Shape-Constrained Array Programming with Size-Dependent Types

Lubin Bailly
Département d'Informatique de l'ENS
École normale supérieure - PSL
Paris, France
catvayor@katvayor.net

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 dependent type system for enforcing array-size consistency in an ML-style functional array language. Our goal is to enforce shape-consistency at compile time and allow nontrivial transformations on array shapes, without the complexity such features tend to introduce in dependently typed languages. Sizes can be arbitrary expressions and size equality is purely syntactical, which fits naturally within a scheme that interprets size-polymorphic functions as having implicit arguments. When non-syntactical equalities are needed, we provide dynamic checking. In contrast to other dependently typed languages, we automate the bookkeeping involved in tracking existential sizes, such as when filtering arrays. We formalise a large subset of the presented type system and prove it sound. We also discuss how to adapt the type system for a real implementation, including type inference, within the Futhark programming language.

CCS Concepts: • **Theory of computation → Type structures**.

Keywords: type systems, parallel programming, functional programming

ACM Reference Format:
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 (FHPNC '23), September 4, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3609024.3609412

FHPNC '23, September 4, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0296-9/23/09…$15.00
https://doi.org/10.1145/3609024.3609412

1 Introduction

For programming in general, it is custom for functions to assume a set of preconditions on the function arguments. Type systems are a common mechanism both for specifying such preconditions and for checking that they are not violated at runtime. The complexity of type systems ranges from the very simple that only ensure consistent use of primitive data types, to the highly elaborate and complicated that can be used to fully prove that a program implements a given specification. Programming language designers often try to strike a balance between maximising the amount of safety provided by the type system and making the type system easy to work with. A powerful type system loses its advantage if it is so difficult to use that programmers simply give up and switch to less expressive types or a less safe language.

Shape constraints often arise in array programming, both as pre- and post-conditions of functions. For example, a dot product requires two arrays of the same size, and `concat` has the postcondition of returning an array having a size equal to the sum of the sizes of its two arguments. Most, although not all, of these constraints are equalities on sizes. In most languages, checking of these equalities is done at runtime, but ideally we would like a way to check statically that violations will never occur.

We present in this paper a *size-dependent type system*, which is inspired by, but much simpler than, full dependent type systems such as those found in Idris [2] or Agda [1]. Our goal is to enable what we consider a "natural" style of functional array programming, which should not be much more complicated than the style of programming solicited by conventional ML-family languages. In particular, we consider multidimensional arrays as "arrays of arrays", and we allow the construction of arrays of arbitrary sizes. One of the main challenges is how to handle arrays whose sizes cannot be expressed cleanly—for example, when a function such as `filter` returns an array of unpredictable size, or when a variable used in an expression defining an array size goes out of scope. In standard dependently typed languages such as Idris [2], this is modelled as a *dependent pair*:

```
filter : (elem -> Bool)
      -> Vect len elem
      -> (p : Nat ** Vect p elem)
```

The `filter` function returns a pair of some size `p` and a vector of just that size `p`. The downside is that a value of type `(p : Nat ** Vect p elem)` cannot be passed to a function that expects a `Vect p elem`, as the types differ. The Idris term `length (filter p xs)` is ill-typed. Instead, the pair must first be unpacked, bringing the size and the vector separately into scope. One of the main goals of our type system is to eliminate such bookkeeping.

One typical challenge regarding the efficient implementation of dependently typed languages is *erasure*: how do we avoid computing and storing types at runtime? In Idris 2, type erasure is achieved using Quantitative Type Theory [3], by which programmers can directly reason about which types must be present at runtime. In our work, the only information that can be extracted from types is their dynamic *shape*, constituting one integer per dimension. Essentially, we support a limited form of dependent pairs where the the first element (the size) can always be determined from the second element (the array), and use this restriction to provide a more ergonomic array programming experience.

Our work is based on the foundations presented in [11], which introduces a size-dependent type system where all sizes must be constants or variables, not compound expressions. Beyond lifting this restriction, we also add the notion of *implicit size polymorphism*, such that sizes of array arguments do not have to be passed explicitly by the programmer.

We begin, in Section 2, by defining a small array-language F, featuring size-polymorphism and size-dependent types. The language is sufficiently rich that it can be used to demonstrate the important features of the type system. In Section 3, we present the theoretical aspects of the language, ending up in a soundness proof. In Section 4, we discuss how to loosen some syntactical restrictions imposed to simplify the theory, in order to enable a more natural programming style. Finally, in Section 5, we discuss how the techniques are implemented in a real functional array language, Futhark [12], and our experience with its use. In Section 6, we describe related work and in Section 7, we conclude and outline future work.

2 The Language F

In this section, we define a small higher-order functional array language F with size polymorphism and size-dependent types. Its full grammar is defined in Figure 1. We assume a countable set of variables, ranged over by x, y, z, with subscripts as necessary. The term level of the language is mostly conventional, featuring integer constants, arithmetic operations, lambda-expressions, function applications, conditionals, pairs, projections, and array indexing. The only primitive type is `int`, which can appear in pair types, array types, and function types, using standard syntax. For conditionals, 0 is considered false and other values true. The only way to construct an array is with `iota` n, which has type $[n]$`int` and creates the array $[0, 1, \ldots, n-1]$, and the only way to transform an array is with `map`, which maps a function over all elements of an array. That is, `map` f $[v_0, \ldots, v_{n-1}]$ produces an array $[f\ v_0, \ldots, f\ v_{n-1}]$.

τ	::=		**Basic types**
	|	`int`	integer
	|	(τ, τ)	pair
	|	$[e]\tau$	array
	|	$(x : \tau) \to \mu$	function
μ	::=		**Return types**
	|	$\exists x.\mu$	existential size
	|	τ	basic type
σ	::=		**Type Schemes**
	|	$\forall x.\sigma$	size polymorphic
	|	τ	basic type
e	::=		**Expressions**
	|	n	constant integer
	|	x	variable
	|	$x^{\overline{e}}$	polymorphic instance
	|	$\lambda(x : \tau).e$	function
	|	$e\ e$	application
	|	$e[e]$	array index
	|	(e, e)	pair
	|	`if` e `then` e `else` e	conditional
	|	`let` $[\overline{x}]\ x : \tau = e$ `in` e	let-bind
	|	`let` $x\ [\overline{x}] : \tau = e$ `in` e	let-gen
	|	$e \triangleright \tau$	type coercion
	|	`fst` e | `snd` e	projection
	|	`iota` e	index array
	|	`map` $e\ e$	map
	|	$e \diamond e$	basic arithmetic
$\diamond$	::=	$+ \mid - \mid \cdot \mid / \mid \leq$	**Infix Operator**
v	::=		**Values**
	|	n	integer
	|	$\langle x, e, \rho \rangle$	closure
	|	$[v, v, \cdots, v]$	array
	|	(v, v)	pair

Figure 1. Grammar of the language.

The language allows conventional `let`-bindings, but with a twist: we can bind otherwise unknown array sizes:

$$\texttt{let}\ [n]\ (y : [n]\texttt{int}) = e_1\ \texttt{in}\ e_2$$

The above expression binds y to the result of e_1 and n to the size of y. Both y and y are in scope in e_2, the latter with type `int`. This can obscure the original size of the array produced by e_1. As we will see in the next section, these *size bindings*

are not just a convenience: the type rules require that some expressions and their sizes are immediately bound to names.

Finally, the language supports the notion of *size coercions* of the form $e \triangleright \tau$, which may be used to check dynamically that the sizes specified in τ match the sizes of the corresponding arrays in e. Whereas τ is guaranteed by the type system to be structural equivalent (to be defined in the next section) to the type of e, in case of a size mismatch, evaluation terminates with an error.

2.1 Types

We use several syntactically distinguished notions of types. A *basic type* τ is an `int`, pair, array (including size), or function type (with a named parameter). A *type scheme* σ is a basic type parameterized by sizes names, and represents a size-polymorphic definition, similarly to type-polymorphic type schemes in conventional polymorphic type systems.

The type to the right of a function arrow is a *return type* μ, which can contain an *existential quantification* of a size. Such quantifications are used to model array sizes that cannot be known statically, and resemble conventional dependent pairs, although we will see that their actual behaviour is somewhat different. The type of expressions is also given by a *return type* μ.

For constructs of the forms $(x : \tau) \to \mu$, $\exists x.\mu$, $\forall x.\sigma$, and $\lambda(x : \tau).e$, x is bound in μ, σ, and e. Moreover, in constructs of the form $\texttt{let}\ [\overline{x}]\ x : \tau = e\ \texttt{in}\ e'$, which is used for unpacking return types, x is bound in e' and $\overline{x}$ are bound in τ and e'. Finally, for expressions of the form $\texttt{let}\ x\ [\overline{z}] : \tau = e\ \texttt{in}\ e'$, which is used for size-polymorphic bindings, x is bound in e' and $\overline{z}$ are bound in τ and e. We consider all constructs identical up to renaming of bound names (alpha-renaming) and write $\mathrm{fv}(o)$ for the free variables of some object o (e.g., a term or a type).

The notion of substituting a term for a variable is central in the formal development that follows. Whenever o is some object, e is some expression, and x is some variable, we write $o\{e/x\}$ to denote the capture-avoiding substitution of e for x in o, assuming that the result is a well-formed entity. The requirement that the result is well-formed rules out exotic substitutions such as $(x^{\overline{e}})\{4/x\}$, which would result in an ill-formed expression. Such substitutions never arise in practice, however, but the restriction is necessary for proving formal properties about the language.

The type $[e]\tau$ represents an array of e elements of type τ, where e must be a well-typed expression of type `int`. Notice that for function types of the form $(x : \tau) \to \mu$, the variable x may appear in μ, as is custom for dependent type systems. We will sometimes write $\tau \to \mu$ for $(x : \tau) \to \mu$ where $x \notin \mathrm{fv}(\mu)$. If this property is important, it will be explicitly written.

