Polymorphic Types with Polynomial Sizes

Jean-Louis Colaço
ANSYS
Toulouse, France
Jean-Louis.Colaco@ansys.com

Baptiste Pauget
ANSYS
Toulouse, France
INRIA
Paris, France
Baptiste.Pauget@ansys.com

Marc Pouzet
Ecole normale supérieure – PSL university ; INRIA
Paris, France
Marc.Pouzet@ens.fr

Abstract

This article presents a compile-time analysis for tracking the size of data-structures in a statically typed and strict functional language. This information is valuable for static checking and code generation. Rather than relying on dependent types, we propose a type-system close to that of ML: polymorphism is used to define functions that are generic in types and sizes; both can be inferred. This approach is convenient, in particular for a language used to program critical embedded systems, where sizes are indeed known at compile-time. By using sizes that are multivariate polynomials, we obtain a good compromise between the expressiveness of the size language and its properties (verification, inference).

The article defines a minimal functional language that is sufficient to capture size constraints in types. It presents its dynamic semantics, the type system and inference algorithm. Last, we sketch some practical extensions that matter for a more realistic language.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Polymorphism**; *Recursion*; Semantics; Automated static analysis; *Embedded software*; *Software safety*; Software usability.

Keywords: array programming, type systems

ACM Reference Format:
Jean-Louis Colaço, Baptiste Pauget, and Marc Pouzet. 2023. Polymorphic Types with Polynomial Sizes. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3589246.3595372

1 Introduction

We are interested in the programming, with a high-level language, of certified real-time embedded software submitted to strong safety requirements, such as those found in avionics, railway and automotive (eg, flight control, braking, electrical engine). In this field, the domain-specific programming language Scade [7], is used for more than twenty years. It inherits the principles and style of the synchronous language Lustre [14]. The specific features of these languages are essentially orthogonal to our discussion, but the targetted applications imposes structuring constraints:

(i) Both memory and execution time must be statically bounded. This ensures that a system can run for an arbitrarily long time and meet its deadlines.
(ii) Programs must be certified by independent authorities. This requires a reference specification, extensive testing, and property checking, both for programs and the tools used to generate code.

To this end, the size of any data-structure in Scade must be known statically. While some functions may depend on size parameters, these sizes get ultimately instantiated at compile-time with a concrete value (e.g., an integer). Moreover, the language and its compiler comply with the highest certification standards for critical software (e.g., DO178C, level A of avionics): the generated code can be used without any further validation that the semantics is preserved.

Modern real-time applications combine complex control code (e.g., hierarchical automata) and intensive computation using arrays (e.g., Kalman filters, neural networks, optimization algorithms). Arrays introduce dynamic accesses to memory that must respect array bounds; otherwise, the risk is, at best a stop of the execution, at worst a silent corruption of the memory. Ensuring the access correctness ranges from programmer's responsibility (e.g., in C) to programmer's proof obligations (e.g., in Spark [3]), and include skeptical compilers that generate defensive code in various ways: by throwing exceptions (e.g., OCaml, Ada), by saturating the index value [13] or by adding a default value [7] — two solutions followed in several synchronous compilers (e.g., Heptagon [1], Lustre V6 [2] and Scade [3]). This results in less efficient generated code and the potential introduction of dead code.[4]

[1] https://gitlab.inria.fr/synchrone/heptagon
[2] https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/reactive-toolbox/
[3] https://www.ansys.com/products/embedded-software/ansys-scade-suite
[4] This last point is not without importance: coverage analysis, an activity required for the certification of critical applications, needs justifications for the code that cannot be covered by a test case.

Functional languages have popularized intensional array operations (e.g., map, fold, transpose) [4] which provide predefined access schemes and thus avoid the explicit manipulation of indexes. Their safety only needs size equalities to be solved instead of inequalities (e.g., bound checking), with algorithms that are simpler and more efficient. That turns out well, the data-flow style of Scade favors intensional definitions: the scalar product is written:

```
function dot «n» (u, v: int32^n) returns (d: int32)
  d = (fold $+$ «n») (0, (map $*$ «n») (u, v));
```

However, writing complex array operations as the ones found in signal or image processing or AI is rapidly tedious in Scade for multiple reasons:

(i) The language is explicitly typed. This leads to long and redundant annotations when size expressions grow or multiple size variables are used. E.g., in the example above, the size n appears four times.
(ii) Array primitives (`map`, `fold`, `concat`, ...) are limited to linear relations between sizes. Sampling or filtering are hardly expressed and inefficiently programmed.
(iii) The Scade compiler checks sizes at program elaboration (i.e. instantiation), where sizes get constant values. Error detection is thus late and non-modular.
(iv) All dynamic array accesses are guarded, leading to code with unnecessary conditionals and dead branches.

Contribution. We believe that the possible improvements for the above remarks share a common seed: a type-like knowledge of sizes, available during the whole compilation process, not only after elaboration, would give new perspectives for verification and code generation. In short, the proposed solution is based on the following elements:

(i) A language of sizes made of multivariate polynomials. It provides a practical compromise between formal manipulations and expressiveness.
(ii) An ML-like type system that extends polymorphism to sizes. Sizes are generalized at declarations and instantiated at variable occurrences, like in ML. This allows to handle sizes and types in the unified manner.
(iii) An inference algorithm. Although incomplete, it allows most sizes to be omitted. The size constraints are restricted to vanishing polynomials $P[X] = 0$.

Although being modest, this extension of the type system deeply affects language properties. First, principal typing (as for dependent types), is lost: some terms may receive multiple (incomparable) types. Second, sizes have a *computational content*, i.e. the language semantics is not type erasable. Both points are challenges for the inference algorithm: it should not only produce a well-typed term but it must also ensure that the semantics is independent from inference choices.

This presentation is purposely conducted on a toy functional language that contains the minimal constructs to highlight the main issues. The article is organized as follows: Section 2 gives a general overview of the proposed contribution. The language and its semantics and typing discipline are defined in Section 3. Then, Section 4 details type and size inference and establishes their meta-theoretical properties. Practical extensions are sketched in Section 5. We briefly present the use of static sizes for both verification and compilation in Section 6. We discuss related works in Section 7 and conclude in Section 8.

An extended version of this type system is implemented in a prototype of compiler for a synchronous language with arrays. A type checker for a simpler, ML-like language with the proposed syntax is available. [5] The appendices can be found in the Supplemental Material at the ACM DL.

2 Overview

For brevity, we introduce a core language $\mathcal{L}^\eta$ that contains the minimal constructs required to introduce a notion of sizes into types. In particular, it has no primitive notion of arrays: they are considered as functions on a bounded domain. This section gives an informal insight of $\mathcal{L}^\eta$, its type system and type inference through simple examples.

$\mathcal{L}^\eta$ is equipped with a separate language for sizes, namely sizes and expressions are distinct syntactical objects. This size language is made of multivariate polynomials. Sizes (ranged over by $\eta, \ldots$) and their variables (ranged over by $\iota, \kappa, \delta, \ldots$) are handled in a similar way to types ($\tau, \ldots$) and type variables ($\alpha, \beta, \gamma, \ldots$).

Intensional Arrays and Size Consistency. In numerous programming languages intended for scientific computations such as Sisal [10], explicit index manipulations are replaced by operators acting on arrays called *combinators* [17, 20]. This style benefits especially to functional and data-flow programming languages by allowing to write array definitions with single expressions. Array combinators provide predefined access patterns that are always correct, avoiding at the same time the need for runtime checks. However, some of these primitives still require array sizes to coincide. To enforce such properties by type checking, sizes need to be expressed in types. The point-wise function application (`map`), its binary variant (`map2`) and the array reduction (`fold`), three operators that are available in Scade, are given the following type schemes in the proposed language $\mathcal{L}^\eta$:

```
val map  : ∀ι·α·β. (α → β) → <ι> → [ι]α → [ι]β
val fold : ∀ι·α·β. (α → β → α) → <ι> → α → [ι]β → α
val map2 : ∀ι·α·β·γ. (α → β → γ) → <ι> → [ι]α → [ι]β → [ι]γ
```

Given a polynomial size η, the type $<\eta>$ (read *value of size* η) denotes a *refinement* of the integer base type, i.e. $\{x : \texttt{int} \mid x = \eta\}$. Actually, this is a singleton type. Here, the second argument of each function thus allows to constrain the value of the size variable ι. Using the notation of Futhark [17], $[\eta]\tau$ is the type of arrays with length η

