the requirements of a full-featured programming language. In this section we will discuss the significant extensions that we found necessary, as well as reflect on their usability for writing real functional array programs. Our exposition of size types in Futhark is still informal—a full formalisation remains future work. In Futhark, sizes are of type `i64`, for 64-bit integers. Generally, the addition of size types did not complicate the compiler's many front-end transformations, such as defunctorisation [8], monomorphisation, lambda lifting, and defunctionalisation [14].

Nontrivial size expressions. The soundness proof in Section 3 assumes that all subexpressions are variables or integer constants. This assumption is awkward when writing real programs, and so the Futhark compiler morally rewrites expressions of the form

```
iota (f x)
```

to

```
let v = f x in iota v
```

for some compiler-generated name v. Further, it also uses the explicit version of the `let` construct to assign internal names to existential sizes. To keep type errors comprehensible, the compiler remembers the original expression for v, so that whenever v occurs as a size in a type error, the compiler will be able to point to the originating expression. Similarly, when existential sizes arise due to names going out of scope, or application of a function that returns an existential size, the compiler tracks the origin of the existential size, so it can explain where it came from. This requires some extra book-keeping in the type checker, but has not complicated later stages or optimisations in the compiler, as merely maintaining the program in Administrative Normal Form [21], provides these properties.

Types $\boxed{\Gamma \vdash \tau\ \mathbf{ok}}$

$$\frac{}{\Gamma \vdash \alpha\ \mathbf{ok}}$$

Expressions $\boxed{\Gamma \vdash e : \mu}$

$$\frac{\Gamma(f) = \forall\bar{\alpha}.\forall\bar{x}.\tau \quad \Gamma \vdash \tau_1\ \mathbf{ok} \quad \cdots \quad \Gamma \vdash \tau_n\ \mathbf{ok} \quad \Gamma \vdash y_1 : \mathtt{int} \quad \cdots \quad \Gamma \vdash y_n : \mathtt{int}}{\Gamma \vdash f : \tau\{\bar{\tau}/\bar{\alpha}, \bar{y}/\bar{x}\}}\ [\text{T-INST}]$$

$$\frac{\{\bar{\alpha}\} \cap \mathrm{ftv}(\Gamma, \tau_2) = \emptyset \quad \{\bar{x}\} \cap \mathrm{fv}(\tau_2) = \emptyset \quad \Gamma, \bar{x} : \mathtt{int} \vdash e_1 : \tau_1 \quad \Gamma, f : \forall\bar{\alpha}.\forall\bar{x}.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}\ f\ [\bar{x}]\ \bar{\alpha} = e_1\ \mathtt{in}\ e_2 : \tau_2}\ [\text{T-GEN}]$$

Figure 9. Type rules for polymorphism, extending those of Figures 2 and 3.

Type inference. While F is explicitly typed, the Futhark type checker is built around a conventional Hindley-Damas-Milner type inference implementation. We have extended this implementation to also support *size inference*. When instantiating a type scheme, each size parameter is turned into a nonrigid *size variable*, which is then unified with other size variables and term-level variables. This allows inference of a (contrived) function such as

```
let f n xs = zip (iota n) xs
```

which is assigned the type

```
val f 't : (n:i64) → (xs:[n]t) → [n](i64,t)
```

Similarly to type variables, unconstrained size variables are turned into size parameters during `let`-generalization. Our type inference algorithm is currently merely a *best-effort* implementation, and will sometimes need type annotations in the source program to assign sizes that would otherwise be ambiguous.

Implicit size parameters. Futhark supports *size parameters*, as in Section 4, which are also the main mechanism we use to express shape constraints in function types. For example::

```
val dotprod [n] :
  (xs:[n]i64) → (ys:[n]i64) → i64
```

During compilation, these implicit parameters are turned into ordinary explicit parameters. For each application `dotprod xs ys`, the compiler will then use the types of `xs` and `ys` to decide what to pass for the implicit arguments.

Extensions to parametric polymorphism. There are some cases where we wish to allow a functional argument to a higher-order function to have an existential size in its result type

```
let apply 'a 'b (f: a → b) (x: a) : b = f x
```

But as discussed in Section 4, `apply (filter p) xs` is not well typed. To address this, we introduce the notion of *size-lifted type parameters*, written `'~a`, which may be instantiated (in negative position) with return types with existential sizes:

```
let apply 'a '~b (f: a → b) (x: a) : b = f x
```

To avoid irregular arrays, size-lifted types may never be array elements, which is enforced by the type checker.

6 Experience Report

When designing a type system that limits flexibility to obtain convenience, we must evaluate whether the loss of flexibility is an obstacle in practice, which is necessarily a subjective question. Our prototype implementation of size types was added to Futhark in version 0.15.1, released in March 2020. The Futhark benchmark suite [4] is the largest single collection of Futhark programs, and serves as a case study for the usability of the type system. The suite comprises about 12,000 lines of code, spread over ports of benchmarks from Rodinia [4] (13 programs), FinPar [1] (3 programs), Parboil [24] (5 programs), the Accelerate [3] examples (13 programs), and an ad-hoc collection of further benchmarks (10 programs). Most of these programs were written before size types were added to Futhark and had to be modified.