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

$$\frac{\Gamma \vdash e : \texttt{int} \quad \Gamma \vdash \tau\ \mathbf{ok}}{\Gamma \vdash [e]\tau\ \mathbf{ok}} \qquad \frac{\Gamma \vdash \tau\ \mathbf{ok} \quad \Gamma \vdash \tau'\ \mathbf{ok}}{\Gamma \vdash (\tau, \tau')\ \mathbf{ok}}$$

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

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

$$\frac{\Gamma \vdash \tau\ \mathbf{ok} \quad \mu \equiv \tau}{\Gamma \vdash \mu\ \mathbf{ok}} \qquad \frac{\Gamma, x : \texttt{int} \vdash \mu\ \mathbf{ok}}{\Gamma \vdash \exists x.\mu\ \mathbf{ok}}$$

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

$$\frac{\Gamma \vdash \tau\ \mathbf{ok} \quad \sigma \equiv \tau}{\Gamma \vdash \sigma\ \mathbf{ok}} \qquad \frac{\Gamma, x : \texttt{int} \vdash \sigma\ \mathbf{ok} \quad x \in \mathrm{fv}(\sigma)}{\Gamma \vdash \forall x.\sigma\ \mathbf{ok}}$$

Figure 2. Well-formedness of types.

Type equality modulo sizes, termed *structural equivalence*, is noted as $\sim_s$ and defined as the equivalence relation satisfying the following equations:

$$\begin{aligned}
[e]\tau \sim_s [e']\tau' &\iff \tau \sim_s \tau' \\
\tau_1 \sim_s \tau_1' \wedge \tau_2 \sim_s \tau_2' &\iff (\tau_1, \tau_2) \sim_s (\tau_1', \tau_2') \\
\tau \sim_s \tau' \wedge \mu \sim_s \mu' &\iff (x : \tau) \to \mu \sim_s (x : \tau') \to \mu' \\
\exists x.\mu \sim_s \mu' &\iff \mu \sim_s \mu' \\
\mu \sim_s \exists x.\mu' &\iff \mu \sim_s \mu'
\end{aligned}$$

Structural equivalence is used for size coercions, where sizes may change but the overall type may not.

We define the set of *witnesses* of a return type, written $\mathrm{wit}(\mu)$, as the set of free variables in the type that are used directly as an array size. Having a value of the corresponding type, we can directly extract the value of a witnessed variable by observing the value. This notion is used to rule out terms whose evaluation would, for example, need to extract a size x from an array of type $[f\ x]\tau$ for some f, as such an extraction would require the possibility of computing the inverse of f, which is not feasible (or even possible) in general.

$$\begin{aligned}
\mathrm{wit}(\exists \overline{x}.\tau) &= \mathrm{wit}(\tau) \setminus \{\overline{x}\} \\
\mathrm{wit}(\texttt{int}) &= \emptyset \\
\mathrm{wit}((\tau_1, \tau_2)) &= \mathrm{wit}(\tau_1) \cup \mathrm{wit}(\tau_2) \\
\mathrm{wit}((z : \tau) \to \mu) &= \emptyset \\
\mathrm{wit}([x]\tau) &= \{x\} \cup \mathrm{wit}(\tau) \\
\mathrm{wit}([e]\tau) &= \mathrm{wit}(\tau)
\end{aligned}$$

Contexts, ranged over by Γ, map variables to type schemes, each of which is either a basic type or a size-polymorphic type. Type assumptions are written $x : \tau$ and extending Γ with such an assumption, written $\Gamma, x : \tau$, assumes that $x \notin \mathrm{fv}(\Gamma)$. Also, we often write $\overline{x}$ for $x_1, \cdots, x_n$. Thus, we sometimes write $\exists \overline{x}.\tau$ instead of $\exists x_1. \cdots \exists x_n.\tau$ (similarly for type schemes) or $\overline{x} : \overline{\tau}$ instead of $x_1 : \tau_1, \cdots, x_n : \tau_n$, and so on.

Return Types $\boxed{\Gamma \vdash \mu \sqsubseteq \mu'}$

$$\frac{\Gamma \vdash e : \texttt{int} \quad \Gamma \vdash \mu' \sqsubseteq \mu\{e/x\} \quad \Gamma \vdash \exists x.\mu \ \textbf{ok}}{\Gamma \vdash \mu' \sqsubseteq \exists x.\mu}$$

$$\frac{\Gamma, x : \texttt{int} \vdash \mu \sqsubseteq \mu'}{\Gamma \vdash \exists x.\mu \sqsubseteq \exists x.\mu'} \qquad \frac{\Gamma \vdash \tau \sqsubseteq \tau' \quad \tau \equiv \mu \quad \tau' \equiv \mu'}{\Gamma \vdash \mu \sqsubseteq \mu'}$$

Basic Types $\boxed{\Gamma \vdash \tau \sqsubseteq \tau'}$

$$\frac{}{\Gamma \vdash \texttt{int} \sqsubseteq \texttt{int}} \qquad \frac{\Gamma \vdash \tau' \sqsubseteq \tau \quad \Gamma, x : \tau \vdash \mu \sqsubseteq \mu'}{\Gamma \vdash (x : \tau) \to \mu \sqsubseteq (x : \tau') \to \mu'}$$

$$\frac{\Gamma \vdash e : \texttt{int} \quad \Gamma \vdash \tau \sqsubseteq \tau'}{\Gamma \vdash [e]\tau \sqsubseteq [e]\tau'} \qquad \frac{\Gamma \vdash \tau_1 \sqsubseteq \tau_1' \quad \Gamma \vdash \tau_2 \sqsubseteq \tau_2'}{\Gamma \vdash (\tau_1, \tau_2) \sqsubseteq (\tau_1', \tau_2')}$$

Figure 3. Subtyping rules.

Types τ (or type schemes σ, or return types μ) are *well-formed* under assumption Γ, written $\Gamma \vdash \tau$ **ok** if such a judgment can be derived from the rules in Figure 2. Existential sizes in return types must be witnessed by the underlying type, which ensures that the values of these sizes can be determined dynamically. Similarly, size parameters in a type scheme must be used in the underlying type. A context Γ' is well-formed under assumption Γ, written $\Gamma \vdash \Gamma'$ **ok**, when $\forall x \in \mathrm{Dom}(\Gamma'), \Gamma \vdash \Gamma'(x)$ **ok**. Finally, $\vdash \Gamma$ **ok** means $\Gamma \vdash \Gamma$ **ok**.

We now define a subtyping relation on return types μ and μ' conditioned by a context Γ, written $\Gamma \vdash \mu \sqsubseteq \mu'$. Judgments of this form are read "μ *is a subtype of* μ' *in the context* Γ" and are derived based on the rules in Figure 3.

The purpose of the subtyping relation is to allow for arbitrary expressions of type `int` that appear inside a type to be replaced with a fresh existentially bound size variable or a fresh size variable bound in a return type. Notice the contravariance in the subtyping rule for function types.

2.2 Type Rules

Type rules for expressions are shown in Figure 4 and produce sentences $\Gamma \vdash e : \mu$, which are read "e *has a result of type* μ *under the context* Γ". In the rule T-APP, the argument name is substituted by the argument expression in the result, much like dependent types, but this is forbidden in T-MAP to guarantee regular arrays. The T-LET rule requires that the name is not free in the type of the body. When needed, T-RELAX can always be used to replace any expression in a size with an unknown size. The freedom to apply T-RELAX whenever we wish means that a program can have many different typing derivations, although these will all be operationally equivalent. T-RELAX is a subtyping rule which is pretty restrictive as we always have structural equivalence between the types before and after applying this rule. This ensures the only information thrown away is size information.

2.3 Dynamic Semantics

We provide semantics only for successful executions-dynamic errors due to size coercions or array indexing are excluded for simplicity of the metatheory, meaning that the type safety property we prove in Section 3 only relates to programs for which a successful evaluation derivation exists. Similarly, we allow only non-empty arrays, which is enforced by the value grammar. Any attempt to construct an array with less than one element fails dynamically. We briefly return to this issue in Section 5.

A dynamic environment ρ maps variables to values. We provide a big-step semantics with the judgment $\rho \vdash e \rightsquigarrow v$, which reads "$e$ *is evaluated to* v *in the environment* ρ". The rules defining this judgment are shown in Figure 7.

The dynamic semantics makes use of two auxiliary judgments: dynamic size matching and dynamic size checking.

Dynamic size matching, written $\tau \vdash_x v \rightsquigarrow n/\bullet$ is defined in Figure 5 and allows the dynamic semantics to extract sizes from values. The result is either $\bullet$, meaning that the extraction failed because x is not witnessed in τ, or an integer denoting the searched size.

Dynamic size checking, written $\rho \vdash v \triangleright \tau$, is defined in Figure 6 and checks that arrays in a value are of the right sizes.

We define a *value equivalence* relation, noted $\sim_{\mathrm{v}}$, which specifies if two values can be considered "interchangeable" in a specific sense. We formally define this notion as the equivalence relation satifying the following rules:

- $n \sim_{\mathrm{v}} n$
- $\langle x, e, \rho \rangle \sim_{\mathrm{v}} \langle x, e', \rho' \rangle \iff$
  $(\rho, x : v \vdash e \rightsquigarrow v' \implies \rho', x : v \vdash e' \rightsquigarrow v'' \wedge v' \sim_{\mathrm{v}} v'') \wedge$
  $(\rho', x : v \vdash e' \rightsquigarrow v' \implies \rho, x : v \vdash e \rightsquigarrow v'' \wedge v' \sim_{\mathrm{v}} v'')$
- $[v_1, \cdots, v_n] \sim_{\mathrm{v}} [v_1', \cdots, v_n'] \iff \forall i, v_i \sim_{\mathrm{v}} v_i'$
- $(v_1, v_2) \sim_{\mathrm{v}} (v_1', v_2') \iff v_1 \sim_{\mathrm{v}} v_1' \wedge v_2 \sim_{\mathrm{v}} v_2'$

The only interesting case is the case for closures. We consider two closures equivalent if they produce equivalent values for the same arguments, but also if they fail on the same arguments.