[5] https://gitlab.inria.fr/bpauget/array-2023

and elements of type τ. Used as an expression, the syntax <η> also designates the only value of type <η>, thus the partial application —map f <9>— can only be given an array of length 9. These signatures highlight polymorphism that act both on type variables (α,β,γ) and size variables (ι). Using them, the scalar product is expressed as:

```
let dot = λu. λv. fold (+) <_> 0 (map2 (*) <_> u v)
                                   [ ∀ι. [ι]int → [ι]int → int ]
```

The second line renders the inferred type scheme. In the definition of dot, the size values —<_>— are omitted for both iterators: they are inferred. The built type scheme forces the sizes of u and v to coincide. The size variable ι cannot be directly constrained: no arguments have type <ι>. Thus, ι will be deduced from the size of u and v. Let assume the definition of a primitive window defining a sliding window of size κ with step 1:

```
val window : ∀ι·κ·α. <κ> → [ι + κ − 1]α → [ι][κ]α
```

This function builds a matrix whose rows are slices of length κ of the input array, starting at the element given by the row index. For example —window <3> $[0, 1, 2, 3, 4]$— produces the matrix $[[0, 1, 2], [1, 2, 3], [2, 3, 4]]$. The size $\iota + \kappa - 1$ encodes the relation between input and output array sizes so that the former is fully read. Filtering data with a kernel K of size κ is a common signal processing operation. It is expressed with a discrete convolution: $(K * I)_i = \sum_{j=0}^{\kappa-1} K_j \cdot I_{i+j}$. This uni-dimensional filter may be defined as:

```
let convolution = λk. λi. map (dot k) <_> (window <_> i)
                   [ ∀ι·κ. [κ]int → [ι + κ − 1]int → [ι]int ]
```

Here as well, the inference is able to determine the missing sizes (and types), making the kernel size coincide with slices length. Inference derives the above type scheme for this declaration. Note that, by a change of variables, it is equivalent to $\forall\iota\cdot\kappa.$ $[\kappa]$int $\rightarrow$ $[\iota]$int $\rightarrow$ $[\iota - \kappa + 1]$int.

Extensional Arrays and Bounds Propagation. Arrays are not primitive constructs: the type $[\eta]\tau$ is a shortcut for $[\eta] \rightarrow \tau$ where $[\eta]$ (read *index of size* η) is a second refinement of type int denoting non-negative integers strictly lesser than η: $\{x : \text{int} \mid 0 \leq x < \eta\}$. Although not realistic for compilation, this simplifies the formalism. Using this refinement, the map2 iterator is expressible in $\mathcal{L}^\eta$:

```
let map2 = λf. λn:<‘ι>. λu. λv. λi:[‘ι]. f (u i) (v i)
        [ ∀ι·α·β·γ. (α → β → γ) → <ι> → [ι]α → [ι]β → [ι]γ ]
```

It defines an 'array', i.e. a function with a bounded domain, whose content is obtained by applying f to u and v elements. Array accesses are denoted by the applications —$(u\ i)$; $(v\ i)$— that respect bounds by construction. The second argument —n— of map2 is unused, but the types annotations <‘ι> and [‘ι], where ‘ι is an *anonymous* size variable alike OCaml's ones, force n to be the size of index i.

The index type only allows to propagate known bounds. Notably, no arithmetic operations are defined for indexes. Values of type $[\eta]$ are obtained by calling functions with a bounded codomain, e.g., the modulo, whose type scheme is $\forall\iota.$ int $\rightarrow$ <ι> $\rightarrow$ $[\iota]$. Although elementary, this refined type allows to separate bound checking from array accesses.

The Ghost Size Issue. In the previous examples, all unspecified sizes were deducible from the types. This is not always that simple. With the annotation [_], the cst function below defines a constant array with an arbitrary size.

```
let cst = λc. λi:[_]. c                      [ ∀ι·α. α → [ι]α ]
let even = fold (+) <_> 0 (cst 2)    (Error: Unconstrained size)
```

In the declaration of even, summing the elements of cst 2 without specifying fold's size leads to an ambiguous value since this size is unconstrained. This must be rejected.

Contrary to types in ML like languages where semantics is type-erasable[6], sizes have a *computational content*: they may determine the semantics of expressions. Inference must thus ensure that the semantics of the reconstructed term was already specified in the source. We formalize this property in subsection 4.5: when type inference succeeds, all well typed annotated versions of the source program evaluate to the same result. Our even example becomes unambiguous by adding an argument:

```
let even = λn. fold (+) n 0 (cst 2)           [ ∀ι. <ι> → int ]
```

3 A Typed Core Functional Language with Size Polymorphism

The type system for our language aims at expressing as many relations between sizes as possible while remaining decidable and largely implicit. It combines two widely studied type systems traits: (i) a restricted form of *refinement types*, as defined in Xi and Pfenning [38], and (ii) the ubiquitous *let-polymorphism* of Milner [27] extended to sizes.

Polynomial sizes lead to constraints that cannot be resolved formally. Nonetheless, the context of Scade ensures that all sizes get ultimately known statically: once elaboration is done, size checking is trivial but late and non modular. We pursue an earlier and modular size checking, that rely on the specialization as a fallback mechanism only.

For clarity, the tightest possible language $\mathcal{L}^\eta$ is used: a core ML (λ-calculus with let bindings) augmented with a few constructs. We propose some extensions that are useful for a realistic language in Section 5.

3.1 Syntax and Semantics

The syntax of $\mathcal{L}^\eta$ is summed-up in Figure 1. It is explicitly typed i.e. type annotations are part of expressions. In the subsequent, n denotes an integer. To emphasize on their

[6] Unless advanced features, such as type classes, are considered.

η	$::=$		***Sizes***
	$\mid$	$\iota,\ \kappa,\ \delta$	variable
	$\mid$	n	constant
	$\mid$	$\eta\ +\ \eta$	sum
	$\mid$	$\eta\ *\ \eta$	product
τ	$::=$		***Types***
	$\mid$	$\alpha,\ \beta,\ \gamma$	variable
	$\mid$	$\tau \to \tau$	function
	$\mid$	`int`	integer
	$\mid$	$<\eta>$	singleton
	$\mid$	$[\eta]$	interval

V	$::=$	$\varepsilon \mid \iota \cdot V \mid \alpha \cdot V$	***Generalization***
S	$::=$	$\varepsilon \mid \eta \cdot S \mid \tau \cdot S$	***Instantiation***
e	$::=$		***Expressions***
	$\mid$	${}^{S}x$	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau .\ e$	abstraction*
	$\mid$	**let** d **in** e	local binding
	$\mid$	$e \triangleright \tau$	coercion
	$\mid$	$<\eta>$	size value
	$\mid$	n	integer*
d	$::=$	$x_V : \tau = e$	***Declarations***

Figure 1. Syntax of $\mathcal{L}^\eta$. Note that the syntax $<\eta>$ is overloaded: it denotes both the singleton type and its unique value. The values used in the semantics are marked with *.

similarities, sizes, types and their variables are designated by Greek letters whereas the Latin ones will be dedicated to terms and program variables.

Name-spaces, Free Variables. Because sizes, types and expressions are syntactically separated, their variables are taken in distinct name-spaces, respectively denoted $\mathcal{V}_\eta$, $\mathcal{V}_\tau$ and $\mathcal{V}_e$, allowing for name reuse without masking. In the formalization, these sets are considered disjointed. Given a syntactical object o, the set $FV(o) \in \mathcal{V}_\eta \cup \mathcal{V}_\tau \cup \mathcal{V}_e$ of *free variables* contains the variables that are not bound by the rules: (i) abstraction —$\lambda x : \tau.\ e$— binds x in e; (ii) local binding —**let** $x_V : \tau = e$ **in** e'— binds x in e' and variables V in τ and e. *Closed* objects are the ones with no free variables.

Sizes and Types. The size language is made of multivariate polynomials with integer coefficients: $\mathbb{Z}[\mathcal{V}_\eta]$. The main benefit of this restricted class of arithmetic expressions lies in the existence of a normal form: a weighted sum of products of variables. This allows for symbolic comparison of sizes that are structurally different (e.g., $(\iota - 1)^2 - 1\ =\ \iota * (\iota - 2)$).

Besides functions, the type language contains a single constructor `int`, along with two refinements, as defined by Xi and Pfenning [38]: (i) the type $<\eta>$ (read *value of size* η) denotes the singleton $\{\overline{\eta}\}$ and (ii) $[\eta]$ (read *index of size* η) represents the interval $[\![0, \overline{\eta} - 1]\!]$, where $\overline{\eta}$ is the value of η, depending on the valuation of size variables. In the syntax refinement types, they are respectively expressed as $\{x : \texttt{int} \mid x = \overline{\eta}\}$ and $\{x : \texttt{int} \mid 0 \le x < \overline{\eta}\}$

Polymorphism. Types, including polymorphism, are explicit in $\mathcal{L}^\eta$: variables —${}^{S}x$— are instantiated with a list S of sizes and types while local bindings —**let** $x_V : \tau = e$ **in** e'— declare the list[7] V of size and type variables that are generalized. We shift away from the standard notation —**let** $x : \forall S.\ \tau\ =\ e$ **in** e— to emphasize on generalized variables' scope: their are bound in both type τ and expression e.