The benchmark suite contains 66 instances of dynamically checked size coercions, including calls to library functions that include such size coercions. Most of these are in initialisation code, where inputs must be packed in some benchmark-specific format. For example, *backprop* from Rodinia accepts and returns a variety of arrays of sizes n and $n + 1$, where the program then splits the latter into arrays of size 1 and n respectively. Since this relationship cannot be directly expressed in our size type system, we are forced to use type coercions. The main computational core of *backprop* does not contain any type coercions. While dynamic checks are never desirable, they are relatively innocent in

[4] https://github.com/diku-dk/futhark-benchmarks/

initialisation code, where they function as a form of input validation.

In some cases we had to change our programming style. For example, a relatively commonly occurring pattern was

```
map (λi → map (λj → ...) (iota (f x)))
    (iota (g y))
```

Since `f x` is not a constant or variable, the innermost `iota` (and hence the entire inner `map`) returns an array of existential size, which makes the outermost `map` type-incorrect, as its function must not return an array of existential size. While `f x` is invariant to the `map`, we did not want the type checker to make such subtle transformations on behalf of the programmer. The immediate solution is to hoist out the size as an outer binding:

```
let n = f x
in map (λi → map (λj → ...) (iota n))
       (iota (g y))
```

This solution is arguably ugly. But if we define a function

```
val tabulate_2d 'a : (n: i64) → (m: i64)
  → (f: i64 → i64 → a) → [n][m]a
```

then we can write the original expression as:

```
tabulate_2d (g y) (f x) (λi j → ...)
```

Similarly, a common pattern was to write

```
zip xs (iota (length xs))
```

to associate each element of an array with its index. This is invalid under size typing, since as far as the type checker is concerned, `length` just returns an arbitrary integer without any relation to the size of `xs`. A solution is to define another helper function

```
val indices [n] 'a : [n]a → [n]i64
```

such that we can write `zip xs (indices xs)`. Both of these functions are ordinary library code, not compiler builtins, and do not contain any size coercions. In this way, we were usually able to both satisfy the type checker and write concise code, although we did have to change old habits.

We also used size-typed Futhark to teach parallel programming to students at the University of Copenhagen. Most had no experience with advanced type systems, and many had little experience with functional programming at all, yet they were able to write nontrivial Futhark programs without tripping over type restrictions too often. We attribute this to the simplicity of the rules, as well as the ease with which they can be broken when necessary. For programmers who are not experts in type theory, it is useful to have a rigid type system that covers only the simple cases and contains escape hatches for the rest, rather than a complex system that can handle all cases.

7 Related Work

Size inference. A theory of shapes was developed by Jay over a series of papers [15, 16]. Jay's specific work focused on "shapely" programs, where the shapes could be determined in advance (possibly by using partial evaluation to construct an inspector [17]), while our size types also support statically undeterminable existential sizes. The Futhark compiler previously used a size inference technique based on *slicing* [12] to e.g. compute in advance the return size of the function passed to `map`, but this has been completely replaced by the size type system, where the type checker provides all the information needed by the compiler.

Array languages. In the APL tradition, k-dimensional arrays are represented as a pair of a k-vector of sizes and another vector of values. This permits a trivial implementation of operations such as reshaping by changing the shape vector, and of *shape polymorphic* functions that operate solely on the value vector. This idea is given a deeper theoretical treatment in the *mathematics of arrays* [20], and also used in typed functional languages such as SAC [22] and TAIL [7, 11]. In contrast, our type system views multidimensional arrays inductively as as "arrays of arrays", and while it supports a form of *size* polymorphism, it does not support *rank* polymorphism. The advantage is that this is a natural fit with conventional ML-style parametric polymorphism, where an array type $[n]\alpha$ can have the element type α substituted with any type, including another array type. This also makes it straightforward to, for example, `map` a function across the rows of a two-dimensional array.

Type systems. Our type system is heavily inspired by dependent types, where tracking the sizes of vectors is one of the classic examples. For example, Dependent ML uses singleton types to eliminate bounds checking in programs [27]. While we do not address bounds checking in our type system and fall back to run-time checks, we assume that most programs will be written "index-free" using combinators such as `map` and `reduce`. The main limitation of our type system versus full-spectrum dependent types is that we only allow one specific term-level type (integers) to appear in types, only for array shapes, and only in a very restricted syntactic form (variables and constants). A notable implementation advantage of this simplification is that it permits straightforward type erasure, as arrays nevertheless have to carry around their shape at run-time, which is usually a negligible amount of data compared to the array elements.