3 Metatheory for F

To establish a soundness property, we first introduce a logical relation stating that a value is of a certain type. Because types can contain expressions, this relation depends on the dynamic environment. The relation is written $\rho \models v : \mu$, reads "*the value* v *is of return type* μ *in* ρ", and is defined by the following rules:

L-INT $\rho \models n : \texttt{int}$

L-PAIR $\rho \models (v_1, v_2) : (\tau_1, \tau_2)$ iff $\rho \models v_1 : \tau_1$ and $\rho \models v_2 : \tau_2$

L-ARR $\rho \models [v_1, \cdots, v_n] : [e]\tau$ iff $\rho \vdash e \rightsquigarrow n$ and $\rho \models v_i : \tau, \forall i \in \{1..n\}$

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

$$\frac{}{\Gamma \vdash n : \mathtt{int}}\ [\textsc{t-int}] \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ [\textsc{t-var}] \qquad \frac{\Gamma(x) = \forall \overline{z}.\tau \quad \Gamma \vdash \overline{e} : \overline{\mathtt{int}}}{\Gamma \vdash x^{\overline{e}} : \tau\{\overline{e}/\overline{z}\}}\ [\textsc{t-inst}]$$

$$\frac{\Gamma \vdash e_l : \mathtt{int} \quad \Gamma \vdash e_r : \mathtt{int}}{\Gamma \vdash e_l \diamond e_r : \mathtt{int}}\ [\textsc{t-arith}] \qquad \frac{\Gamma \vdash e : \mu \quad \Gamma \vdash \mu \sqsubseteq \mu'}{\Gamma \vdash e : \mu'}\ [\textsc{t-relax}] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)}\ [\textsc{t-pair}]$$

$$\frac{\Gamma \vdash e : (\tau, \tau')}{\Gamma \vdash \mathtt{fst}\ e : \tau}\ [\textsc{t-fst}] \qquad \frac{\Gamma \vdash e : (\tau, \tau')}{\Gamma \vdash \mathtt{snd}\ e : \tau'}\ [\textsc{t-snd}] \qquad \frac{\Gamma \vdash e : [e_s]\tau \quad \Gamma \vdash e' : \mathtt{int}}{\Gamma \vdash e[e'] : \tau}\ [\textsc{t-index}]$$

$$\frac{\Gamma \vdash e : \mathtt{int}}{\Gamma \vdash \mathtt{iota}\ e : [e]\mathtt{int}}\ [\textsc{t-iota}] \qquad \frac{\Gamma \vdash e_f : (x : \tau) \to \tau' \quad x \notin \mathrm{fv}(\tau') \quad \Gamma \vdash e_a : [e_s]\tau}{\Gamma \vdash \mathtt{map}\ e_f\ e_a : [e_s]\tau'}\ [\textsc{t-map}]$$

$$\frac{\Gamma \vdash e_c :\ \mathtt{int} \quad \Gamma \vdash e_t :\ \mu \quad \Gamma \vdash e_f :\ \mu}{\Gamma \vdash \mathtt{if}\ e_c\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f : \mu}\ [\textsc{t-if}] \qquad \frac{\Gamma, y : \tau \vdash e : \mu \quad \Gamma \vdash \tau\ \mathbf{ok}}{\Gamma \vdash \lambda(y : \tau).e :\ (y : \tau) \to \mu}\ [\textsc{t-lam}]$$

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash \tau\ \mathbf{ok} \quad \tau \sim_{\mathrm{s}} \tau'}{\Gamma \vdash e \triangleright \tau : \tau}\ [\textsc{t-coerce}] \qquad \frac{\Gamma \vdash e : (x : \tau) \to \mu \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \mu\{e'/x\}}\ [\textsc{t-app}]$$

$$\frac{\Gamma \vdash e : \exists \overline{x}.\tau \quad \Gamma, \overline{x} : \overline{\mathtt{int}}, y : \tau \vdash e' : \mu \quad \overline{x} \subseteq \mathrm{wit}(\tau) \quad \overline{x}, y \notin \mathrm{fv}(\mu)}{\Gamma \vdash \mathtt{let}\ [\overline{x}]\ y : \tau = e\ \mathtt{in}\ e' : \mu}\ [\textsc{t-let}]$$

$$\frac{\Gamma, \overline{z} : \overline{\mathtt{int}} \vdash e : \tau \quad \Gamma, y : \forall \overline{z}.\tau \vdash e' : \mu \quad \overline{z} \subseteq \mathrm{fv}(\tau) \quad y \notin \mathrm{fv}(\mu)}{\Gamma \vdash \mathtt{let}\ y\ [\overline{z}] : \tau = e\ \mathtt{in}\ e' : \mu}\ [\textsc{t-let-gen}]$$

Figure 4. Expression type rules.

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

$$\frac{}{\mathtt{int} \vdash_x v \rightsquigarrow \bullet}\ [\textsc{m-int}] \qquad \frac{}{(y : \tau) \to \mu \vdash_x v \rightsquigarrow \bullet}\ [\textsc{m-fun}] \qquad \frac{\tau_1 \vdash_x v_1 \rightsquigarrow \bullet \quad \tau_2 \vdash_x v_2 \rightsquigarrow r}{(\tau_1, \tau_2) \vdash_x (v_1, v_2) \rightsquigarrow r}\ [\textsc{m-pair2}]$$

$$\frac{\tau_1 \vdash_x v_1 \rightsquigarrow n}{(\tau_1, \tau_2) \vdash_x (v_1, v_2) \rightsquigarrow n}\ [\textsc{m-pair1}] \qquad \frac{}{[x]\tau \vdash_x [v_1, \cdots, v_n] \rightsquigarrow n}\ [\textsc{m-arr1}] \qquad \frac{x \neq e \quad \tau \vdash_x v_1 \rightsquigarrow r}{[e]\tau \vdash_x [v_1, \cdots, v_n] \rightsquigarrow r}\ [\textsc{m-arr2}]$$

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

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

Figure 5. Dynamic size matching

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

$$\frac{}{\rho \vdash n \triangleright \mathtt{int}}\ [\textsc{c-int}] \qquad \frac{\rho \vdash v_1 \triangleright \tau_1 \quad \rho \vdash v_2 \triangleright \tau_2}{\rho \vdash (v_1, v_2) \triangleright (\tau_1, \tau_2)}\ [\textsc{c-pair}] \qquad \frac{\rho \vdash e \rightsquigarrow n \quad \rho \vdash v_1 \triangleright \tau \cdots \rho \vdash v_n \triangleright \tau}{\rho \vdash [v_1, \cdots, v_n] \triangleright [e]\tau}\ [\textsc{c-arr}]$$

Figure 6. Dynamic size-checking for the language.

L-CLOS $\rho \models \langle x, e', \rho' \rangle : (x : \tau) \to \mu$ iff
$\forall v_1, v_2, (\rho \models v_1 : \tau \wedge \rho', x : v_1 \vdash e \rightsquigarrow v_2) \implies$
$\rho, x : v_1 \models v_2 :\ \mu$

L-UKWN $\rho \models v : \exists x.\mu$ iff $\exists n$ s.t. $\rho, x : n \models v : \mu$

L-SCHM $\rho \models v : \forall \overline{z}.\tau$ iff $\rho \models v : (\overline{z} : \overline{\mathtt{int}}) \to \tau$

We extend the logical relation point-wise to relate dynamic environments and well-formed contexts:

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

$$\frac{}{\rho \vdash n \rightsquigarrow n}\ [\text{D-INT}] \qquad \frac{\rho(x) = v}{\rho \vdash x \rightsquigarrow v}\ [\text{D-VAR}] \qquad \frac{\rho(x) = \langle \overline{z}, e_i, \rho' \rangle \quad \rho \vdash \overline{e} \rightsquigarrow \overline{n} \quad \rho', \overline{z} : \overline{n} \vdash e_i \rightsquigarrow v}{\rho \vdash x^{\overline{e}} \rightsquigarrow v}\ [\text{D-INST}]$$

$$\frac{\rho \vdash e_1 \rightsquigarrow v_1 \quad \rho \vdash e_2 \rightsquigarrow v_2}{\rho \vdash (e_1, e_2) \rightsquigarrow (v_1, v_2)}\ [\text{D-PAIR}] \qquad \frac{\rho \vdash e \rightsquigarrow (v_1, v_2)}{\rho \vdash \texttt{fst}\ e \rightsquigarrow v_1}\ [\text{D-FST}] \qquad \frac{\rho \vdash e \rightsquigarrow (v_1, v_2)}{\rho \vdash \texttt{snd}\ e \rightsquigarrow v_2}\ [\text{D-SND}]$$

$$\frac{\rho \vdash e_1 \rightsquigarrow [v_0, \ldots, v_{m-1}] \quad \rho \vdash e_2 \rightsquigarrow n \quad 0 \leq n < m}{\rho \vdash e_1[e_2] \rightsquigarrow v_n}\ [\text{D-INDEX}] \qquad \frac{}{\rho \vdash \lambda(x : \tau).e \rightsquigarrow \langle x, e, \rho \rangle}\ [\text{D-LAM}]$$

$$\frac{\rho \vdash e \rightsquigarrow v \quad \rho \vdash v \triangleright \tau}{\rho \vdash e \triangleright \tau \rightsquigarrow v}\ [\text{D-COERCE}] \qquad \frac{\rho \vdash e_1 \rightsquigarrow \langle x, e_0, \rho' \rangle \quad \rho \vdash e_2 \rightsquigarrow v' \quad \rho', x : v' \vdash e_0 \rightsquigarrow v}{\rho \vdash e_1\ e_2 \rightsquigarrow v}\ [\text{D-APP}]$$

$$\frac{\rho \vdash e \rightsquigarrow n \quad n > 0}{\rho \vdash \texttt{iota}\ e \rightsquigarrow [0, \cdots, n-1]}\ [\text{D-IOTA}] \qquad \frac{\rho \vdash e_l \rightsquigarrow n \quad \rho \vdash e_r \rightsquigarrow m \quad k = n \diamond m}{\rho \vdash e_l \diamond e_r \rightsquigarrow k}\ [\text{D-ARITH}]$$

$$\frac{\rho \vdash e_f \rightsquigarrow \langle x, e_0, \rho' \rangle \quad \rho \vdash e_a \rightsquigarrow [v_1, \cdots, v_n] \quad \rho', x : v_i \vdash e_0 \rightsquigarrow v_i'}{\rho \vdash \texttt{map}\ e_f\ e_a \rightsquigarrow [v_1', \cdots, v_n']}\ [\text{D-MAP}]$$

$$\frac{\rho \vdash e \rightsquigarrow n \quad n \neq 0 \quad \rho \vdash e_1 \rightsquigarrow v}{\rho \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow v}\ [\text{D-IF-0}] \qquad \frac{\rho \vdash e \rightsquigarrow 0 \quad \rho \vdash e_2 \rightsquigarrow v}{\rho \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow v}\ [\text{D-IF-F}]$$

$$\frac{\rho \vdash e \rightsquigarrow v \quad \tau \vdash_{\overline{x}} v \rightsquigarrow \overline{n} \quad \rho, \overline{x} : \overline{n}, y : v \vdash e' \rightsquigarrow v'}{\rho \vdash \texttt{let}\ [\overline{x}]\ y : \tau = e\ \texttt{in}\ e' \rightsquigarrow v'}\ [\text{D-LET}] \qquad \frac{\rho, y : \langle \overline{z}, e, \rho \rangle \vdash e' \rightsquigarrow v}{\rho \vdash \texttt{let}\ y\ [\overline{z}] : \tau = e\ \texttt{in}\ e' \rightsquigarrow v}\ [\text{D-LET-GEN}]$$

Figure 7. Dynamic semantics for the language.