[7] The use of lists simplifies the association of generalized variables with their instantiations

Expressions. Integers occur in two ways: —n— denotes immediate values while —$<\eta>$— stands for the only value of type $<\eta>$, that is defined by size variable valuation.

Last, the coercion —$e \triangleright \tau$— represents an explicit type constraint. Because of refined types, it plays a central role in the definition of the semantics (see subsection 3.2).

Arrays. $\mathcal{L}^\eta$ has no support for array manipulation, neither in types nor in expressions. For typing purposes, arrays are essentially functions on a bounded domain: that is the role of the index refinement. To make examples more intuitive, we will use the notation of Futhark $[\eta]\tau$ [17] as a shorthand for the type $[\eta] \to \tau$. Allowing sub-typing from functions to arrays seems irrelevant, both for clarity and compilation perspectives. However, the typing issues that arise from refinement types would still occur with a dedicated language support for arrays.

3.2 Semantics

The big-step semantics of $\mathcal{L}^\eta$ —$e \rightsquigarrow v$— associates to some closed expressions a *value*. As defined in Figure 1, values are either integers or abstractions. The deduction rules are detailed in Figure 2. They are syntax-directed, thus $\mathcal{L}^\eta$ semantics is deterministic.

Substitutions. Substitutions (ranged over by $\rho, \ldots$) are defined for each syntactical class and variable/element pairs. Their application is written $e\{\rho\}$. Explicit ones are uniformly denoted $\cdot/\cdot$. Thus, $e\{\eta/\iota\}$ is the substitution in expression e of the occurrences of the size variable ι by the size η, including in sizes and types contained in e. This notation is naturally extended to generalization and instantiation lists, assuming they are compatible, i.e. that each size variable is substituted with a size and likewise for types.

When evaluating let bindings (rule E-Let), each occurrence of the defined variable —${}^{S}x$— get substituted with its coerced definition in which generalized size and type variables have been instantiated — $(e \triangleright \tau)\{S/V\}$.

Semantics of closed Expressions $\boxed{e \rightsquigarrow v}$

$$\text{E-SIZE}\ \frac{}{\texttt{<}n\texttt{>} \rightsquigarrow n} \qquad \text{E-APP}\ \frac{e_1 \rightsquigarrow \lambda x\!:\!\tau.\ e \qquad e_2 \triangleright \tau \rightsquigarrow v \qquad e\{v/^{\varepsilon}x\} \rightsquigarrow v'}{e_1\ e_2 \rightsquigarrow v'}$$

$$\text{E-COERCE}\ \frac{e \rightsquigarrow v \qquad v \triangleright \tau \rightsquigarrow v'}{e \triangleright \tau \rightsquigarrow v'} \qquad \text{E-LET}\ \frac{e'\{(e \triangleright \tau)\{S/V\}/^{S}x\} \rightsquigarrow v}{\texttt{let}\ x_V\!:\!\tau = e\ \texttt{in}\ e' \rightsquigarrow v}$$

Semantics of Coercions $\boxed{v \triangleright \tau \rightsquigarrow v'}$

$$\text{C-SIZE}\ \frac{n' = n}{n' \triangleright \texttt{<}n\texttt{>} \rightsquigarrow n'} \qquad \text{C-INDEX}\ \frac{n' \in [\![0, n-1]\!]}{n' \triangleright \texttt{[}n\texttt{]} \rightsquigarrow n'}$$

$$\text{C-INT}\ \frac{}{n \triangleright \texttt{int} \rightsquigarrow n} \qquad \text{C-FUN}\ \frac{v = \lambda x\!:\!\tau.\ e}{v \triangleright \tau_1 \to \tau_2 \rightsquigarrow \lambda x\!:\!\tau_1.\ v\ x \triangleright \tau_2}$$

Figure 2. Semantics of $\mathcal{L}^\eta$.

Refinements and Coercions. The semantics of $\mathcal{L}^\eta$ is not type-erasable. This obviously transpires in the rule E-SIZE that extracts a value from a size. Moreover, refinements discriminate between values of the same *shape* (or base type) and they must be checked in several places. For instance, the term —$(\lambda x : \texttt{[4]}.\ e)\ 8$— should not be reduced further since the argument —8— is not a value of the expected type: $\texttt{[4]}$. More generally, for any substitution of a term variable, the substituting value must fulfill the substituted variable refinement. Therefore, the E-APP and E-LET rules insert coercions; hence the need of an explicit coercion construction in expressions.

For integer refinements, coercions check that the refinement is fulfilled (rules C-INT, C-SIZE and C-INDEX). Similarly to λ^H [11], function coercions reduce into delayed coercions on argument and result (rule C-FUN), that will be evaluated upon application. The coercion on argument is actually inserted by the evaluation of the introduced application: $v\ x$.

Type Independence. Although $\mathcal{L}^\eta$ semantics is not type-erasable, only sizes have computational content i.e. the observational semantics of an expression does not depend on the valuation of its type variables. Changing types (hence possible refinements) only restrict semantics domain.

Definition 3.1 (Observational equivalence)**.** Two closed expressions e_1 and e_2, are *observationally equivalent* —$e_1 \equiv e_2$— if and only if, for any closed expressions $a_1, \ldots, a_k$ and integers n_1, n_2:

$$\left\{\begin{array}{l} e_1\ a_1\ \ldots\ a_k \rightsquigarrow n_1 \\ e_2\ a_1\ \ldots\ a_k \rightsquigarrow n_2 \end{array}\right. \implies n_1 = n_2$$

Expressions for which no such common evaluation environment exist are considered equivalent. Used along with typing assumptions to rule out silly cases (e.g., expressions have different types), it allows to compare functions on their common domain.

Definition 3.2 (Equality modulo types)**.** Two expressions e_1 and e_2 are *equal modulo types* —$e_1 \approx_\tau e_2$— if and only if it exists an expression e, free type variables $\overline{\alpha}$ of e and types $\overline{\tau_1}, \overline{\tau_2}$ such that the following syntactical equalities hold:

$$e_1 = e\{\overline{\tau_1}/\overline{\alpha}\} \ \wedge\ e_2 = e\{\overline{\tau_2}/\overline{\alpha}\}$$

Equivalence modulo type is preserved by the semantics:

Theorem 3.3 (Type independence)**.** *Given two closed terms e_1, e_2 such that $e_1 \approx_\tau e_2$, then*

$$\forall\ v_1\ v_2,\ \left\{\begin{array}{l} e_1 \rightsquigarrow v_1 \\ e_2 \rightsquigarrow v_2 \end{array}\right. \implies v_1 \approx_\tau v_2$$

Proof. This invariant holds across semantics rules thanks to the following observations (detailed in Appendix A):

- Size substitutions cannot capture types. The instances of E-SIZE rule are thus equals and yield the same value.
- Selecting between C-FUN and other coercion's rules depends only on value shape. □

Corollary 3.4 (Type observable independence)**.** *Expressions that are equal modulo types are observationally equivalent:*

$$\forall\ e_1\ e_2,\ e_1 \approx_\tau e_2 \implies e_1 \equiv e_2$$

Proof. Given e_1, e_2 such that $e_1 \approx_\tau e_2$, and closed expressions $a_1, \ldots, a_k$, we immediately have $e_1\ a_1\ \ldots\ a_k \approx_\tau e_2\ a_1\ \ldots\ a_k$. Observational equivalence follows because equality modulo types for integers implies equality: if both applications reduce, and one of the result is an integer, then the other is identical. Thus expressions coincide on the intersection of their domain (that might be empty). □

3.3 Typing

A type discipline, based on the one of Hindley and Milner [18], filters the terms for whom a semantics exists, i.e. expressions that may be reduced to a value.

Environment. Expressions are typed in an environment Γ defined as a pair (Γ_v, Γ_e) where $\Gamma_v \subset \mathcal{V}_\eta \cup \mathcal{V}_\tau$ is the set of bound size and type variables and Γ_e is a partial map from term variables to type schemes —$\sigma ::= \forall V.\ \tau$—. In the following, terms are supposed to be named so that no clashes occur. The environment is thus unordered and insertions —$\Gamma,\ x\!:\!\forall V.\ \tau,\ V$ — assume that added variables —x and V— are unbound in Γ.