Qube [26] is an interesting combination of APL-style shapes and dependent types, including support for shape polymorphic programming. Qube verifies all size constraints and index operations at compile-time, using an SMT solver to handle complex constraints. However, Qube still requires the array element type to be monomorphic, which in particular means that it does not support ML-style parametric polymorphism. In contrast to both Futhark and F, Qube supports irregular arrays via dependent products—we believe the same approach could also be used with our size types.

Remora [23] is an array language that is similarly dependently typed, and focuses on capturing APL-style implicit aggregate operations in the type system. In particular, Remora supports arrays of functions, allowing a programming style that encodes control flow as data. It has been shown that parts of this type discipline can be encoded using the fragment of dependent types available in Haskell [9]

Finally, Dex [19] is an array language developed at Google Research. In contrast to most such languages, it embraces explicit indexing, making it safe by making assigning distinct types (and types of index values) to each distinct dimension of an array. Dex uses conventional existential types to handle operations such as `filter`.

8 Conclusions and Future Work

In this paper we have presented a type system for array programming that is fundamentally similar to existing ML-style type systems, yet allows the checking of array size invariants in function applications. We formalised the core of the type system and explained informally how we have added it to the Futhark language. The type system has the significant restriction that only variables or constants can be used in array sizes. Despite this limitation, it is flexible enough to handle the programs found in the Futhark benchmark suite, most of which are ported from other languages, while needing few dynamic checks. This suggests that the type system is a good fit for regular nested data parallel algorithms.

However, there is more work to do. It is awkward that the type system is not strong enough to express the shapes that arise from catenating or flattening arrays. It is likely that such extensions will require the compiler to solve integer problems at compile-time, as in Dependent ML or Qube.

Finally, our formal treatment of size types is still incomplete. It lacks important features such parametric polymorphism, does not describe a technique for type inference, and contains some simplifying assumptions—most critically, that arrays are never empty.

Acknowledgments

This research has been partially supported by a grant from the Independent Research Fund Denmark, under the research project *FUTHARK: Functional Technology for High-performance Architectures*.

A Proofs of Propositions 3.3 and 3.6

Proposition 3.3 (Logical Relation Extensibility)

1. If $\delta \models v : \tau$ and $\delta' \approx_{\mathrm{fv}(\tau)} \delta$ then $\delta' \models v : \tau$.
2. If $\delta \models v : \mu$ and $\delta' \approx_{\mathrm{fv}(\mu)} \delta$ then $\delta' \models v : \mu$.
3. If $\delta \models v : \sigma$ and $\delta' \approx_{\mathrm{fv}(\sigma)} \delta$ then $\delta' \models v : \sigma$.
4. If $\delta \models \rho : \Gamma$ and $\delta' \approx_{\mathrm{fsv}(\Gamma)} \delta$ then $\delta' \models \rho : \Gamma$.

Proof. For proving property 1, there is one interesting case. The remaining cases either follow immediately or can be shown by straightforward induction.

<u>CASE $\tau = (x : \tau') \to \mu$:</u> From assumption and L-CLOS, we have $\lceil 1 \rceil$ $v = \langle x, e', \rho \rangle$ and $\lceil 2 \rceil$ $\forall v_1, v_2$, ($\delta \models v_1 : \tau'$ and $\rho, x : v_1 \vdash e' \rightsquigarrow v_2$) $\implies$ (if $\tau' = \mathtt{int}$ then $\delta, x : v_1 \models v_2 : \mu$ else $\delta \models v_2 : \mu$).

To establish $\delta' \models v : \tau$, we must establish $\lceil 3 \rceil$ $\forall v_1, v_2$, ($\delta' \models v_1 : \tau'$ and $\rho, x : v_1 \vdash e' \rightsquigarrow v_2$) $\implies$ (if $\tau' = \mathtt{int}$ then $\delta', x : v_1 \models v_2 : \mu$ else $\delta' \models v_2 : \mu$).

We first assume $\lceil 4 \rceil$ $\delta' \models v_1 : \tau'$ and $\lceil 5 \rceil$ $\rho, x : v_1 \vdash e' \rightsquigarrow v_2$. From assumption, we have $\lceil 6 \rceil$ $\delta' \approx_{\mathrm{fv}(\tau)} \delta$. From the definition of agreement and because $\mathrm{fv}(\tau') \subseteq \mathrm{fv}(\tau)$, we have $\lceil 7 \rceil$ $\delta' \approx_{\mathrm{fv}(\tau')} \delta$. By induction on $\lceil 4 \rceil$ and $\lceil 7 \rceil$, we have $\lceil 8 \rceil$ $\delta \models v_1 : \tau'$. We can now apply $\lceil 2 \rceil$ to $\lceil 8 \rceil$ and $\lceil 5 \rceil$ to get $\lceil 9 \rceil$ (if $\tau' = \mathtt{int}$ then $\delta, x : v_1 \models v_2 : \mu$ else $\delta \models v_2 : \mu$).

There are now two cases.