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

1. $\mathrm{Dom}(\rho) = \mathrm{Dom}(\Gamma)$
2. $\vdash \Gamma$ **ok**
3. $\forall x \in \mathrm{Dom}(\Gamma), \rho \models \rho(x) : \Gamma(x)$

The requirement of $\vdash \Gamma$ **ok** allows us to ensure well–formedness of the typing context in all the inductives proofs that follow.

3.1 Properties of Substitutions

Typing and type well-formedness are preserved under substitution of a variable by an expression of a compatible type. This property allows us to ensure that the typing always produces well-formed types.

Proposition 3.1 (Substitution preserves well-formedness)**.** *If* $\Gamma \vdash e' : \tau'$ *then:*

1. $\Gamma, x : \tau' \vdash e : \tau \implies \Gamma \vdash e\{e'/x\} : \tau\{e'/x\}$
2. $\Gamma, x : \tau' \vdash \tau \sqsubseteq \tau'' \implies \Gamma \vdash \tau\{e'/x\} \sqsubseteq \tau''\{e'/x\}$
3. $\Gamma, x : \tau' \vdash \mu \sqsubseteq \mu' \implies \Gamma \vdash \mu\{e'/x\} \sqsubseteq \mu'\{e'/x\}$
4. $\Gamma, x : \tau' \vdash \tau$ **ok** $\implies \Gamma \vdash \tau\{e'/x\}$ **ok**
5. $\Gamma, x : \tau' \vdash \mu$ **ok** $\implies \Gamma \vdash \mu\{e'/x\}$ **ok**
6. $\Gamma, x : \tau' \vdash \sigma$ **ok** $\implies \Gamma \vdash \sigma\{e'/x\}$ **ok**

Proof. The first property is proven by induction on the typing derivation, utilising the proofs of the remaining properties, which are proven using mutual induction and the property that $\texttt{int}\{e/x\} = \texttt{int}$, for any e and x. □

Proposition 3.2 (Typing produces well-formed types)**.**

1. *If* $\vdash \Gamma$ **ok** *and* $\Gamma \vdash \mu \sqsubseteq \mu'$ *then* $\Gamma \vdash \mu$ **ok** *and* $\Gamma \vdash \mu'$ **ok**.
2. *If* $\vdash \Gamma$ **ok** *and* $\Gamma \vdash e : \mu$ *then* $\Gamma \vdash \mu$ **ok**.

Proof. These two properties are proven by induction on the subtyping derivation and the typing derivation, respectively, using Proposition 3.1. Most cases are straightforward, but some of them are a bit more interesting:

Case Arrow subtyping: In this case, the typing derivation of $\Gamma, x : \tau \vdash \mu'$ **ok** (from the inductive hypothesis) has to be slightly modified to insert T-RELAX rules just after every use of the x variable so that we can have $\Gamma, x : \tau' \vdash \mu'$ **ok** and then conclude.

Case [T-MAP]: we have $\lceil 1 \rceil\ \Gamma \vdash (x : \tau) \to \tau'$ **ok**, $\lceil 2 \rceil\ x \notin \mathrm{fv}(\tau')$, and $\lceil 3 \rceil\ \Gamma \vdash [e_s]\tau$ **ok** from [T-MAP] and the induction hypothesis. From $\lceil 1 \rceil$, we have $\Gamma, x : \tau \vdash \tau'$ **ok** and from $\lceil 2 \rceil$ we have $\lceil 4 \rceil\ \Gamma \vdash \tau'$ **ok**. From $\lceil 3 \rceil$ we have $\Gamma \vdash e_s : \texttt{int}$, which, combined with $\lceil 4 \rceil$, gives us $\Gamma \vdash [e_s]\tau'$ **ok**, as required.

Case [T-LET]: Here we have $\overline{x}, y \notin \mathrm{fv}(\mu)$, which allows us to remove $\overline{x}$ and y from $\Gamma, \overline{x} : \overline{\texttt{int}}, y : \texttt{int} \vdash\ \mu$ **ok**. □

Similar properties are also required for the dynamic semantics. We begin with properties about value equivalence, showing that we can effectively *swap* two equivalent values and still obtain equivalent results.

Proposition 3.3 (Properties about value equivalence)**.**

1. *If* $v \sim_{\mathrm{v}} v'$ *and* $\rho, x : v_0 \vdash v \triangleright \tau$ *then* $\rho, x : v_0 \vdash v' \triangleright \tau$
2. *If* $v \sim_{\mathrm{v}} v'$ *and* $\tau \vdash_{\overline{x}} v \rightsquigarrow \overline{n}$ *then* $\tau \vdash_{\overline{x}} v' \rightsquigarrow \overline{n}$
3. *If* $v \sim_{\mathrm{v}} v'$ *and* $\rho, x : v \vdash e \rightsquigarrow v_r$ *then* $\rho, x : v' \vdash e \rightsquigarrow v'_r$ *and* $v_r \sim_{\mathrm{v}} v'_r$
4. *If* $v_0 \sim_{\mathrm{v}} v'_0$ *and* $\rho, x : v_0 \vdash v \triangleright \tau$ *then* $\rho, x : v'_0 \vdash v \triangleright \tau$
5. *If* $v \sim_{\mathrm{v}} v'$ *and* $\rho \models v : \mu$ *then* $\rho \models v' : \mu$

Proof. Properties 1, 2, and 5 are shown by straightforward induction, while Properties 3 and 4 are shown by mutual induction. □

The notion of value equivalence is used for specifying the following substitution properties:

Proposition 3.4 (Substitution preserves semantics)**.**

1. *If* $\rho, x : v_0 \vdash e \rightsquigarrow v$ *and* $\rho \vdash e' \rightsquigarrow v_0$ *then* $\rho \vdash e\{e'/x\} \rightsquigarrow v'$ *and* $v \sim_{\mathrm{v}} v'$
2. *If* $\rho, x : v_0 \vdash v \triangleright \tau$ *and* $\rho \vdash e' \rightsquigarrow v_0$ *then* $\rho \vdash v \triangleright \tau\{e'/x\}$
3. *If* $\rho \vdash e\{e'/x\} \rightsquigarrow v$ *and* $\rho \vdash e' \rightsquigarrow v_0$ *then* $\rho, x : v_0 \vdash e \rightsquigarrow v'$ *and* $v \sim_{\mathrm{v}} v'$
4. *If* $\rho \vdash v \triangleright \tau\{e'/x\}$ *and* $\rho \vdash e' \rightsquigarrow v_0$ *then* $\rho, x : v_0 \vdash v \triangleright \tau$
5. *If* $\rho, x : v_0 \models v : \mu$ *and* $\rho \vdash e \rightsquigarrow v_0$ *then* $\rho \models v : \mu\{e/x\}$
6. *If* $\rho \models v : \mu\{e/x\}$ *and* $\rho \vdash e \rightsquigarrow v_0$ *then* $\rho, x : v_0 \models v : \mu$

Proof. Properties 1 and 2 are shown by mutual induction on the derivations of evaluation and dynamic size checking. Properties 3 and 4 are shown by mutual induction on the structure of e, and Properties 5 and 6 are shown by induction using Properties 1 and 3. □

We say that two environments ρ and ρ' "*agree on a set* X *of variables*", noted $\rho \approx_X \rho'$, if and only if $\forall x \in X, \rho(x) \sim_{\mathrm{v}} \rho'(x)$, that is, they are *point-wise equivalent* on the set X. We can show by a straightforward induction (or just applying precedent properties) that agreement on free variables allows us to replace an environment by the other.

Proposition 3.5 (Logical relation extensibility)**.**

1. *If* $\rho \vdash e \rightsquigarrow v$ *and* $\rho' \approx_{\mathrm{fv}(e)} \rho$ *then* $\rho' \vdash e \rightsquigarrow v'$ *and* $v' \sim_{\mathrm{v}} v$
2. *If* $\rho \vdash v \triangleright \tau$ *and* $\rho' \approx_{\mathrm{fv}(\tau)} \rho$ *then* $\rho' \vdash v \triangleright \tau$
3. *If* $\rho \models v : \tau$ *and* $\rho' \approx_{\mathrm{fv}(\tau)} \rho$ *then* $\rho' \models v : \tau$
4. *If* $\rho \models v : \mu$ *and* $\rho' \approx_{\mathrm{fv}(\mu)} \rho$ *then* $\rho' \models v : \mu$
5. *If* $\rho \models v : \sigma$ *and* $\rho' \approx_{\mathrm{fv}(\sigma)} \rho$ *then* $\rho' \models v : \sigma$
6. *If* $\rho \models \Gamma$ *and* $\rho' \approx_{\mathrm{Dom}(\Gamma)} \rho$ *then* $\rho' \models \Gamma$

3.2 Dynamic Properties

We finally show some properties linking the type rules and dynamic semantics. The following two properties relate dynamic size matching and value typing to ensure dynamic size matching has the semantics we want, even if not required in the soundness proof.

Proposition 3.6 (All witnesses can be extracted)**.**

1. *If* $\rho \models v : \exists x.\tau$ *and* $x \in \mathrm{wit}(\tau)$ *then* $\exists n, \tau \vdash_x v \rightsquigarrow n$.
2. *If* $\exists n, \tau \vdash_x v \rightsquigarrow n$ *then* $x \in \mathrm{wit}(\tau)$

Proof. Both properties are proven by induction on the derivation of the hypothesis. □

Notice that the first property uses that an array is non-empty, because, in the current formalisation, an empty array has the property that for any τ, we have $\rho \models v : [0]\tau$, but $[0][x]\tau \vdash_x v$ would fail as it would require the first value of this empty array.

Proposition 3.7 (Dynamic size matching)**.**

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

Proof. Proved by induction on the structure of τ. □

We also ensure that *dynamic size checking* correctly ensures the types of values by this property, shown by induction on the check.