Judgments. The typing judgment —$\Gamma \vdash e : \tau$— reads 'in the environment Γ, the expression e has type τ'. This relation implicitly assumes that τ and e are *well-formed*, i.e. that their free variables are bound in Γ. It is defined alongside the sub-typing relation —$\tau_1 <: \tau_2$— in Figure 3.

It is worth mentioning that type equality, used in S-REFL rule among others, requires a syntactical identity between the sizes appearing in refinements. For instance, types $\texttt{[}\iota\texttt{]}$ and $\texttt{[}2 - \iota\texttt{]}$ are considered different even though they yield equal types when instantiated with $\iota = 1$.

Expressions Typing $\boxed{\Gamma \vdash e : \tau}$

$$\text{T-VAR}\ \frac{\Gamma(x) = \forall V.\ \tau}{\Gamma \vdash {}^{S}x : \tau\{S/V\}} \qquad \text{T-SUBTYPE}\ \frac{\Gamma \vdash e : \tau \qquad \tau <: \tau'}{\Gamma \vdash e : \tau'} \qquad \text{T-COERCE}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \triangleright \tau : \tau}$$

$$\text{T-ABS}\ \frac{\Gamma,\ x{:}\forall \varepsilon.\ \tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.\ e : \tau \rightarrow \tau'} \qquad \text{T-APP}\ \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\text{T-LET}\ \frac{\Gamma,\ V \vdash e : \tau \qquad \Gamma,\ x{:}\forall V.\ \tau \vdash e' : \tau'}{\Gamma \vdash \mathtt{let}\ x_V{:}\tau = e\ \mathtt{in}\ e' : \tau'} \qquad \text{T-SIZE}\ \frac{}{\Gamma \vdash \mathtt{<}\eta\mathtt{>} : \mathtt{<}\eta\mathtt{>}} \qquad \text{T-ISIZE}\ \frac{}{\Gamma \vdash n : \mathtt{<}n\mathtt{>}} \qquad \text{T-IINDEX}\ \frac{0 \leq n < p}{\Gamma \vdash n : \mathtt{[}p\mathtt{]}}$$

Sub-typing $\boxed{\tau_1 <: \tau_2}$

$$\text{S-SIZE}\ \frac{}{\mathtt{<}\eta\mathtt{>} <: \mathtt{int}} \qquad \text{S-INDEX}\ \frac{}{\mathtt{[}\eta\mathtt{]} <: \mathtt{int}}$$

$$\text{S-REFL}\ \frac{}{\tau <: \tau} \qquad \text{S-FUN}\ \frac{\tau_2 <: \tau_1 \qquad \tau_1' <: \tau_2'}{\tau_1 \rightarrow \tau_1' <: \tau_2 \rightarrow \tau_2'}$$

Figure 3. Type system for $\mathcal{L}^\eta$, non syntax directed

Refinements and Sub-typing. General dependent type systems such as DML [38] provide rich sub-typing relations based on refinement implication, at the cost of static checking undecidability. For that reason as well as inference perspectives, sub-typing is restricted to inserting or dropping refinements (with respect to the variance). Thus, the relations $\mathtt{[}\eta\mathtt{]} <: \mathtt{int}$ and $\mathtt{int} \rightarrow \alpha <: \mathtt{[}\eta\mathtt{]}\alpha$ are valid, whereas the semantically correct relation $\mathtt{[}\iota\mathtt{]} <: \mathtt{[}\iota + 1\mathtt{]}$ is invalid. This flat order between refined types, illustrated bellow, is the key restriction to keep type checking decidable: correction only relies on size equalities, instead of general inequalities on polynomials.

$$\begin{tikzcd}
 & & \mathtt{int} \arrow[dll, dash] \arrow[dl, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
\mathtt{<}\eta\mathtt{>} & \mathtt{<}\eta'\mathtt{>} & \cdots & \mathtt{[}\eta'\mathtt{]} & \mathtt{[}\eta\mathtt{]}
\end{tikzcd}$$

The Types of Constants. The only immediate values we consider here are integers. The rules T-ISIZE and T-IINDEX allows any semantically correct refined type whose size is an immediate value. This is needed to state *type preservation*. Otherwise the expression $\mathtt{<}n\mathtt{>}$ of type $\mathtt{<}n\mathtt{>}$ would reduce to n, that could not be given the same type. In λ^H, Flanagan [11] faces a similar situation: constant are given singleton types, so that any possible refinements may be derived using sub-typing. Because our sub-typing lattice is not closed by intersection, we directly provide all the possible types.

Preservation and Soundness. This type system enjoys both preservation and soundness: types are preserved by reduction and well-typed terms have a semantics. Formally:

Theorem 3.5 (Preservation and soundness)**.**
Given an expression e and a type τ such that $\vdash e : \tau$.
Then there exists a value v such that $e \rightsquigarrow v$. *(Soundness)*
Moreover, $\vdash v : \tau$. *(Preservation)*

Proof. The generic construction for big step semantics set up by Dagnino et al. [8] allows to establish these results from three local properties on the type system and the semantics (see Appendix B):local preservation, ∃-progress and ∀-progress. To establish them, a key step is the normalization of typing derivations, that confines the instances of T-SUBTYPE rule to specific premises. □

4 Inference

Although type annotations might be helpful for documentation purposes (e.g., in interfaces), they tend to obfuscate programs as size expressions get larger. They should be inferred. However, pursuing a full and complete type inference as the HM type discipline enjoys [18] is vain: the size language, that allows non-linear arithmetic expressions, will surely cause unsolvable constraints. Despite this, the size relations that occurs in data-intensive applications are often simple, giving the opportunity to omit most of them. Figure 4 gives implicitly typed definitions of simple linear algebra operation and their inferred type.

One point must be carefully handled: $\mathcal{L}^\eta$ semantic is specified over *closed typed* terms. Inference must ensure that the semantics of reconstructed terms is fully defined by implicitly typed ones. The *ghost size* issue sketched in Section 2 is crucial here: unconstrained size variables should not get defined during reconstruction. As a result, inference must ensure that no unnecessary size relations are introduced.

4.1 Implicitly Typed $\mathcal{L}^\eta$

As an implicitly typed language, a slight variation of $\mathcal{L}^\eta$ is used: generalization and instantiation places, i.e. V and S in $\mathcal{L}^\eta$ syntax, are omitted. Contrary to polymorphism, type annotations are still present in implicitly typed terms, but they might contain size and type variables that are unbound. Given a term e, the inference builds a *completion* of e by providing definitions for polymorphism places, i.e. a list of size and type variables that are generalized or used for instantiation, alongside a substitution of unbound size and type variables. In examples, place-holders (_) stand for fresh size or type variables.

4.2 Algorithm

Size equality constraints amount to vanishing polynomials. Unlike types, whose unification is structural, these constraints cannot be solved easily. For that reason, instead of building a substitution on the fly as done by Algorithm $\mathcal{W}$ [27], sub-typing constraints and unbound size and type variables are collected by a term traversal and the resulting system is solved at generalization places (i.e. **let**), in the hope of using the simplest constraints to simplify the most complex ones. In the context of sub-typing, similar

```
let dot = λu. λv. fold (+) <_> 0 (map2 (*) <_> u v)            [ ∀ι. [ι]int → [ι]int → int ]
let mat_vec = λa. λv. map (dot u) <_> (transpose a)       [ ∀ι·κ. [ι][κ]int → [ι]int → [κ]int ]
let mat_mat = λa. λb. map (mat_vec b) <_> a           [ ∀ι·κ·δ. [ι][κ]int → [κ][δ]int → [ι][δ]int ]
```

Figure 4. Usual linear algebra primitives defined with iterators

algorithms were proposed, e.g., by Aiken and Wimmers [1], where constraints are simplified at generalization points. The constraint collecting algorithm is explained in details in Appendix C.