SUBCASE $\tau' = \mathtt{int}$: From the definition of agreement and from $\lceil 6 \rceil$, we have $\lceil 10 \rceil$ $(\delta', x : v_1) \approx_{\mathrm{fv}(\mu)} (\delta, x : v_1)$. By induction on $\lceil 9 \rceil$ (case $\tau' = \mathtt{int}$) and $\lceil 10 \rceil$, we get $\delta', x : v_1 \models v_2 : \mu$, as required.

SUBCASE $\tau \neq \mathtt{int}$: In this case $x \notin \mathrm{fv}(\mu)$, thus, from $\lceil 6 \rceil$ and the definition of agreement, we have $\lceil 11 \rceil$ $\delta' \approx_{\mathrm{fv}(\mu)} \delta$. It now follows directly using induction on $\lceil 9 \rceil$ (case $\tau' \neq \mathtt{int}$) and $\lceil 11 \rceil$ that $\delta' \models v_2 : \mu$, as required.

For proving property 2, there is one case to consider:

<u>CASE $\mu = \exists x.\mu'$:</u> By renaming, we can assume $x \notin \mathrm{Dom}(\delta')$. From L-RT, we have there exists n such that $\delta, x : n \models v : \mu'$. From the definition of agreement, we have $(\delta', x : n) \approx_{\mathrm{fv}(\mu') \cup \{x\}} (\delta, x : n)$ and, thus, $(\delta', x : n) \approx_{\mathrm{fv}(\mu')} (\delta, x : n)$. We can now apply induction to get $\delta', x : n \models v : \mu'$. From L-RT, we now have $\delta' \models v : \mu$, as required.

Property 3 follows immediately using property 2.

For proving property 4, there is again only one case to consider. We are given $\delta \models \rho : \Gamma$ and we need to establish $\delta' \models \rho : \Gamma$. The second property in the expanded definition of $\models$ on environments (i.e., L-ENV) are established directly using property 3 above and by the definition of $\mathrm{fsv}(\Gamma)$. The first property is independent of δ' and thus follows immediately. The third and fourth properties follow immediately from the definition of agreement and $\mathrm{fsv}(\Gamma)$. □

Proposition 3.6 (Soundness) *If* $\Gamma \vdash e : \mu$ *and* $\delta \models \rho : \Gamma$ *and* $\rho \vdash e \rightsquigarrow v$ *then* $\delta \models v : \mu$.

Proof. By induction on the structure of the typing derivation. We proceed by case analysis and show most of the cases, leaving out the case for conditions (which follow by straightforward induction), the cases for booleans (which follow the case for integers), and some cases that can be treated similarly to other cases.

<u>CASE $e = n$, $\mu = \mathtt{int}$:</u> Follows immediately from rule T-INT, rule D-INT, and L-INT.

<u>CASE $e = x, \mu = \tau$</u>: From assumptions, rule T-VAR, and rule D-VAR, we have $\Gamma(x) = \tau$ and $\rho(x) = v$ and $\delta \models \rho : \Gamma$. From L-ENV, we have $\delta \models \rho(x) : \Gamma(x)$, for all $x \in \mathrm{Dom}(\Gamma)$. Thus, we have $\delta \models v : \mu$, as required.

<u>CASE $e = (e_1, e_2), \mu = \exists\bar{x}_1\bar{x}_2.(\tau_1, \tau_2)$</u>: From rule T-PAIR, we have ⌈1⌉ $\Gamma \vdash e_1 : \exists\bar{x}_1.\tau_1$ and ⌈2⌉ $\Gamma \vdash e_2 : \exists\bar{x}_2.\tau_2$. From rule D-PAIR, we have ⌈3⌉ $\rho \vdash e_1 \rightsquigarrow v_1$ and ⌈4⌉ $\rho \vdash e_2 \rightsquigarrow v_2$ and $v = (v_1, v_2)$. By induction on ⌈1⌉, ⌈2⌉, ⌈3⌉, and ⌈4⌉, we have $\delta \models v_1 : \exists\bar{x}_1.\tau_1$ and $\delta \models v_2 : \exists\bar{x}_2.\tau_2$. Now, let $\mu_1 = \exists\bar{x}_1\bar{x}_2.\tau_1$ and $\mu_2 = \exists\bar{x}_1\bar{x}_2.\tau_2$. By appropriate renaming, we have $\delta \models v_1 : \mu_1$ and $\delta \models v_2 : \mu_2$. From L-RT, we know there exists $\bar{n}_1$ and $\bar{n}_2$ such that $\delta' = \delta, \bar{x}_1 : \bar{n}_1, \bar{x}_2 : \bar{n}_2$ and $\delta' \models v_1 : \tau_1$ and $\delta' \models v_2 : \tau_2$. From L-PAIR, we now have $\delta' \models (v_1, v_2) : (\tau_1, \tau_2)$. We can now use L-RT again to get $\delta \models v : \mu$, as required.