Proposition 3.8 (Dynamic size checking)**.**
If $\rho \vdash v \triangleright \tau$ *then* $\rho \models v : \tau$

Combining these properties, we obtain soundness, which guarantees a well-typed result from a well-typed program. This property assumes the program terminates, and does not apply to failing programs. The proof relies on the subtyping also applying to value typing. The complete proof of these two properties is based on mutual induction and can be found in the appendix.

Proposition 3.9 (Value subtyping)**.**
If $\rho \models \Gamma$ *and* $\Gamma \vdash \mu \sqsubseteq \mu'$ *and* $\rho \models v : \mu$ *then* $\rho \models v : \mu'$.

Proposition 3.10 (Soundness)**.**
If $\Gamma \vdash e : \mu$ *and* $\rho \models \Gamma$ *and* $\rho \vdash e \rightsquigarrow v$ *then* $\rho \models v : \mu$.

As emphasised earlier, the soundness result considers only value-terminating expressions, which allows us to avoid specifying dynamic evaluation rules for propagating dynamic errors when dynamic size checking fails (i.e., rule C-ARR and rule D-COERSE), when array indexing fails (i.e., rule D-INDEX), or when attempting to create an array with fewer than one element (i.e., rule D-IOTA). We emphasise here that we are guaranteed by the type system that the dynamic extraction of sizes (i.e., rule D-LET) succeed. Proper handling of inner sizes of empty arrays requires that shape information is available dynamically in cases where an array can be empty. Whereas Futhark happily works with empty arrays, we consider it future work to amend the formalism to support empty arrays.

By specifying also dynamic evaluation rules for propagating dynamic errors, it would not be difficult to establish a termination result based on a logical relation proof similar to the one we have presented. Such a proof would closely follow Tait's strong-normalisation proof from 1967 for the simply-typed lambda calculus [22], which has later been established as a fundamental proof technique [8] and adapted for many other use cases [4].

4 Inference and Normalisation

In the following we will discuss how to adapt F to be a useful programming language, rather than merely a calculus. In our examples we will assume the existence of functions with the following type schemes:

$$\begin{aligned}\texttt{filter} &: \forall[n].(\texttt{int} \to \texttt{int}) \to [n]\texttt{int} \to \exists m.[m]\texttt{int}\\ \texttt{zip} &: \forall[n].[n]\texttt{int} \to [n]\texttt{int} \to [n](\texttt{int},\texttt{int})\\ \texttt{length} &: \forall[n].[n]\texttt{int} \to \texttt{int}\end{aligned}$$

The language F requires subexpressions producing unknown sizes to be immediately let-bound, which largely requires the program to be in Administrative Normal Form (ANF) [19]. This requirement simplifies the theoretical treatment. Consider the expression $\texttt{length}^?\ (\texttt{filter}\ p\ x)$. Here we need to instantiate `length` with the size of the result of `filter`, but this size has no name. Requiring $\texttt{filter}\ p\ x$ to be let-bound lets us introduce a name for this size.

This constraint is impractical for a real programming language. Fortunately, it is straightforward to rewrite a program with arbitrary nested expressions to be in ANF. Together with a type inference algorithm quite similar to conventional Hindley-Milner type inference [17], where we treat sizes as type parameters of a distinct kind, and where we perform unification on expressions, we construct a language with implicit size polymorphism.

However, for the ANF transformation to be possible, we need to require programs to be *causally coherent* with respect to how sizes of arrays appear in types.

4.1 Causality

For the purpose of type inference, we define an evaluation order where arguments are evaluated before function expressions: that is, right to left. We consider `iota` and `map` intrinsics as functions for this rule. The type checker rejects programs that are not well-typed in this form.

To illustrate, consider the following program, where we have elided types and polymorphic instantiation:

```
let iiota [n] = iota n in
let t = iiota in
let xs = filter (λx.x ≤ 5) (iota 10) in
zip t xs
```

Here `iiota` is an implicit version of `iota`, where its instantiation determines the size. The problem with this program is that we cannot infer the size of `t`. By the type of `zip`, it must be the same as the size of `xs`, the size of `xs` being defined at its definition because of the existential type of `filter`. So, when writing the inferred types, we have:

```
let iiota [n] : [n]int = iota n in
let t : [d]int = iiota^d in
let [d] xs : [d]int = filter (λx.x ≤ 5) (iota 10)in
zip t xs
```

The introduced size `d` is used to instantiate `iiota` in the definition of `t` before `d` is in scope, hence this program is not *causally coherent*. In this case we could flip the definition order of `t` and `xs`, but there are (contrived) cases where a circular dependency exists—consider if the filtering function made use of the size of `t`. While it might be possible to check whether *any* program ordering produces a well-typed program, in order to keep the rules for the programmer simple, we have defined a single evaluation order that determines when sizes are available. In practice we have found that few programs encounter this restriction, and the universal solution is to explicitly `let`-bind the expression producing the unknown size.

4.2 Inference

Type inference deduces the instantiation of polymorphic bindings, allowing us to write

$$\texttt{length}\ (\texttt{iota}\ e)$$

for

$$\texttt{length}^e\ (\texttt{iota}\ e).$$

This is done using an adapted version of Hindley-Milner type inference, where sizes exist as a kind of type variable. Since expressions can now appear in types, this involves the definition of unification (and equality) of expressions. One quirk is that we use entirely syntactical unification, meaning that the types $[a+b]\tau$ and $[b+a]\tau$ do not unify—despite arrays of these types obviously having the same dynamic size, due to the commutativity of integer addition. It would certainly be possible to perform some kind of arithmetic normalisation to recognise that these two expressions are equivalent, but doing so might have unintended effects. The reason is that F allows size-polymorphic functions of the form

$$\texttt{let tricky [n][m]} = \lambda(\texttt{x} : [\texttt{n}+\texttt{m}]\texttt{int}).(\texttt{n},\texttt{m})$$

where the *static structure* of the size of the argument affects the result of evaluation. This means that the terms

$$\texttt{tricky}\ (\texttt{iota}\ (1+2))$$

and

$$\texttt{tricky}\ (\texttt{iota}\ (2+1))$$

are not equivalent. In types, arithmetic operations can be seen as *un-interpreted* constants. It would be quite undesirable for arbitrary choices made by a type inference algorithm (e.g., reordering terms in an addition) to influence the program result.

It is unclear whether such a strictly syntactical view of types is ultimately beneficial, but it does avoid surprises. We are considering a form of *static* size coercions that allows changes in the structure of size expressions, but verifies that they will be dynamically equivalent.

4.2.1 Duplication of computation. The instantiation of polymorphic bindings can duplicate expressions, as shown in the `length` example above, where e is duplicated. According to the model of size parameters as implicit parameters, this is not just a syntactical duplication—that e must actually be evaluated, resulting in duplication of computation. Worse, if the instantiation is in a function that is passed to `map`, this can result in an asymptotic cost increase if e is a costly expression. This issue can be avoided by fully ANF-transforming the program such that all subexpressions are bound to names, as these can be duplicated without computational cost.

4.2.2 Negative sizes. While sizes must be non-negative, they can be compound expressions where subterms are negative. This property creates interesting possibilities for program errors. Suppose we have a size-polymorphic function for unflattening arrays:

$$\texttt{unflatten} : \forall[n][m].[n \cdot m]\texttt{int} \to [n][m]\texttt{int}$$

Now consider an array $\texttt{iota}(-2 \cdot -3)$. Because $-2 \cdot -3 > 0$, this is a valid array of type $[-2 \cdot -3]\texttt{int}$. This means the expression $\texttt{unflatten}(\texttt{iota}(-2 \cdot -3))$ has type $[-2][-3]\texttt{int}$. But such an array can of course never be constructed at runtime—`unflatten` will fail dynamically. The specific failure is likely invoking $\texttt{iota}(-2)$. In order to prevent the program from going wrong, a non-syntactic precondition must be applied to the type of `unflatten`: n and m must be non-negative.

Similarly, for extracting the second element of an array we may be tempted to define a function of the following type:

$$\texttt{snd} : \forall[n].[2+n]\texttt{int} \to \texttt{int}$$

Again, this specification does not provide any safety regarding the number of elements in the argument array, as n can simply be instantiated with a negative number as in the expression $\texttt{snd}(\texttt{iota}(2 + (-1)))$.

Most dependently typed languages solve this issue by making sizes natural numbers. We could also do that, but it complicates the language (by introducing yet another number type) and opens the question about how to handle subtraction. In Idris, a `Nat` subtraction that produces a negative number is simply truncated to zero, but it is reasonable for the size of an array to be described by a computation where intermediate results are negative, as long as the final result is not. We conjecture that a proper solution to this issue involves a notion of *refinements*, inspired by Dependent ML [26] and Liquid Haskell [25], such that inequalities such as $\texttt{n} > 0$ can be imposed on size parameters.

5 Implementation

The ideas behind the F type system has been implemented in a prototype version of the compiler for the functional array language Futhark, using a slightly different syntax. The implementation automatically performs inference and normalisation (Section 4) as necessary. The T-RELAX rule, which can in principle be applied anywhere, is applied only when a variable used in the return type of a `let`-binding or function body goes out of scope. The implementation supports empty arrays through a value representation that explicitly stores the full shape, which allows the size to be extracted even when an outer dimension is zero. Empty array literals are allowed whenever type inference can determine the shape of the elements. Futhark also supports conventional parametric polymorphism, which permits `map` to be a standard function, although it requires constraints on type variables to guarantee regularity of arrays [11].

5.1 Type-level programming

Parametric polymorphism, together with the ability to define abstract types, allows a provably-safe coercion facility to be defined by the programmer, as shown on Figure 8. The abstract type `eq[n][m]` is parametric in the two sizes `n` and `m`, and encodes a witness that `n==m`. The `coerce` function accepts a proof that `n==m` and can convert an array of size `n` into an array of size `m`. The definition of `coerce` uses a dynamically checked size coercion, but if we ensure that only valid values of `eq` can be constructed, we know this coercion cannot fail.

This form of type level programming is well known in the functional programming community, although allowing normal expressions in types makes it more convenient than having to define type-level analogues for term-level constructs. The downside of this approach is that it requires the programmer to manually assemble proof terms, as shown in the `proof` binding at the bottom of Figure 8, which is very tedious without automation. We do not expect this style of programming to be acceptable to the majority of Futhark programmers, and instead investigate automation by exploiting the fact that the majority of size expressions in real Futhark programs are simple arithmetic expressions where equivalence can be decided using standard techniques, such as Fourier-Motzkin elimination.