Let bindings introduce generalization: once the definition has been traversed, sub-typing constraints are solved. As for simple ML, the remaining type and size variables that do not appear in the environment are generalized. Moreover, two extra checks are performed on the generalized variables:

1. They should not appear in remaining constraints.
2. They must appear in declaration's type.

The former allows to keep simple polymorphism, while the latter detects unconstrained variables. This last check is crucial since term's semantics depend on sizes: the Section 2 gives an example of such ambiguous term:

```
let even = fold (+) <_> 0 (cst 2)      (Error: Unconstrained size)
```

4.3 Principal Typing

Before presenting the constraint resolution strategy, let us focus on a thorn in our side: this type system does not enjoy principal types, i.e. some declarations do not have a most general type scheme. Comparison of type schemes is defined by the *subsumption* relation presented by Jones et al. [22]. Informally $\sigma_1 \preccurlyeq \sigma_2$ if and only if any instance of σ_2 may be obtained by instantiating σ_1 and using sub-typing. σ_1 is then *more general* than σ_2. It naturally defines a notion of equivalence, that amounts for simple ML types (without sizes), to a renaming of type variables. Because size equality is not structural, this relation widens here: the uni-dimensional convolution defined in Section 2 may be given the following type schemes:

```
val convolution : ∀ι·κ. [κ]int → [ι]int → [ι − κ + 1]int
val convolution : ∀ι·κ. [κ]int → [ι + κ − 1]int → [ι]int
```

Any instance of the first is an instance of the second, and reciprocally. More importantly, some terms may be given multiple type schemes that have no common generalization; this must be carefully handled by the inference. There are two reasons for this:

1. Polynomial Sizes. Allowing more than linear expressions for sizes surely causes constraints with multiple solutions. Given a function `split`, declared below, that transforms a 1-dimensional array into a 2-dimensional one, its application to an 'array' of size 4 raises several possible types, corresponding to different semantics:

```
val split : ∀ι·κ·α. [ι ∗ κ]α → [ι][κ]α
```

```
                                      ⎧ [1][4]int
let mat = split (λi:[4]. 0)           ⎨ [2][2]int
                                      ⎩ [4][1]int
```

In such a situation, the underlying constraint ($\iota * \kappa - 4 = 0$) will not be solved (see section 4.4), and inference will fail, asking for more annotations.

2. Sub-typing and Simple Polymorphism. Because polymorphism is unconstrained, all the refinements are selected at definition. The `slope` function below computes the ratio of images' difference over arguments' difference, assuming suitable arithmetic operators defined over integers.

```
let slope = λf. λi. λj. (f i - f j) / (i - j)
```

The subtlety comes from the simultaneous applications of f to i and j: should f's domain and both argument share the same refinement ? Indeed, possible type schemes include:

$$\begin{array}{llr}
\forall\iota. & (\texttt{<}\iota\texttt{>} \to \texttt{int}) \to \texttt{<}\iota\texttt{>} \to \texttt{<}\iota\texttt{>} \to \texttt{int} & (\sigma_s^s) \\
\forall\iota\cdot\kappa. & (\texttt{int} \to \texttt{int}) \to \texttt{<}\iota\texttt{>} \to \texttt{<}\kappa\texttt{>} \to \texttt{int} & (\sigma_s^b) \\
 & (\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int} & (\sigma_b^b) \\
\forall\iota\cdot\kappa. & (\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int} & (\sigma_i^b) \\
\forall\iota. & ([\iota] \to \texttt{int}) \to [\iota] \to [\iota] \to \texttt{int} & (\sigma_i^i)
\end{array}$$

Among them, $\sigma_b^b \preccurlyeq \sigma_b^s$ and $\sigma_b^b \preccurlyeq \sigma_b^i$. Others are incompatible pair-wise (denoted $\not\approx$) for multiple reasons: refinements...

1. are incompatible: $\sigma_s^s \not\approx \sigma_i^i$; $\sigma_s^b \not\approx \sigma_i^b$; $\sigma_s^b \not\approx \sigma_i^i$; $\sigma_s^s \not\approx \sigma_i^b$
2. appear covariant and contravariant: $\sigma_s^s \not\approx \sigma_b^b$; $\sigma_i^i \not\approx \sigma_b^b$
3. impose extra size constraints: $\sigma_s^s \not\approx \sigma_s^b$; $\sigma_i^i \not\approx \sigma_i^b$.

Constrained Polymorphism. Coupling sub-typing and simple polymorphism is unusual. The general theory proposed by Aiken and Wimmers [1] provides constrained types schemes. Shrinking the constraint set at generalization point is then the key to avoid an exponential blow-up of constraints [32]. Such systems enjoy the principal types property. In this context, the function `slope` would be given the type scheme:

$$\forall\alpha\cdot\beta\cdot\gamma \mid \alpha <: \texttt{int} \wedge \beta <: \alpha \wedge \gamma <: \alpha.\ (\alpha \to \texttt{int}) \to \beta \to \gamma \to \texttt{int}$$

However, modularity would be sacrificed here, by deferring size constraints resolution to monomorphic instantiations. Coupled with the loss of readability of such types, this is the main reason for keeping simple polymorphism.

Inference and Semantics. This issue about principal type is all the more crucial because our semantics is not type erasable. Sizes have computational contents in our language. For that reason, inference should ensure that no sizes have been arbitrarily defined. We formalize this in subsection 4.5.

```
let slope = λf:[_] → _. λi: _ . λj: _ . (f i - f j) / (i - j)        [∀ι. ([ι] → int) → [ι] → [ι] → int]
let slope = λf: _  → _. λi:[ι]. λj:[ι]. (f i - f j) / (i - j)        [∀ι. ([ι] → int) → [ι] → [ι] → int]
let slope = λf: _  → _. λi:[_]. λj:[_]. (f i - f j) / (i - j)      [∀ι·κ. (int → int) → [ι] → [κ] → int]
```

Figure 5. Different type annotations in the `slope` example lead to different type schemes.

4.4 Constraint Solving

Solving the constraint system aims at extracting from the set of sub-typing constraints a *most general unifier*, i.e. a necessary substitution of the free variables. This is achieved gradually: (i) types (without refinements) are inferred using structural unification; (ii) necessary refinements of type `int` are selected; (iii) sizes constraints are solved; (iv) refinements are propagated. Similar stratification has been previously used for inference in extended type systems [24, 34, 37]. However these steps are utterly entangled in our proposal: instead of separating phases across multiple passes, types, refinements and sizes get partially defined at each solving point (**`let`**), allowing an easier handling of polymorphism than it would be possible with disconnected inference passes.

To illustrate our overview of the solving process, three slightly modified version of the `slope` example used previously are defined in Figure 5: some annotations are added, constraining the refinements at different places.

(i) Types. To begin with, refinements are ignored to build *simple types* that will be made precise in the subsequent phases. By replacing every refinements with `int`, sub-typing relations are turned into equalities. They are solved using structural unification, failing in the usual modalities (e.g., top-level type constructor inequality, cyclic types). It generates a *most general unifier* [27]. At that point, Each declaration of `slope` get type `(int → int) → int → int → int`.

(ii) Refinements. The integer types previously derived may now be refined: each occurrence of `int` in the substitution are replaced by fresh type variables. Sub-typing constraints are distributed with the S-Fun rule (the usual variance rule), leading to simple constraints containing variables and refined types. The ones of the form α <: `[_]`, α <: `<_>` and `int` <: α define variable α's refinement while the unsolvable constraints such as `int` <: `[_]` lead to errors that are reported to the programmer. Refinements are not propagated further: this is postponed after size resolution, since adding refinements may introduce constraints between sizes that would otherwise be unrelated. Unconstrained variables get thus substituted with `int`.

In our example, the first definition of the `slope` function get type `([_] → int) → [_] → [_] → int` while two others get `(int → int) → [_] → [_] → int`.

(iii) Sizes. Then, sub-typing constraints are distributed again, extracting size equalities, i.e. vanishing polynomials $\eta_1 - \eta_2 = 0$ for the constraints of the form `<`η_1`>` <: `<`η_2`>` or `[`η_1`]` <: `[`η_2`]`. The resulting polynomial system C^η is solved, by deriving a *most general substitution*:

Definition 4.1 (Most general substitution)**.** Given a constraint set C, a substitution ρ is *most general* if and only if for any substitution ρ', such that $\vdash C\{\rho'\}$, then there exists a substitution ρ'' such that $\rho' = \rho \circ \rho''$.

For now, we have implemented a simple resolution strategy that eliminates *isolated variables*, i.e. substituting η for ι when a constraint $\iota - \eta = 0$ exists. The resulting substitution is immediately a most general one. This task could be delegated to an external solver, but this is unpractical in the context of safety-critical sofware for certification purposes[8]. Moreover, this elementary strategy works for most of the size constraints we encountered. Adding some annotations helps for the remaining cases. At this point, the three versions of `slope` get respectively types:

1. `([ι] → int) → [ι] → [ι] → int`
2. `(int → int) → [ι] → [ι] → int`
3. `(int → int) → [ι] → [κ] → int`

In particular, ι and κ sizes are not unified in the last declaration, because sub-typing is at each application.

(iv) Propagation. Last, refinements are propagated. This aims at making types more accurate. Among the type variables introduced during refinement inference, the ones whose lower bound only contains a unique type `[ι]` or `<ι>`, are defined accordingly.

In 2., refinements are now equal: they are propagated, yielding type `([ι] → int) → [ι] → [ι] → int`. Conversely, they differ in the third version: f domain cannot be refined.