<u>CASE $e = \texttt{fst}\ e', \mu = \exists\bar{x}.\tau'$</u>: This case is proven by induction, L-PAIR, rule T-FST, rule D-FST, multiple uses of L-RT, and from return types being considered identical up to removal of variables that do not appear in the body.

<u>CASE $e = \texttt{snd}\ e', \mu = \exists\bar{x}.\tau'$</u>: Similar to the case for $e = \texttt{fst}\ e'$.

<u>CASE $e = [e_1, \cdots, e_n], \mu = [n]\tau$</u>: From rule T-ARRAY, we have $\Gamma \vdash e_i : \tau, i \in \{1..n\}$. From rule D-ARRAY, we have $\rho \vdash e_i \rightsquigarrow v_i, i \in \{1..n\}$. By induction, we have $\delta \models v_i : \tau$, $i = \{1..n\}$. From L-ARR-N, we have $\delta \models [v_1, \cdots, v_n] : [n]\tau$, as required.

<u>CASE $e = \texttt{iota}\ d, \mu = [d]\texttt{int}$</u>: From rule T-IOTAD, we have ⌈0⌉ $\Gamma \vdash d : \texttt{int}$. From rule D-IOTA, we have $v = [0, \cdots, n-1]$ and ⌈1⌉ $\rho \vdash d \rightsquigarrow n$. From rule T-INT, we have $\Gamma \vdash v_i : \texttt{int}$, where $v_i = i - 1$, $i = \{1..n\}$. From L-INT, we have ⌈2⌉ $\delta \models v_i : \texttt{int}$. There are two subcases corresponding to whether $d = n$ (for some n) or $d = x$ (for some x).

If $d = n$, for some integer n, we immediately have from L-ARR-N that $\delta \vdash [v_1, \cdots, v_n] : [n]\texttt{int}$, as required.

If $d = x$, from ⌈1⌉ and rule D-VAR, we have ⌈3⌉ $\rho(x) = n$. From ⌈0⌉ and rule T-VAR, we have ⌈4⌉ $\Gamma(x) = \texttt{int}$. From assumption, we have $\delta \models \rho : \Gamma$, thus, from L-ENV and ⌈4⌉, we have $x \in \mathrm{TyDom}(\delta)$ and ⌈5⌉ $\delta(x) = \rho(x)$. From ⌈5⌉ and ⌈3⌉ it thus follows that ⌈6⌉ $\delta(x) = n$. From, L-ARR-X, ⌈2⌉, and ⌈6⌉, we now have $\rho \models [0, \cdots, n-1] : [d]\texttt{int}$, as required.

<u>CASE $e = \lambda(x : \tau').e', \mu = (x : \tau') \to \mu'$</u>: From rule T-LAM, we have ⌈0⌉ $\Gamma, x : \tau' \vdash e' : \mu'$ and $\Gamma : \tau' :$ **ok**. From rule D-LAM, we have $v = \langle x, e', \rho\rangle$. From L-CLOS, we have that to establish $\delta \models v : \mu$, we need to show

> ⌈1⌉ $\forall v_1, v_2, (\delta \models v_1 : \tau'$ and $\rho, x : v_1 \vdash e' \rightsquigarrow v_2) \implies$ (if $\tau' = \texttt{int}$ then $\delta, x : v' \models v_2 : \mu'$ else $\delta \models v_2 : \mu'$)

We first assume ⌈2⌉ $\delta \models v_1 : \tau'$ and ⌈3⌉ $\rho, x : v_1 \vdash e' \rightsquigarrow v_2$. There are now two subcases.

SUBCASE $\tau' = \texttt{int}$: Because $v_1 = n$ for some n, we have from L-INT that ⌈4⌉ $\delta, x : n \models v_1 : \tau'$. From assumption and Property 3.3(4), we have ⌈4a⌉ $\delta, x : n \models \rho : \Gamma$. From ⌈4⌉ and ⌈4$a$⌉ and from L-ENV (and because we can demonstrate $\vdash \Gamma, x : \tau'$ **ok**), we have ⌈5⌉ $(\delta, x : n) \models (\rho, x : n) : (\Gamma, x : \texttt{int})$. Now, by induction on ⌈0⌉, ⌈3⌉, and ⌈5⌉, we get $\delta, x : v_1 \models v_2 : \mu'$, as required.

SUBCASE $\tau' \neq \texttt{int}$: From ⌈2⌉, assumptions, and from L-ENV, we have ⌈7⌉ $\delta \models (\rho, x : v') : (\Gamma, x : \tau')$. By induction on ⌈0⌉, ⌈3⌉, and ⌈7⌉, we have $\delta \models v_2 : \mu'$, as required.