5.2 Performance

Futhark is intended for high performance computing, and any overhead induced by the type system must be carefully considered. In this respect, all modules, all type polymorphism, and all higher-order functions are eliminated at compile time [6, 13]. The only dynamic checks arising from the size type system are in explicit coercions, although `iota` and array indexing is also dynamically checked [9]. Further, no matter how complicated size expressions may be at compile time, at run-time each array dimension is associated with a single 64-bit integer, meaning there is no storage overhead.

```
module meta : {
  type eq[n][m]                                                  -- Proof of n==m
  val coerce     [n][m] 't : eq[n][m] -> [n]t -> [m]t            -- Use a proof
  val refl             [n] : eq[n][n]                            -- Equality properties
  val comm           [n][m] : eq[n][m] -> eq[m][n]
  val trans      [n][m][k] : eq[n][m] -> eq[m][k] -> eq[n][k]
  val plus_comm      [a][b] : eq[a+b][b+a]                       -- Axioms
  val plus_assoc [a][b][c] : eq[(a+b)+c][a+(b+c)]
  val plus_lhs   [a][b][c] : eq[a][b] -> eq[a+c][b+c]
  val plus_rhs   [a][b][c] : eq[c][b] -> eq[a+c][a+b]
} = {
  type eq[n][m] = [0][n][m]()
  def coerce [n][m] 't (_: eq[n][m]) (a: [n]t) = a :> [m]t
  def refl = []
  ...
}

def main [n][m][l] (xs: [n]i32) (ys: [m]i32) (zs: [l]i32) =
  let proof : meta.eq[m+(n+l)][(n+m)+l] =
    meta.trans (meta.comm meta.plus_assoc) (meta.plus_lhs meta.plus_comm)
  in zip ((xs ++ ys) ++ zs) (meta.coerce proof (ys ++ (xs ++ zs)))
```

Figure 8. Type level metaprogramming example in Futhark. The ++ operator denotes concatenation. The definitions within the first set of braces constitute the module type, which can be considered an abstract interface. The definitions within the second set of braces are the actual definitions of the names specified by the module type.

6 Related Work

The literature on both dependent types and type systems for array programming is long and rich. Here we will focus on their combination, as that is where our contribution lies.

Qube [24] uses dependent types to model APL-style shape polymorphism, utilising an SMT solver to solve complex constraints. Qube requires the array element type to be known, which in particular means that it does not support conventional parametric polymorphism. A later work in this area is Remora [21], which interestingly supports arrays of functions, allowing a programming style that encodes control flow as data. This is also possible using the fragment of dependent types in Haskell [7], and of course in full dependently typed languages, such as Agda, as shown in [23].

Although Dependent ML [26] is not an array language, its type system can be used to express predicates on lengths of lists, in a way that is similar to our equality constraints on shapes. Dependent ML guarantees efficient compilation through erasure, but type checking is undecidable, as subtyping can involve checking arbitrary predicates. In practice, this is done by invoking external SMT solvers. An alternative approach is restricting the refinements (i.e., the dependent parts of types) to a decidable sublanguage.

Array type systems not based on dependent types have also been investigated. Jay [14, 15] developed a theory of "shapely programs", which are programs where the shape of a function result depends only on the shape of its arguments. This is used to extract a condensed *size program*, that can be statically evaluated to determine whether the original array program contains any violations of shape constraints. There are two main differences from our work. First, we allow existential sizes, which is necessary for functions such as `filter`. Second, our approach is based on conventional type checking, rather than on static evaluation. Array type systems for tracking ranks of arrays and the sizes of shape vectors have also been been used for compiling a subset of APL to Futhark through an array intermediate language [5, 10]. This work relies on a representation of arrays that separates the shape vector from the implementation array, which is also the foundation for other array language formalisms [18] and implementations, such as [20].

Another recent work aimed at embedded programming uses a type system that tracks sizes known at compile time, and where ML-style parametric polymorphism is used to propagate sizes through polymorphic functions [16]. Like our work, the goal is to strike a balance between expressiveness and the ability to verify and infer program properties. However, as befits the domain of embedded programming, they tend to prioritise the latter. Their key restriction is that size expressions can only be multivariate polynomials, where equality is decidable in their context, while we allow arbitrary expressions and permit size coercions as a dynamically checked escape hatch.

7 Conclusions

We have presented an ML-like type system for array programming that supports complex array shape equality constraints. We have formally described the type system and informally explained how the type system is used in the Futhark programming language. Size constraints are checked syntactically and the type system allows for concise size polymorphism with implicit arguments. The syntactic approach is quite conservative, and as an escape hatch we permit dynamically checked size coercions.

There are several possibilities for future work. First, it may be frustrating for a programmer to have type errors arising from $n + m \neq m + n$. Besides using dynamic type coercions, size equalities could be discharged by compile-time equation solving as done in Qube. However, this approach raises questions regarding size polymorphism, as unification-based type inference is then insufficient to infer polymorphic arguments. Second, we have not here formalised type polymorphism, even though size polymophism is supported and formalised. Finally, the problems arising from empty arrays are not dealt with in the formalisation, which make us require the strong assumption that arrays are never empty.

A Proofs of Proposition 3.10

Proposition 3.9 (Value subtyping).
If $\rho \models \Gamma$ *and* $\Gamma \vdash \mu \sqsubseteq \mu'$ *and* $\rho \models v : \mu$ *then* $\rho \models v : \mu'$.

Proof. Shown by induction on the derivation of $\Gamma \vdash \mu \sqsubseteq \mu'$.

In every cases, we have the hypothesis $\lceil 0 \rceil$ $\rho \models \Gamma$.

Case $\Gamma \vdash \mu' \sqsubseteq \exists x.\mu$: We have by hypothesis $\lceil 1 \rceil$ $\rho \models v : \mu'$, and the hypothesis of the derivation are $\lceil 2 \rceil$ $\Gamma \vdash e : \texttt{int}$ and $\lceil 3 \rceil$ $\Gamma \vdash \mu' \sqsubseteq \mu\{e/x\}$.

By induction, we have $\lceil 4 \rceil$ $\rho \models v : \mu\{e/x\}$. We then distinguish two cases:

1. $\rho \vdash e \rightsquigarrow n$ then by 3.4.6, we have $\rho, x : n \models v : \mu$ and so $\rho \models v : \exists x.\mu$ as required.
2. Else e can't be evaluated in ρ (if it is to v', then by soundness $v' \models \texttt{int}$: it's the first sub-case) so for every expression containing x in μ, x is not evaluated (as the full expression can be evaluated) so, even if $\mathrm{fv}(\mu) \not\subseteq \mathrm{Dom}(\rho)$, $\rho \models v : \mu$, we can then have $\rho \models v : \exists x.\mu$ as required.

Case $\Gamma \vdash \exists x.\mu \sqsubseteq \exists x.\mu'$: We have by hypothesis $\lceil 1 \rceil$ $\rho \models v : \exists x.\mu$, and the hypothesis of the derivation is $\lceil 2 \rceil$ $\Gamma, x : \texttt{int} \vdash \mu \sqsubseteq \mu'$.

$\lceil 1 \rceil$ can be derived to $\rho, x : n \models v : \mu$ for some n, and then by induction we have $\rho, x : n \models v : \mu'$. We then have $\rho \models v : \exists x.\mu'$ as required.

Case $\Gamma \vdash \texttt{int} \sqsubseteq \texttt{int}$: Nothing to say.

Case $\Gamma \vdash (x : \tau) \to \mu \sqsubseteq (x : \tau') \to \mu'$: We have by hypothesis $\lceil 1 \rceil$ $\rho \models v : (x : \tau) \to \mu$, and the hypothesis of the derivation are $\lceil 2 \rceil$ $\Gamma \vdash \tau' \sqsubseteq \tau$ and $\lceil 3 \rceil$ $\Gamma, x : \tau \vdash \mu \sqsubseteq \mu'$. We know by L-CLOS, $v = \langle x, e, \rho' \rangle$.

We want to prove $\lceil goal \rceil$ $\forall v_1, v_2, (\rho \models v_1 : \tau' \wedge \rho', x : v_1 \vdash e \rightsquigarrow v_2) \implies \rho, x : v_1 \models v_2 : \mu'$.

For given v_1 and v_2, we asume $\lceil 4 \rceil$ $\rho \models v_1 : \tau'$ and $\lceil 5 \rceil$ $\rho', x : v_1 \vdash e \rightsquigarrow v_2$.

By induction with $\lceil 2 \rceil$ and $\lceil 4 \rceil$, we have $\lceil 6 \rceil$ $\rho \models v_1 : \tau$, so with $\lceil 5 \rceil$ in $\lceil 1 \rceil$, we obtain $\lceil 7 \rceil$ $\rho, x : v_1 \models v_2 : \mu$.

We can extend $\lceil 0 \rceil$ with $\lceil 6 \rceil$ to have $\rho, x : v_1 \models \Gamma, x : \tau$. So, with $\lceil 7 \rceil$ and $\lceil 3 \rceil$, we have $\rho, x : v_1 \models v_2 : \mu'$.

We can then conclude by abstracting v_1 and v_2.

Case $\Gamma \vdash [e]\tau \sqsubseteq [e]\tau'$: We have by hypothesis $\lceil 1 \rceil$ $\rho \models v : [e]\tau$, and the hypothesis of the derivation is $\lceil 2 \rceil$ $\Gamma \vdash \tau \sqsubseteq \tau'$. We know by L-ARR, $\lceil 3 \rceil$ $\rho \vdash e \rightsquigarrow n$ and $\lceil 4 \rceil$ $\forall i, \rho \models v_i : \tau$ (with $v = [v_1, \cdots, v_n]$).

by induction, we have $\forall i, \rho \models v_i : \tau'$, and then by L-ARR, we have $\rho \models v : [e]\tau'$ as required.

Case $\Gamma \vdash (\tau_1, \tau_2) \sqsubseteq (\tau_1', \tau_2')$: Much like the precedent case, we unfold L-PAIR, then refold it after using the induction. □

Proposition 3.10 (Soundness).
If $\Gamma \vdash e : \mu$ *and* $\rho \models \Gamma$ *and* $\rho \vdash e \rightsquigarrow v$ *then* $\rho \models v : \mu$.