4.5 Inference Properties

The expressiveness of the size language allows no hope for a complete inference. Nevertheless, it is sound and we expect inference to be *non-specializing*, i.e. that it rejects any terms with ambiguous semantics.

Theorem 4.2 (Inference soundness)**.** *Given an implicitly typed expression, if inference succeeds, the reconstructed term is well-typed.*

Proof sketch. The detailed proof is available in Appendix C,alongside a formalization of the inference algorithm. It established the following invariant: for each sub-term and a substitution that solves the constraints gathered by the algorithm, it exists a type derivation for the substituted term, in the same environment. □

[8] Even though the compiler is not embedded, it must fulfill the highest certification level, hence the need of a certified solver...

Conjecture 4.3 (Inference non-specialization)**.** *Given an expression e, if inference succeeds and produces a closed term e', then for any possible well-typed completion e'' of e, $e' \equiv e''$.*

This proof has not been fully conducted yet. The main difficulty lies in the handling of let bindings that induce diverging constraint sets.

5 Toward a Realistic Array Language

Our simplistic language $\mathcal{L}^\eta$ provides the basis for an ML-like language where sizes are handled in a same way than types, i.e. with polymorphism. This section gives an insight of some extensions that are necessary for a more realistic array language. They are implemented in a compiler prototype for a domain specific language, that is synchronous and data-flow. Because it is first order, we prefer to present these extensions in a larger context here.

5.1 Locally Abstract Sizes

In OCaml, *locally abstract types* allows to declare types within a scope. These types may not escape their scope, i.e. no substitution of an outer free type variable may capture them. They serve advanced purposes (first-class modules, GADTs, ...) by introducing type variables that can be generalized outside of their scope.

Providing a similar mechanism, for both sizes and types, has a simpler use in our context: local existential quantification — **let size** $\iota = e$ **in** e'. It defines an abstract size ι in e', using the value of an arbitrary expression e. Such a mechanism is useful to overcome size language limits.

5.2 Polymorphic Recursion

Recursive algorithms on array such as the Fast Fourier Transform calls themselves on sub-arrays whose sizes vary at each call. Because sizes are part of types, polymorphic recursion is needed. This extension has been extensively studied [26, 29].

Semantics. While adding recursive declarations requires few changes on the implementation, it impacts deeply the formalization: diverging terms now exist, they must be distinguished from blocked ones in $\mathcal{L}^\eta$ big-step semantics. Fortunately, the Dagnino et al. [8] formalization was designed for non-deterministic semantics. By giving a non-deterministic evaluation of fixpoint (either stopping with an error value or reducing further), the preservation and soundness results (subsection 3.3) may be extended.

Inference. As shown by Henglein [15], polymorphic recursion turns inference into an undecidable problem. We follow the classical approach, by considering fix-points monomorphic unless explicitly generalized at declarations.

An extra check is required to ensure that the actual type of the declaration is indeed as polymorphic as the specified one, as defined in the subsumption relation in subsection 4.3. This validates *a posteriori* that the recursive occurrences of the introduced variable have been correctly instantiated.

5.3 Explicit Coercions

The size language might get too limited, especially when local existential sizes are used. We provide an *explicit coercion* —$e \blacktriangleright \tau$— to spot some size properties that cannot be check by the type system: expression's type and τ's sizes must only have the same structure, allowing for size mismatches.

As mentioned by Jay and Sekanina [20], coercions may be checked in various ways: at run-time, with defensive code or using alternative formal verification tools. In the context of static sizes, Nielson and Nielson [30] proposed *binding times*, to ensure that coercions (and local existential sizes) are computable at compiler time.

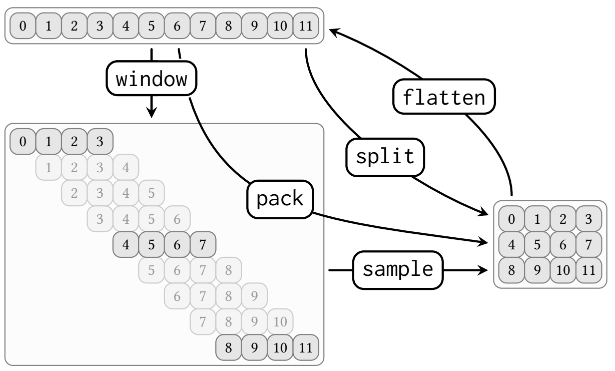
5.4 Language Support for Arrays

Arrays deserve special language constructs, both for readability and compilation purposes. Besides distinguishing their types from function's one, $\mathcal{L}^\eta$ should be extended with dedicated syntax for accesses —$e[e']$— and array definition —$[e, \dots, e]$—. As far as typing is concerned, these constructs amount for type constraint insertions, i.e. $e[e']$ requires sub-expressions to have type $[\iota]\alpha$ and $[\iota]$.

To avoid operator overloading, index manipulations are provided as a set of *first order combinators* that transform arrays' shape. They provide a safe way to introduce correct index computation. In addition to the usual `transpose`, `reverse` and `concat` linear primitives[9], the following ones are added. They are illustrated bellow.

```
val window  : ∀ι·κ·α. <κ> → [ι + κ − 1]α → [ι][κ]α
val sample  : ∀ι·κ·α. <κ> → [ι ∗ κ − κ + 1]α → [ι]α
val split   : ∀ι·κ·α. <κ> → [ι ∗ κ]α → [ι][κ]α
val flatten : ∀ι·κ·α. <κ> → [ι][κ]α → [ι ∗ κ]α

let pack  =  λs. λx. sample s (window <_> x)
                 [ ∀ι·κ·δ·α. <δ> → [ι ∗ δ − δ + κ]α → [ι][κ]α ]
```



The `window` function (see Section 2) builds a matrix whose rows are slices of the input array. The `sample` function extracts one element out of every κ, selecting both ends of the

[9] With the iterators, these are the available array functions in Scade.

array. The size of the input array must thus by a multiple of κ plus 1. Composing these functions defines a general sampling operator `pack`. It selects ι slices of size κ that are uniformly distributed and cover both ends of the input array, whose size is obtained by considering a sampling step δ. Note that the size argument of `sample` and `pack` are necessary since the associated variable might not be deduced from array size (because of non linear sizes). Although redundant, the `split` primitive carries a extra information: it defines a bijection between arrays. Defining filters or convolutions requires such building blocks (see Section 2). Here is an example of `pack` application:

$$\texttt{pack} <2> (\lambda i : [7].\ i) \triangleright [3][3]\texttt{int} \rightsquigarrow \begin{bmatrix} 0 & 1 & 2 \\ 2 & 3 & 4 \\ 4 & 5 & 6 \end{bmatrix}$$

5.5 Implicit Size Parameters

Our proposal allows to infer any *size*. However, functions' arguments of type $<\eta>$ might not be syntactically omitted: an explicit *size value* expression with an unspecified size —<_>— must at least be provided.

To make these arguments fully implicit, some syntactical restrictions are needed, so as to determine the places where such unspecified size values should be inserted. For instance, providing n-ary size abstraction —$\lambda <\iota_1, \ldots, \iota_n>.\ e$— and application —$e <\iota_1, \ldots, \iota_n>$— without curryfication makes

6 Purposes of Sized Types

The size information has several usages, both for program verification and compilation. Carrying size alongside types has proven to be practical in our ongoing experimentation.

6.1 Verification

As mentioned in Section 2, array combinators turn bound checking into size consistency (e.g., `map2` arguments must have the same size). Our type system precisely ensures this property. For decidability purposes, the type system only handles size *equalities*. In particular, it does not ensure that size are positive. In this context, array safety is based on the emptiness of type $[\eta]$ when η is negative or null: none of the language's primitives deliver indexes of negative size.

Thinking of array combinators as pure index computations, as we breifly discuss about below, linear array primitives (`concat`, `reverse`, `window`, ...) as well as index producers (`mapi`, the modulo), are indeed safe, but Pandora's box opens when providing non-linear primitives: the use of a negative step κ in `sample` would allow to build an array of positive (thus nonempty) size from a negatively sized one, hence introducing faulty accesses.

To rule out these hazardous uses, constraints must be added to type schemes. The `split` primitive is restricted to strictly positive steps[10] with the type:

$$\texttt{val sample} : \forall \iota \cdot \kappa \cdot \alpha.\ <\kappa> \to [\iota * \kappa - \kappa + 1]\alpha \to [\iota]\alpha \ \ \texttt{where}\ \ \kappa > 0$$

These constraints are ignored by the inference. They are checked either symbolically or at final instantiation, where they become trivial relations on integer values. [11]

6.2 Compilation

Conveying sizes into types is useful for compilation purposes. In particular, the declarative style favors definitions of complex data by pieces that are aggregated (e.g., using array concatenation). To avoid extra memory consumption and data moves, the placement of each part must be carefully chosen. This is, for example, the role of the *built-in-place* optimization designed by Gaudiot et al. [12] for Sisal. For arrays, it strongly relies on size information.