<u>CASE $e = e_1\ e_2, \mu = \exists x.\mu'$</u>: From assumptions and rule T-APP and rule D-APP, we have ⌈0⌉ $\delta \models \rho : \Gamma$, ⌈1⌉ $\Gamma \vdash e_1 : (x : \tau) \to \mu'$, ⌈2⌉ $\Gamma \vdash e_2 : \tau$, ⌈4⌉ $\rho \vdash e_1 \rightsquigarrow \langle x, e', \rho'\rangle$, ⌈5⌉ $\rho', x : v' \vdash e' \rightsquigarrow v$, and ⌈6⌉ $\rho \vdash e_2 \rightsquigarrow v'$.

By induction applied to ⌈1⌉, ⌈0⌉, and ⌈4⌉, we get ⌈7⌉ $\delta \models \langle x, e', \rho'\rangle : (x : \tau) \to \mu'$. From L-CLOS and ⌈7⌉, we have ⌈9⌉ $\forall v_1, v_2, (\delta \models v_1 : \tau$ and $\rho', x : v_1 \vdash e' \rightsquigarrow v_2) \implies$ (if $\tau = \texttt{int}$ then $\delta, x : v_1 \models v_2 : \mu'$ else $\delta \models v_2 : \mu'$).

By induction on ⌈2⌉, ⌈0⌉, and ⌈6⌉, we have ⌈10⌉ $\delta \models v' : \tau$. We can now apply ⌈9⌉ to ⌈10⌉ and ⌈5⌉, choosing $v_1 = v'$ and $v_2 = v$, to get ⌈11⌉ (if $\tau = \texttt{int}$ then $\delta, x : v' \models v : \mu'$ else $\delta \models v : \mu'$).

There are now two subcases.

SUBCASE $\tau = \texttt{int}$: From L-RT and ⌈11⌉, and because $\mu = \exists x.\mu'$, we have $\delta \models v : \mu$, as required.

SUBCASE $\tau \neq \texttt{int}$: In this case, we know that $x \notin \mathrm{fv}(\mu')$, thus, $\mu = \mu'$ and it follows directly from ⌈11⌉ that $\delta \models v : \mu$, as required.

<u>CASE $e = e_1 \triangleright \tau, \mu = \tau$</u>: From rule T-COERCE, we have ⌈1⌉ $\Gamma \vdash e_1 : \tau_1$ and ⌈2⌉ $\Gamma \vdash \tau$ <u>**ok**</u> and ⌈3⌉ $\tau = \tau_1\{\bar{d}/\bar{x}\}$. From rule D-COERCE, we have ⌈4⌉ $\rho \vdash e_1 \rightsquigarrow v$ and $\rho \vdash v \triangleright \tau$. From assumption, we have ⌈6⌉ $\delta \models \rho : \Gamma$. By induction on ⌈6⌉, ⌈1⌉, and ⌈4⌉, we have ⌈7⌉ $\delta \models v : \tau_1$. From ⌈6⌉ and L-ENV, we have ⌈8⌉ $\forall x \in \mathrm{TyDom}(\Gamma), (\Gamma(x) = \texttt{int} \implies \rho(x) = \delta(x))$. From ⌈2⌉ and ⌈8⌉, we have ⌈9⌉ $\forall x \in \mathrm{fv}(\tau)$, $x \in \mathrm{TyDom}(\Gamma)$ and $\Gamma(x) = \texttt{int}$. Thus, we have ⌈10⌉ $\rho(x) = \delta(x), \forall x \in \mathrm{fv}(\tau)$.

From Proposition 3.5, ⌈7⌉, ⌈3⌉, ⌈5⌉, and ⌈10⌉, we have $\delta \models v : \tau$ and, thus, $\delta \models v : \mu$, as required.

<u>CASE $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2, \mu = \exists\bar{x}x.\mu'$</u>: From assumption and rule T-LET, we have ⌈1⌉ $\delta \models \rho : \Gamma$, ⌈2⌉ $\Gamma \vdash e_1 : \exists\bar{x}.\tau$, and $\Gamma, \bar{x} : \bar{\star}, x : \tau \vdash e_2 : \mu'$. From assumption and rule D-LET, we have ⌈4⌉ $\rho \vdash e_1 \rightsquigarrow v_1$ and $\rho, x : v_1 \vdash e_2 \rightsquigarrow v$. By induction on ⌈1⌉, ⌈2⌉, and ⌈4⌉, we get ⌈6⌉ $\delta \models v_1 : \exists\bar{x}.\tau$. From L-RT and ⌈6⌉, we have there exists $\bar{n}$ such that ⌈7⌉ $\delta, \bar{x} : \bar{n} \models v_1 : \tau$. There are now two cases. We first assume $\tau \neq \texttt{int}$. Let $\delta' = \delta, \bar{x} : \bar{n}$ and $\Gamma' = \Gamma, \bar{x} : \bar{\star}, x : \tau$, and $\rho' = \rho, x : v_1$. From ⌈1⌉ and because $\delta' \approx_{\mathrm{fsv}(\Gamma)} \delta$, we have $\delta' \models \rho : \Gamma$. Now, from L-ENV, we have