Proof. Shown by induction on the derivation of $\Gamma \vdash e : \mu$.

In every cases, we have the hypothesis $\lceil 0 \rceil$ $\rho \models \Gamma$.

Case T-INT: Direct with $e = v = n$ for some n.

Case T-VAR: By $\rho \models \Gamma$, $\rho \models v = \rho(x) : \Gamma(x) = \mu$.

Case T-INST: Similar to the case T-APP. By breaking the T-INST hypothesis we have $\lceil 2 \rceil$ $\Gamma(x) = \forall \overline{z}.\tau$. By breaking the rule D-INST, we have $\lceil 3 \rceil$ $\rho \vdash \overline{e} \rightsquigarrow \overline{n}$, $\lceil 4 \rceil$ $\rho(x) = \langle \overline{z}, e_i, \rho' \rangle$ and $\lceil 5 \rceil$ $\rho', \overline{z} : \overline{n} \vdash e_i \rightsquigarrow v$. By $\lceil 0 \rceil$, $\rho \models \rho(x) : \Gamma(x)$, and so we have $\lceil 6 \rceil$ $\rho \models \langle \overline{z}, e_i, \rho' \rangle : (\overline{z} : \overline{\texttt{int}}) \to \tau$. And with the rule L-INT, $\lceil 7 \rceil$ $\rho \models \overline{n} : \overline{\texttt{int}}$.

We apply the property of L-CLOS of $\lceil 6 \rceil$ with the hypothesis $\lceil 7 \rceil$ and $\lceil 5 \rceil$ to have $\lceil 8 \rceil$ $\rho, \overline{z} : \overline{n} \models v : \tau$. Then, by applying 3.4.1 over $\lceil 3 \rceil$, and knowing that two integers are similar if and only equals, we obtain $\rho \models v : \tau\{\overline{e}/\overline{z}\}$ as required.

Case T-ARITH: The hypothesis are $\rho \vdash e_l \diamond e_r \rightsquigarrow k$ and $\Gamma \vdash e_l \diamond e_r : \texttt{int}$. By the rule L-INT, we have $\rho \models k : \texttt{int}$ as required.

Case T-RELAX: Breaking T-RELAX, we have $\lceil 1 \rceil$ $\Gamma \vdash e : \mu$ and $\lceil 2 \rceil$ $\Gamma \vdash \mu \sqsubseteq \mu'$. Also we have, $\rho \vdash e \rightsquigarrow v$. By induction we have $\rho \models v : \mu$, then by proposition 3.9, $\rho \models v : \mu'$ as required.

Case T-PAIR: Breaking the rule T-PAIR we have $\lceil 1 \rceil$ $\Gamma \vdash e_1 : \tau_1$ and $\lceil 2 \rceil$ $\Gamma \vdash e_2 : \tau_2$, and then by breaking D-PAIR we obtain $\lceil 3 \rceil$ $\rho \vdash e_1 \rightsquigarrow v_1$ and $\lceil 4 \rceil$ $\rho \vdash e_2 \rightsquigarrow v_2$.

By induction on $\lceil 0 \rceil$, $\lceil 1 \rceil$ and $\lceil 3 \rceil$, and a second time on $\lceil 0 \rceil$, $\lceil 2 \rceil$ and $\lceil 4 \rceil$, we obtain $\lceil 5 \rceil$ $\rho \models v_1 : \tau_1$ and $\lceil 6 \rceil$ $\rho \models v_2 : \tau_2$. We then use L-PAIR to have $\rho \models (v_1, v_2) : (\tau_1, \tau_2)$ as required.

Case T-FST: Breaking the rules T-FST and D-FST, we have $\lceil 1 \rceil$ $\Gamma \vdash e : (\tau_1, \tau_2)$ and $\lceil 2 \rceil$ $\rho \vdash e \rightsquigarrow (v_1, v_2)$. By induction we obtain $\lceil 3 \rceil$ $\rho \models (v_1, v_2) : (\tau_1, \tau_2)$. We then break L-PAIR to have $\rho \models v_1 : \tau_1$ as required.

It's the same proof for T-SND and T-INDEX, by replacing L-PAIR by L-ARR.

Case t-iota: By breaking the rule d-iota, we have $\rho \vdash e \rightsquigarrow n$. Knowing that $\forall i, \rho \models i - 1 : \texttt{int}$, we can use l-arr to have $\rho \models [0, \cdots, n-1] : [e]\texttt{int}$ as required.

Case t-map: By breaking the rule t-map, we have $\lceil 1 \rceil\ \Gamma \vdash e_f : (x : \tau) \to \tau'$, $\lceil 2 \rceil\ x \notin \text{fv}(\tau')$ and $\lceil 3 \rceil\ \Gamma \vdash e_a : [e_s]\tau$, then by breaking d-map, we have $\lceil 4 \rceil\ \rho \vdash e_f \rightsquigarrow \langle x, e, \rho' \rangle$, $\lceil 5 \rceil\ \rho \vdash e_a \rightsquigarrow [v_1, \cdots, v_n]$ and $\lceil 6 \rceil\ \forall i, \rho', x : v_i \vdash e \rightsquigarrow v_i'$.

We apply the induction on $\lceil 0 \rceil$, $\lceil 1 \rceil$ and $\lceil 4 \rceil$, and a second times on $\lceil 0 \rceil$, $\lceil 3 \rceil$ and $\lceil 5 \rceil$ to have $\lceil 7 \rceil\ \rho \models \langle x, e, \rho' \rangle : (x : \tau) \to \tau'$ and $\lceil 8 \rceil\ \rho \models [v_1, \cdots, v_n] : [e_s]\tau$. Breaking the rule l-arr of $\lceil 8 \rceil$, we have $\lceil 10 \rceil\ \forall i, \rho \models v_i : \tau$.

We then apply the rule l-clos of $\lceil 7 \rceil$ with $\lceil 6 \rceil$ and $\lceil 10 \rceil$ to have $\lceil 11 \rceil\ \forall i, \rho, x : v_i \models v_i' : \tau'$.

By 3.2, we know that $\Gamma, x : \tau \vdash \tau'$ **ok**, so, with $\lceil 2 \rceil$, we can have $\lceil 12 \rceil\ \text{fv}(\tau') \subseteq \text{Dom}(\rho) = \text{Dom}(\Gamma)$. We can then transform $\lceil 11 \rceil$ into $\lceil 11' \rceil\ \forall i, \rho \models v_i' : \tau'$. We also have $\lceil 14 \rceil$ $\rho \vdash e_s \rightsquigarrow n$ by $\lceil 8 \rceil$.

We can then conclude with l-arr, $\lceil 14 \rceil$ and $\lceil 11' \rceil$ to have $\rho \models [v_1', \cdots, v_n'] : \tau'$ as required.

Case t-if: By breaking the rule t-if, we have $\lceil 1 \rceil\ \Gamma \vdash e_t : \mu$ and $\lceil 2 \rceil\ \Gamma \vdash e_f : \mu$.

We distinguish two sub-cases:

1. The evaluation is done with d-if-0: we then have $\lceil 3 \rceil$ $\rho \vdash e_t \rightsquigarrow v$ so by induction we obtain $\rho \models v : \mu$ as required
2. The evaluation is done with d-if-f: we then have $\lceil 4 \rceil$ $\rho \vdash e_f \rightsquigarrow v$ so by induction we obtain $\rho \models v : \mu$ as required

Case t-lam: By breaking the hypothesis $\lceil 0 \rceil$, we have $\lceil 0_1 \rceil$ $\text{Dom}(\Gamma) = \text{Dom}(\rho)$, $\lceil 0_2 \rceil \vdash \Gamma$ **ok** and $\lceil 0_3 \rceil\ \forall x \in \text{Dom}(\Gamma), \rho \models \rho(x) : \Gamma(x)$. By breaking the rule t-lam, we have $\lceil 1 \rceil\ \Gamma, y : \tau \vdash e : \mu$ and $\lceil 2 \rceil\ \Gamma \vdash \tau$ **ok**.

We need to prove $\lceil goal \rceil\ \forall v_1, v_2, (\rho \models v_1 : \tau \wedge \rho, y : v_1 \vdash e \rightsquigarrow v_2) \implies \rho, y : v_1 \models v_2 : \mu$.

For given v_1, v_2, we suppose $\lceil 3 \rceil\ \rho \models v_1 : \tau$ and $\lceil 4 \rceil\ \rho, y : v_1 \vdash e \rightsquigarrow v_2$. We transform $\lceil 3 \rceil$ into $\lceil 3' \rceil\ \rho, y : v_1 \models v_1 : \tau$ with $y \notin \text{fv}(\tau)$.

We define $\Gamma' = \Gamma, y : \tau$ and $\rho' = \rho, y : v_1$. We then have $\lceil 6_1 \rceil$ $\text{Dom}(\Gamma') = \text{Dom}(\rho')$ from $\lceil 0_1 \rceil$ and definitions, $\lceil 6_2 \rceil$ $\vdash \Gamma'$ **ok** from $\lceil 0_2 \rceil$, $\lceil 2 \rceil$ and 3.2, and $\lceil 6_3 \rceil\ \forall x \in \text{Dom}(\Gamma'), \rho' \models \rho'(x) : \Gamma'(x)$ from $\lceil 0_3 \rceil$ and $\lceil 3' \rceil$. We can merge them into $\lceil 6 \rceil\ \rho' \models \Gamma'$. Then by induction with $\lceil 6 \rceil$ on $\lceil 1 \rceil$ and $\lceil 4 \rceil$ we have $\rho, y : v_1 \models v_2 : \mu$.

We then have $\lceil goal \rceil$ by abstracting v_1 and v_2.

Case t-coerce: Directly obtainned with 3.8 of $\rho \vdash v \triangleright \tau$

Case t-app: By breaking the rule t-app, we have $\lceil 1 \rceil\ \Gamma \vdash e : (y : \tau) \to \mu$ and $\lceil 2 \rceil\ \Gamma \vdash e' : \tau$. By breaking the rule d-app, we have $\lceil 3 \rceil\ \rho \vdash e \rightsquigarrow \langle y, e_c, \rho' \rangle$, $\lceil 4 \rceil\ \rho \vdash e' \rightsquigarrow v'$ and $\lceil 5 \rceil$ $\rho', y : v' \vdash e_c \rightsquigarrow v$.