Iterator Fusion. Complex transformations are expressed by composing extensional primitives that produce intermediate arrays. Fusing these operations is an indispensable step for compiler of functional languages that target efficient software. In some of them [17, 35], this is achieved by using a set of rewrite rules with the drawback of requiring new rules for additional primitives or some fallback mechanism.

Other proposals such as Obsidian [6] or Dex [31] rely on the array-function analogy to provide forms that compose arbitrarily. We experiment a similar approach, restricted to array combinators by representing them in a uniform way: functions that map indexes. For instance, reversing an array of size η is described by the function $x \in [\eta] \mapsto <\eta - 1> - x \in [\eta]$, which captures the size. During code generation, these index functions induce computed array accesses that are correct by construction.

Unchecked Accesses. Currently in Scade [7], every dynamic array accesses are guarded, by providing a default value in the event that the index is out of bound. For accesses where bounds are actually met, such as the iterator `mapi` (point wise application with index), this generates dead code and an extra branching. The index refinement allows to decouple array access from bound verification: a value of type $[\eta]$ may be used in several places without any dynamic check that it is indeed within bounds.

7 Discussion and Related Works

The definition of a typed functional language with array operations offers several design choices that must be assessed. How expressive are the language and its type system; how

[10] A zero step could make sense here, by accessing the single value of an array (of size 1), but this stricter version enjoys an extra property: it is injective, which gives additional compilation perspectives.
[11] Similar constraints on type variables are already used in Scade, as shown in the linear algebra examples of Section 7

difficult and modular are type checking and type reconstruction; what about the verbosity of the code; what is diagnosis like in case of errors?

The motivation of the present work is the extension of the domain-specific functional and synchronous language Scade [7] that is used for implementing real-time embedded software. Scade stands out from general purpose functional language by being first-order with a predefined set of higher-order operators on arrays, extending a proposal for Lustre [25]. The expressiveness of the language is purposely limited to ensure safety properties (e.g., memory and execution time are bounded and known statically). Moreover, applications in Scade are almost exclusively developed graphically by connecting blocks in diagrams so that annotating elements (wires and blocks) with types is rapidly cumbersome, in particular when size expressions get larger. Neither type nor size inference are currently available in Scade. We aim at extending the expressiveness of the language beyond the linear iterators (e.g., `map`, `fold`) and providing size inference. By comparison, the matrix product of Figure 4 is written in Scade in the following way.

```
-- Scalar product of two vectors: u(n) · v(n)
function dot «n» (u, v: 'T^n)
returns (w:'T) where 'T numeric
  w = (fold $+$ «n») (0, (map $*$ «n») (u, v));

-- Product of a matrix by a vector: A(m,n) * u(n)
function mat_vec «m, n» (A : 'T^m^n; u : 'T^n)
returns (w: 'T^m) where 'T numeric
  w = (map (dot «n») «m») (transpose (A; 1; 2), u^m);

-- Matrix product: A(m,n) * B(n,p)
function mat_mat «m, n, p» (A : 'T^m^n; B : 'T^n^p)
returns (C:'T^m^p) where 'T numeric
  C = (map (mat_vec «m, n») «p») (A^p, B);
```

Here, sizes need to be expressed both in types and instantiations of array iterators and functions. The proposition presented in the paper increases significantly what it is possible to express with the current version of Scade with lighter-weight notations for both definitions and interfaces.

We hope this proposition to be applicable in a wider context than the one of Scade. Actually, this approach based on polymorphism deals with features that are not available with Scade such as higher-order and recursion.

Modularity. Circuit design languages such as Lava [5] and Wired [2] extensively use arrays. Because of their target, programs are fully expanded before size checking. This allows using arbitrary (static) expressions in sizes that are evaluated at compile time to ensure correct array use. The same approach is used for Halide [33], that produces GPU kernels: functions are compiled and optimized once sizes have been given concrete values.

For safety critical embedded software, sizes are also statically fixed, but both checking and compilation gain from modularity, e.g., by allowing easier definition of libraries, or in a view to produce modular code. For error tracking, this type system allows to spot defects of polymorphic definitions before their use in a monomorphic context.

However, since sizes are static, we do not restrict our language to the formally type-provable programs. We provide coercions as a fallback mechanism for remaining checks to be performed after specialization.

A Rudimentary System of Refined Types. Our proposal uses a very restricted form of refinement types, by providing only singleton ($<\eta>$) and interval ($[\eta]$) refinements, without sub-typing between them. This is a key for both type checking and inference.

The general theory of dependent types worked out by Xi and Pfenning [38] allows to express arbitrary predicates in type systems. However, this has a cost: type checking is undecidable in general, mainly because sub-typing amounts to proof obligations of predicate implication. These authors also delineated in [37] some restrictions for arrays size checking. Trojahner and Grelck [36] extended a similar type system to provide dependently typed rank polymorphism. Both works extract sets of arithmetic constraints that are resolved with an external procedure (SMT solvers). Besides requiring heavy machinery for type checking at the risk of opaque errors, size relations are mainly limited to linear expressions, for the constraint system to be solvable. To lift this reductions, λ^H proposes *hybrid type checking* [11], a system of refinement types that allows deferring unprovable implications to run time. Our proposal resembles, using static evaluation to eliminate remaining checks.

Inference in extended type systems has been studied in the context of λ^H [24] and Liquid Types [34]. Both approaches are similar to ours: after collecting the set of sub-typing constraints, a most general solution is extracted (using external tools such as SMT solvers).

Explicit Proof Obligations (coercions). We provide an escape mechanism for the terms that cannot be checked by the type systems In λ^H [11] such coercions are ubiquitous, although implicit: they are systematically inserted at function applications, generating checks that are eliminated at compile time if possible. In the restricted context of arrays, far less coercions are needed that in the general setting. Thus, we follow the approach of Futhark [16] and Dex [31] (coercions occur at `fromOrdinal` calls): every possibly failing coercion is explicit. This strengthens the guarantees provided by typing: coercion errors may occur only at explicitly marked points of the program.

Jay and Sekanina [20] mention several ways to check shape constraints, that apply for our coercions: defensive run-time code generation, static checking with advanced formal methods or partial evaluation at compile time.

A Separated Size Language. Our sizes are syntactically distinguished from general expressions. This matters for cross-compilation (which is common for embedded applications), because sizes are symbolically manipulated at compile-time i.e. on the development host, whereas integer values must be represented on the targeted device with machine (finite) integers that are submitted to overflows. Converting sizes into machine integer is thus non-trivial: then compiler must ensures size's value is actually representable within the concrete type.

The idea of using a separated language for sizes appeared already in [19] to express bounds on the size of recursive data-types. The set of sizes is extended a limit ω that to represent data of arbitrary size.

The VEC language of Jay and Sekanina [20] follows a different path, by enforcing size expressions (a subset of expressions of the language) to be independent of data that come with dedicated typing rules. This results in an analogous restrictions: sizes are static.

Indexed Types. Our proposal strongly resembles the one of Zenger [39], where types are parametrized by polynomial *indexes*. Alike our type system, dependent types are avoided by considering indexes as types, that are syntactically separated from terms. It is even stricter: the values may not depend on indexes. Hence, the semantics remains type-erasable and no sub-typing is required, but bounded integers are hardly representable.

The purpose of inference differs too: instead of reconstructed omitted sizes, inference for indexed types constraints and checks whether they are satisfied. To this end, it builds on the general theory of radical ideals, that is applicable for complex polynomials. For size reconstruction, this would lead to meaningless sizes (e.g., non-integer).

Comparison to FUTHARK and DEX. The FUTHARK [9] and DEX [31] languages shares strong similarities with this work. The founding principle seems similar: most array sizes should be controlled in some inexpensive ways, without trying to fully check arrays, at the risk of limiting language expressiveness. Instead of proving predicates, these type systems keep track of values' properties (bounds), allowing to decouple their verification, either static (argument assumptions) or dynamic (coercions) and their uses (array accesses). This finer control also benefits the compilation by helping to rule out redundant checks.

Both DEX and FUTHARK provide dependent types that allow arbitrary size computations, but static verifications are limited to syntactic equality. Hence, neither `concat` nor `reshape` are given a satisfactory type. Our sizes must be static, but the size language is more expressive. Despite the loss of completeness for inference, such polynomial sizes seems useful because they are easily checkable.