> ⌈8⌉ $\mathrm{Dom}(\rho) = \mathrm{TyDom}(\Gamma)$ and $\vdash \Gamma$ **ok**
> ⌈9⌉ $\forall x \in \mathrm{TyDom}(\Gamma), \delta' \models \rho(x) : \Gamma(x)$
> ⌈10⌉ $\forall x \in \mathrm{TyDom}(\Gamma), \Gamma(x) = \texttt{int} \implies \delta'(x) = \rho(x)$
> ⌈11⌉ $\mathrm{StarDom}(\Gamma) \subseteq \mathrm{Dom}(\delta')$

From ⌈8⌉, the definitions of Γ' and ρ', and properties about well-formedness of types, we have that ⌈16⌉ $\mathrm{Dom}(\rho') = \mathrm{TyDom}(\Gamma')$ and $\vdash \Gamma'$ **ok**. From ⌈9⌉, ⌈7⌉, and the definitions of δ' and Γ', we have ⌈17⌉ $\forall x \in \mathrm{TyDom}(\Gamma'), \delta' \models \rho'(x) : \Gamma'(x)$. Because $\tau \neq \texttt{int}$ (separate case), we have from ⌈10⌉ that ⌈18⌉ $\forall x \in \mathrm{TyDom}(\Gamma'), (\Gamma'(x) = \texttt{int} \implies \delta'(x) =$

$\rho'(x)$). From ⌈11⌉ and the definitions of Γ' and δ', we have $\mathrm{StarDom}(\Gamma') \subseteq \mathrm{Dom}(\delta')$. From ⌈16⌉, ⌈17⌉, ⌈18⌉, and ⌈19⌉, we have ⌈20⌉ $\delta' \models \rho' : \Gamma'$. By induction on ⌈20⌉, ⌈3⌉, and ⌈5⌉, using the definitions of δ', ρ', and Γ', we have ⌈21⌉ $\delta' \models v : \mu'$. From L-RT and ⌈21⌉, we have ⌈22⌉ $\delta \models v : \exists\bar{x}.\mu'$. From ⌈1⌉, we have $\Gamma' \vdash \mu'$ **ok** and because $\Gamma'(x) \neq \texttt{int}$ and $\Gamma'(x) \neq \star$, we know that $x \notin \mathrm{fv}(\mu')$. Thus, $\exists x.\mu' = \mu'$, which, together with ⌈22⌉ gives us $\delta \models v : \mu$, as required. If, instead, we assume $\tau = \texttt{int}$, we know $\bar{x}$ is empty, but, instead, we have from ⌈7⌉ and L-INT, that $v_1 = n$ for some n. Now, let $\delta' = \delta, x : n$, $\Gamma' = \Gamma, x : \texttt{int}$, and $\rho' = \rho, x : n$. From ⌈1⌉ and from L-ENV, we have

⌈23⌉ $\mathrm{Dom}(\rho) = \mathrm{TyDom}(\Gamma)$ and $\vdash \Gamma$ **ok**
⌈24⌉ $\forall x \in \mathrm{TyDom}(\Gamma), \delta \models \rho(x) : \Gamma(x)$
⌈25⌉ $\forall x \in \mathrm{TyDom}(\Gamma), \Gamma(x) = \texttt{int} \implies \delta(x) = \rho(x)$
⌈26⌉ $\mathrm{StarDom}(\Gamma) \subseteq \mathrm{Dom}(\delta)$

From ⌈23⌉, the definition of well-formed environments, and the definitions of ρ' and Γ', we have that ⌈27⌉ $\mathrm{Dom}(\rho') = \mathrm{TyDom}(\Gamma')$ and $\vdash \Gamma'$ **ok**. From ⌈6⌉ and because $\delta' \approx_{\mathrm{fv}(\Gamma'(x))} \delta$, forall $x \in \mathrm{TyDom}(\Gamma')$, and from the definitions of ρ' and Γ', we have from Proposition 3.3 that ⌈28⌉ $\forall x \in \mathrm{TyDom}(\Gamma')$, $\delta' \models \rho'(x) : \Gamma'(x)$. From ⌈25⌉ and from the definitions of Γ', δ', and ρ', we have ⌈29⌉ $\forall x \in \mathrm{TyDom}(\Gamma'), \Gamma'(x) = \texttt{int} \implies \delta'(x) = \rho'(x)$. From ⌈26⌉ and the definitions of Γ' and δ', we have ⌈30⌉ $\mathrm{StarDom}(\Gamma') \subseteq \mathrm{Dom}(\delta')$. Now, from ⌈27⌉, ⌈28⌉, ⌈29⌉, and ⌈30⌉, we have ⌈31⌉ $\delta' \models \rho' : \Gamma'$. By induction on ⌈31⌉, ⌈3⌉, and ⌈5⌉, using the definitions of δ', ρ', and Γ', we have ⌈32⌉ $\delta' \models v : \mu'$. From L-RT and ⌈32⌉, we have $\delta \models v : \exists x.\mu'$ and because $\mu = \exists x.\mu'$ ($\bar{x}$ is empty), we have $\delta \models v : \mu$, as required.