By induction on $\lceil 0 \rceil$ with $\lceil 1 \rceil$ and $\lceil 3 \rceil$, then with $\lceil 2 \rceil$ and $\lceil 4 \rceil$, we have $\lceil 6 \rceil\ \rho \models \langle y, e_c, \rho' \rangle : (y : \tau) \to \mu$ and $\lceil 7 \rceil$ $\rho \models v' : \tau$.

We then use l-clos of $\lceil 6 \rceil$ with $\lceil 7 \rceil$ and $\lceil 5 \rceil$, we have $\rho, y : v' \models v : \mu$ which, with 3.4.5, is transformed to $\rho \models v : \mu\{e'/y\}$ as required.

Case t-let: By breaking the hypothesis $\lceil 0 \rceil$, we have $\lceil 0_1 \rceil$ $\text{Dom}(\Gamma) = \text{Dom}(\rho)$, $\lceil 0_2 \rceil \vdash \Gamma$ **ok** and $\lceil 0_3 \rceil\ \forall x \in \text{Dom}(\Gamma), \rho \models \rho(x) : \Gamma(x)$.

By breaking the rule t-let, we have $\lceil 1 \rceil\ \Gamma \vdash e : \exists \overline{x}.\tau$, $\lceil 2 \rceil$ $\Gamma, y : \tau, \overline{x} : \overline{\texttt{int}} \vdash e' : \mu$ and $\lceil 3 \rceil\ \overline{x}, y \notin \text{fv}(\mu)$. By breaking the rule d-let, we have $\lceil 4 \rceil\ \rho \vdash e \rightsquigarrow v'$, $\lceil 5 \rceil\ \tau \vdash_{\overline{x}} v' \rightsquigarrow \overline{n}$ and $\lceil 6 \rceil$ $\rho, y : v', \overline{x} : \overline{n} \vdash e' \rightsquigarrow v$.

By induction on $\lceil 0 \rceil$, $\lceil 1 \rceil$ and $\lceil 4 \rceil$, we have $\rho \models v' : \exists \overline{x}.\tau$ which is transformed into $\lceil 7 \rceil\ \rho, \overline{x} : \overline{n} \models v' : \tau$ with 3.7 and $\lceil 5 \rceil$. We also have $\lceil 8 \rceil\ \Gamma, \overline{x} : \overline{\texttt{int}} \vdash \tau$ **ok** with $\Gamma \vdash \exists \overline{x}.\tau$ **ok** from $\lceil 1 \rceil$.

We define $\Gamma' = \Gamma, y : \tau, \overline{x} : \overline{\texttt{int}}$ and $\rho' = \rho, y : v', \overline{x} : \overline{n}$. We then have $\lceil 9_1 \rceil$ $\text{Dom}(\Gamma') = \text{Dom}(\rho')$ from $\lceil 0_1 \rceil$ and definitions, $\lceil 9_2 \rceil \vdash \Gamma'$ **ok** from $\lceil 0_2 \rceil$ and $\lceil 8 \rceil$, and $\lceil 9_3 \rceil\ \forall x \in \text{Dom}(\Gamma'), \rho' \models \rho'(x) : \Gamma'(x)$ from $\lceil 0_3 \rceil$ and $\lceil 7 \rceil$. We can merge them into $\lceil 9 \rceil\ \rho' \models \Gamma'$.

We then have $\rho' \models v : \mu$ by induction on $\lceil 9 \rceil$, $\lceil 2 \rceil$ and $\lceil 6 \rceil$, which, by $\lceil 3 \rceil$, is transformed into $\rho \models v : \mu$ as required.

Case t-let-gen: This is like doing t-lam and t-let at the same time.

By breaking the hypothesis $\lceil 0 \rceil$, we have $\lceil 0_1 \rceil$ $\text{Dom}(\Gamma) = \text{Dom}(\rho)$, $\lceil 0_2 \rceil \vdash \Gamma$ **ok** and $\lceil 0_3 \rceil\ \forall x \in \text{Dom}(\Gamma), \rho \models \rho(x) : \Gamma(x)$.

By breaking the rule t-let-gen, we have $\lceil 1 \rceil\ \Gamma, \overline{z} : \overline{\texttt{int}} \vdash e : \tau$, $\lceil 2 \rceil\ \Gamma, y : \forall \overline{z}.\tau \vdash e' : \mu$ and $\lceil 4 \rceil\ y \notin \text{fv}(\mu)$. And d-let-gen gives us $\lceil 6 \rceil\ \rho, y : \langle \overline{z}, e, \rho \rangle \vdash e' \rightsquigarrow v$.

(t-lam part)

We want to prove the sub-goal $\lceil 7 \rceil\ \rho \models \langle \overline{z}, e, \rho \rangle : \forall \overline{z}.\tau$, which requires to prove $\forall \overline{n}, v_2, (\rho, \overline{z} : \overline{n} \vdash v_2) \implies \rho, \overline{z} : \overline{n} \models v_2 : \tau$.

For given $\overline{n}, v_2$, we suppose $\lceil 8 \rceil\ \rho, \overline{z} : \overline{n} \vdash v_2$.

We define $\Gamma' = \Gamma, \overline{z} : \overline{\texttt{int}}$ and $\rho' = \rho, \overline{z} : \overline{n}$. We then have $\lceil 9_1 \rceil$ $\text{Dom}(\Gamma') = \text{Dom}(\rho')$ from $\lceil 0_1 \rceil$, $\lceil 9_2 \rceil \vdash \Gamma'$ **ok** from $\lceil 0_2 \rceil$ and $\lceil 9_3 \rceil\ \forall x \in \text{Dom}(\Gamma'), \rho' \models \rho'(x) : \Gamma'(x)$ from $\lceil 0_3 \rceil$. We can merge them into $\lceil 9 \rceil\ \rho' \models \Gamma'$.

We then have by induction on $\lceil 9 \rceil$, $\lceil 8 \rceil$ and $\lceil 1 \rceil\ \rho' \models v_2 : \tau$. $\lceil 7 \rceil$ is then obtainned by abstracting $\overline{n}, v_2$.

(t-let part)

With 3.2 of $\lceil 1 \rceil$ and $\lceil 3 \rceil$, we have $\lceil 11 \rceil\ \Gamma \vdash \forall \overline{z}$ **ok**.

We define $\Gamma'' = \Gamma, y : \forall \overline{z}.\tau$ and $\rho'' = \rho, \langle \overline{z}, e, \rho \rangle$. We then have $\lceil 12_1 \rceil$ $\text{Dom}(\Gamma'') = \text{Dom}(\rho'')$ from $\lceil 0_1 \rceil$, $\lceil 12_2 \rceil \vdash \Gamma''$ **ok** from $\lceil 0_2 \rceil$ and $\lceil 11 \rceil$, and $\lceil 12_3 \rceil\ \forall x \in \text{Dom}(\Gamma''), \rho'' \models \rho''(x) : \Gamma''(x)$ from $\lceil 0_3 \rceil$ and $\lceil 7 \rceil$. We can merge them into $\lceil 12 \rceil\ \rho'' \models \Gamma''$.

We then have $\rho'' \models v : \mu$ by induction on $\lceil 12 \rceil$, $\lceil 2 \rceil$ and $\lceil 6 \rceil$, which, by $\lceil 4 \rceil$, is transformed into $\rho \models v : \mu$ as required. □

References

[1] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda-A Functional Language with Dependent Types.. In *TPHOLs*, Vol. 5674. Springer, 73–78. https://doi.org/10.1007/978-3-642-03359-9_6

[2] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[3] Edwin Brady. 2021. Idris 2: Quantitative type theory in practice. *arXiv preprint arXiv:2104.00480* (2021). https://doi.org/10.48550/arXiv.2104.00480

[4] 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).

[5] 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

[6] 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 (jul 2018), 30 pages. https://doi.org/10.1145/3236792

[7] 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

[8] 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.

[9] Troels Henriksen. 2021. Bounds Checking on GPU. *International Journal of Parallel Programming* 49, 6 (2021), 761–775. https://doi.org/10.1007/s10766-021-00703-4

[10] 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

[11] 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

[12] 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

[13] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2019. High-Performance Defunctionalisation in Futhark. In *Trends in Functional Programming*, Michał Pałka and Magnus Myreen (Eds.). Springer International Publishing, Cham, 136–156. https://doi.org/10.1007/978-3-030-18506-0_7

[14] C.Barry Jay. 1995. A semantics for shape. *Science of Computer Programming* 25, 2 (1995), 251–283. https://doi.org/10.1016/0167-6423(95)00015-1 Selected Papers of ESOP'94, the 5th European Symposium on Programming.

[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] Marc Pouzet Jean-Louis Colaço, Baptiste Pauget. 2023. Polymorphic Types with Polynomial Sizes. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Orlando, USA) *(ARRAY 2023)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3589246.3595372

[17] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[18] Lenore Marie Restifo Mullin. 1988. *A Mathematics of Arrays*. Ph. D. Dissertation. USA. https://dl.acm.org/doi/book/10.5555/915213 AAI8914581.

[19] Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-passing Style. *SIGPLAN Lisp Pointers* V, 1 (Jan. 1992), 288–298. https://doi.org/10.1145/141471.141563

[20] Sven-Bodo Scholz. 1994. Single-assignment C — Functional Programming Using Imperative Style. In *6th International Workshop on Implementation of Functional Languages (IFL'94), Norwich, England, UK*, John Glauert (Ed.). University of East Anglia, Norwich, England, UK, 211–2113. https://www.sac-home.org/_media/publications:pdf:sac-overview-norwich-94.pdf

[21] 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. https://doi.org/10.1007/978-3-642-54833-8_3

[22] W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32, 2 (1967), 198–212. https://doi.org/10.2307/2271658

[23] Peter Thiemann and Manuel M. T. Chakravarty. 2013. Agda Meets Accelerate. In *Implementation and Application of Functional Languages (IFL '13)*, Ralf Hinze (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 174–189. https://doi.org/10.1007/978-3-642-41582-1_11

[24] Kai Trojahner and Clemens Grelck. 2009. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming* 78, 7 (2009), 643–664. https://doi.org/10.1016/j.jlap.2009.03.002 The 19th Nordic Workshop on Programming Theory (NWPT 2007).

[25] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. *SIGPLAN Not.* 49, 9 (aug 2014), 269–282. https://doi.org/10.1145/2692915.2628161

[26] 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

Received 2023-05-31; accepted 2023-06-28