Polymorphism, Dependent Types and Dynamic Sizes. Separating the size language has a direct consequence: notions of scopes, abstractions and applications are needed to express terms that are generic in sizes. Because they are not terms, dependent types are inadequate here. Following Hughes et al. [19], we handle similarly sizes and types (let generalization), which is consistent, at least in the context of static sizes.

The dimension type system proposed by Kennedy [23] also resembles ours: polymorphism over dimensions is considered. Its inference enjoys principal types, since the dimension language is simpler: it is equipped with a single operation, the product, that is both associative and commutative. However, some difficulties still echo to ours: most general types are not unique and polymorphic recursion seems rapidly necessary.

Even FUTHARK, whose type system uses dependent types, shares strong similarities: its normalized form [16] is obtained by adding let bindings so as to name and scope existentially quantified size variables.

Moreover, the work on indexed types [39] illustrates how existential quantification in data types allows to handle dynamically sized data-structure. Since the work of Mitchell and Plotkin [28], first class polymorphism [21, 22] generalized these data-types to universal quantification. In this context, typing heavily relies on the local quantification sketched in Section 5.

8 Conclusion and Perspectives

This article have presented an ML-like type system which adds a size information into types: genericity on sizes is expressed through polymorphism. The size language, made of multivariate polynomials, allows to express a large class of array manipulations, while being easily checked.

Our proposal is not restricted to safety critical languages like SCADE: it may provide the key elements to track array sizes in a functional language, and to highlight the parts that cannot be checked with a simple ML-like type system. This information is valuable for both checking and compilation steps, in particular to reduce the need for defensive code.

Besides assessing the compatibility of this type system with temporal constructs of synchronous languages, the efficient compilation of the proposed array manipulations (array iterators, recursion on size) in the context of safety critical software will be studied next. In particular, targeting devices that are used for intensive computation such as GPUs remains an open question for such applications.

Acknowledgments. We thanks the anonymous referees for their helpful and relevant comments.

References

[1] Alexander Aiken and Edward L. Wimmers. 1993. Type Inclusion Constraints and Type Inference. In *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, John Williams (Ed.). ACM, 31–41. https://doi.org/10.1145/165180.165188

[2] Emil Axelsson, Koen Claessen, and Mary Sheeran. 2005. Wired: Wire-Aware Circuit Design. In *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3725)*, Dominique Borrione and Wolfgang J. Paul (Eds.). Springer, 5–19. https://doi.org/10.1007/11560548_4

[3] John G. P. Barnes. 2003. *High Integrity Software - The SPARK Approach to Safety and Security.* Addison-Wesley. http://www.addison-wesley.de/main/main.asp?page=englisch/bookdetails&ProductID=88293

[4] Richard S. Bird and Philip Wadler. 1988. *Introduction to functional programming.* Prentice Hall.

[5] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 174–184. https://doi.org/10.1145/289423.289440

[6] Koen Claessen, Mary Sheeran, and Joel Svensson. 2012. Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the POPL 2012 Workshop on Declarative Aspects of Multicore Programming, DAMP 2012, Philadelphia, PA, USA, Saturday, January 28, 2012*, Umut A. Acar and Vítor Santos Costa (Eds.). ACM, 21–30. https://doi.org/10.1145/2103736.2103740

[7] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development (invited paper). In *11th International Symposium on Theoretical Aspects of Software Engineering, TASE 2017, Sophia Antipolis, France, September 13-15, 2017*, Frédéric Mallet, Min Zhang, and Eric Madelaine (Eds.). IEEE Computer Society, 1–11. https://doi.org/10.1109/TASE.2017.8285623

[8] Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. 2020. Soundness Conditions for Big-Step Semantics. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 169–196. https://doi.org/10.1007/978-3-030-44914-8_7

[9] Martin Elsman, Troels Henriksen, and Niels Gustav Westphal Serup. 2019. Data-parallel flattening by expansion. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ARRAY@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Jeremy Gibbons (Ed.). ACM, 14–24. https://doi.org/10.1145/3315454.3329955

[10] John Feo, David C. Cann, and R. R. Oldehoeft. 1990. A Report on the Sisal Language Project. *J. Parallel Distributed Comput.* 10, 4 (1990), 349–366. https://doi.org/10.1016/0743-7315(90)90035-N

[11] Cormac Flanagan. 2006. Hybrid type checking. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 245–256. https://doi.org/10.1145/1111037.1111059

[12] J-L Gaudiot, Wim Bohm, Walid Najjar, Tom DeBoni, John Feo, and Patrick Miller. 1997. The Sisal model of functional programming and its implementation. In *Parallel Algorithms/Architecture Synthesis (Proceedings. Second Aizu International Symposium)*. IEEE, 112–123.

[13] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. 2012. A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler. In *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2012, LCTES '12, Beijing, China - June 12 - 13, 2012*, Reinhard Wilhelm, Heiko Falk, and Wang Yi (Eds.). ACM, 51–60. https://doi.org/10.1145/2248418.2248426

[14] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320. https://doi.org/10.1109/5.97300

[15] Fritz Henglein. 1993. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 2 (1993), 253–289. https://doi.org/10.1145/169701.169692

[16] Troels Henriksen and Martin Elsman. 2021. Towards size-dependent types for array programming. In *ARRAY 2021: Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, Virtual Event, Canada, 21 June, 2021*, Tze Meng Low and Jeremy Gibbons (Eds.). ACM, 1–14. https://doi.org/10.1145/3460944.3464310

[17] 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, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 556–571. https://doi.org/10.1145/3062341.3062354

[18] Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society* 146 (1969), 29–60.

[19] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 410–423. https://doi.org/10.1145/237721.240882

[20] C Barry Jay and Milan Sekanina. 1997. Shape checking of array programs. In *Computing: the Australasian Theory Seminar, Proceedings.* Australian Computer Science Communications, Vol. 19. University of Technology, Sydney, Australia, 113–121.

[21] Mark P. Jones. 1997. First-class Polymorphism with Type Inference. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 483–496. https://doi.org/10.1145/263699.263765

[22] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of functional programming* 17, 1 (2007), 1–82. https://doi.org/10.1017/S0956796806006034

[23] Andrew Kennedy. 1994. Dimension Types. In *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 788)*, Donald Sannella (Ed.). Springer, 348–362. https://doi.org/10.1007/3-540-57880-3_23

[24] Kenneth L. Knowles and Cormac Flanagan. 2007. Type Reconstruction for General Refinement Types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4421)*, Rocco De Nicola (Ed.). Springer, 505–519. https://doi.org/10.1007/978-3-540-71316-6_34

[25] Florence Maraninchi and Lionel Morel. 2004. Arrays and Contracts for the Specification and Analysis of Regular Systems. In *4th International Conference on Application of Concurrency to System Design*

(ACSD 2004), 16-18 June 2004, Hamilton, Canada. IEEE Computer Society, 57–66. https://doi.org/10.1109/CSD.2004.1309116

[26] Lambert G. L. T. Meertens. 1983. Incremental Polymorphic Type Checking in B. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum (Eds.). ACM Press, 265–275. https://doi.org/10.1145/567067.567092

[27] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[28] John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 3 (1988), 470–502. https://doi.org/10.1145/44501.45065

[29] Alan Mycroft. 1984. Polymorphic Type Schemes and Recursive Definitions. In *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings (Lecture Notes in Computer Science, Vol. 167)*, Manfred Paul and Bernard J. Robinet (Eds.). Springer, 217–228. https://doi.org/10.1007/3-540-12925-1_41

[30] Hanne Riis Nielson and Flemming Nielson. 1988. Automatic Binding Time Analysis for a Typed lambda-Calculus. *Science of computer programming* 10, 1 (1988), 139–176. https://doi.org/10.1016/0167-6423(88)90025-1

[31] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473593

[32] François Pottier. 2001. Simplifying Subtyping Constraints: A Theory. *Information and computation* 170, 2 (2001), 153–183. https://doi.org/10.1006/inco.2001.2963

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 519–530. https://doi.org/10.1145/2491956.2462176

[34] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

[35] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 205–217. https://doi.org/10.1145/2784731.2784754

[36] Kai Trojahner and Clemens Grelck. 2009. Dependently typed array programs don't go wrong. *J. Log. Algebraic Methods Program.* 78, 7 (2009), 643–664. https://doi.org/10.1016/j.jlap.2009.03.002

[37] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 249–257. https://doi.org/10.1145/277650.277732

[38] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 214–227. https://doi.org/10.1145/292540.292560

[39] Christoph Zenger. 1997. Indexed Types. *Theoretical computer science* 187, 1-2 (1997), 147–165. https://doi.org/10.1016/S0304-3975(97)00062-5

Received 2023-03-31; accepted 2023-04-21