Case $\texttt{let}\ [\bar{x}]\ x : \tau = e_1\ \texttt{in}\ e_2$, $\mu = \exists\bar{x}x.\mu'$: The proof for this case is similar to the proof for rule T-LET, although size variables are bound to `int` types in type environments. The proof case also makes essential use of Proposition 3.4. □

References

[1] 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

[2] Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. ACM Trans. Program. Lang. Syst. 23, 5 (Sept. 2001), 657–683. https://doi.org/10.1145/504709.504712

[3] Manuel M.T. 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 (Austin, Texas, USA) (DAMP '11). Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

[4] 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

[5] Kevin Donnelly and Hongwei Xi. 2007. A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F. Electronic Notes in Theoretical Computer Science 174, 5 (2007), 109–125. https://doi.org/10.1016/j.entcs.2007.01.021 Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006).

[6] D. Dreyer, A. Ahmed, and L. Birkedal. 2009. Logical Step-Indexed Logical Relations. In 2009 24th Annual IEEE Symposium on Logic In Computer Science. 71–80. https://doi.org/10.1109/LICS.2009.34

[7] 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

[8] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. 2018. Static Interpretation of Higher-Order Modules in Futhark: Functional GPU Programming in the Large. Proc. ACM Program. Lang. 2, ICFP, Article 97 (July 2018), 30 pages. https://doi.org/10.1145/3236792

[9] Jeremy Gibbons. 2016. APLicative Programming with Naperian Functors (Extended Abstract). In Proceedings of the 1st International Workshop on Type-Driven Development (Nara, Japan) (TyDe 2016). Association for Computing Machinery, New York, NY, USA, 13–14. https://doi.org/10.1145/2976022.2976023

[10] Jean Yves Girard. 1971. Interpretation Fonctionnelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur. In Proceedings of the Second Scandinavian Logic Symposium. North-Holland, 63–92.

[11] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. 2016. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In Proceedings of the 5th International Workshop on Functional High-Performance Computing (Nara, Japan) (FHPC 2016). Association for Computing Machinery, New York, NY, USA, 38–43. https://doi.org/10.1145/2975991.2975997

[12] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. 2014. Size Slicing: A Hybrid Approach to Size Inference in Futhark. In Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing (Gothenburg, Sweden) (FHPC '14). ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/2636228.2636238

[13] 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). ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[14] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2018. High-performance defunctionalization in Futhark. In Symposium on Trends in Functional Programming (TFP'18). Springer-Verlag.

[15] C.B. Jay. 1999. Denotational Semantics of Shape:: Past, Present and Future. Electronic Notes in Theoretical Computer Science 20 (1999), 320–333. https://doi.org/10.1016/S1571-0661(04)80081-1 MFPS XV, Mathematical Foundations of Progamming Semantics, Fifteenth Conference.

[16] C. Barry Jay. 1995. A Semantics for Shape. Science of Computer Programming 25 (1995), 25–251.

[17] C. Barry Jay and Milan Sekanina. 1997. Shape Checking of Array Programs. Technical Report. In Computing: the Australasian Theory Seminar, Proceedings.

[18] Xavier Leroy. 1992. Unboxed Objects and Polymorphic Typing. In Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Albuquerque, New Mexico, USA) (POPL '92). Association for Computing Machinery, New York, NY, USA, 177–188. https://doi.org/10.1145/143165.143205

[19] Dougal Maclaurin, Alexey Radul, Matthew J. Johnson, and Dimitrios Vytiniotis. 2019. Dex: array programming with typed indices. In Poster

session at Workshop on Program Transformations for Machine Learning. *Associated with NeurIPS '2019.*

[20] *Lenore Mullin. 1988.* A mathematics of arrays. *School of Computer and Information Science, Syracuse University.*

[21] *Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. In* LISP AND SYMBOLIC COMPUTATION. *288–298.*

[22] *Sven-Bodo Scholz. 1994. Single Assignment C - Functional Programming Using Imperative Style. In* In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages. University of East Anglia.

[23] *Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In* Programming Languages and Systems, *Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–46.*

[24] *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).*

[25] *William W. Tait. 1967. Intensional interpretations of functionals of finite type.* Journal of symbolic logic *32 (1967), 198–212.*

[26] *Kai Trojahner and Clemens Grelck. 2009. Dependently typed array programs don't go wrong.* J. Log. Algebr. Program. *78 (08 2009), 643–664.* *https://doi.org/10.1016/j.jlap.2009.03.002*

[27] *Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In* Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation *(Montreal, Quebec, Canada)* (PLDI '98). *Association for Computing Machinery, New York, NY, USA, 249–257.* *https://doi.org/10.1145/277650.277732*