FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master thesis

Henriks Urms (mgs837) and Anna Sofie Kiehn (mvq686)

Refinement types in Futhark

Supervisor: Martin Elsman

Handed in: September 2, 2019

Abstract

Among programming languages supporting arrays, there are those that raise runtime errors when an array is indexed out of bounds and those that omit these checks for the sake of performance as dynamic array bound check takes time during execution. Futhark is a programming language designed to target general purpose computing on graphics processing units (GPUs). We propose that extending the Futhark type system with refinement types will allow the programmer to refine a program using types in such a way that array bounds checking can be avoided at runtime. In this thesis we present a small subset of the Futhark language with additional refinement type annotations and a transformation function that transforms programs to an internal language representation. The internal language features explicit polymorphism, dependent quantifiers and refinement types. We also present a set of bidirectional type rules for the internal language and a type checker implemented in Haskell based on the type rules. We have shown that we can refine the types of a small program in such a way that array bounds checks can be safely omitted.

Contents

1 Introduction

Among programming languages supporting arrays, there are those that raise runtime errors when an array is indexed out of bounds and those that omit these checks for the sake of performance as dynamic array bounds checks take time during execution. Futhark is a programming language designed to target general purpose computing on graphics processing units (GPUs). Futhark also has a C backend. In Futhark, array accesses are checked at runtime unless these checks are turned off by the programmer using the keyword `unsafe` [7, 12, 17]. Omitting array bounds checks makes it easy to introduce hard-to-find bugs and in many programming languages, checking array bounds has a negligible influence on performance, which make array bounds checking a practical solution in many contexts. In the context of the Futhark language and its implementation targeting GPUs, the goal is to generate high-performance code, which means that it becomes paramount to avoid runtime array bounds checking. Moreover, it is often unclear what should happen in the case of a runtime error on the GPU since all threads need to finish before the error can be propagated to the CPU, which in the end is responsible for reporting the error to the user.

For now, the Futhark compiler will refuse to compile programs with runtime array bounds checks for the GPU and thus, Futhark programmers have to turn array bounds checks off using `unsafe` if they want to run their code on the GPU.

One solution for avoiding runtime array bounds checks, while still making sure that all array accesses are in-bounds, is to use a type system that allows the compiler to reason about array indexing at compile time. Refinement types provide us with such a type system [25].

In the most general sense, a refinement type system is an extension of a base type system that adds an extra layer of precision to the type system. It is a conservative extension in the sense that all programs that are typeable in the refinement type system are also typeable in the base system. That is, refinement types do not make more programs typeable.

The added information to the type system can be used to express the length of an arrays as part of the type or that all elements of an array are in the range 0 to 5.

Consider the following function from the Futhark segmented library [13, 15]:

```
1 val replicated_iota [n] : [n]i32 -> []i32
```

The expression `replicated_iota reps` returns an array with each index (starting from 0) repeated according to the repetition array `reps`. As an example, `replicated_iota [2,3,1]` returns the array `[0,0,1,1,1,2]`. A property of this function, which will be useful later, is that all values in the result array are nonnegative and strictly less than n, the length of the argument array. Refinement types can be used to express this property:

```
val replicated_iota [n] : [n]i32 -> []{i32(i) | i >= 0 && i < n}
```

Below is a Futhark function `segmented_replicate`, which uses an unsafe array access and, which uses the `replicated_iota` function:

```
let segmented_replicate [n] (reps:[n]i32) (vs:[n]i32) : []i32 =
  let idxs = replicated_iota reps
  in map (\i -> unsafe vs[i]) idxs
```

Because the function `replicated_iota` is used to create the indices that are used in the array access, the compiler can use the extended type information to establish that the array access is safe.

Even for the Futhark C backend, bounds checking slows down the performance because more instructions need to be executed. Thus, also in this case, we can benefit from a refinement type system. We can evaluate the performance benefits of omitting runtime array bounds checks by comparing the execution times of programs with and without array bounds checks.

We propose that extending the Futhark type system with refinement types will allow the programmer to refine a program using types in such a way that array bounds checking can be avoided at runtime.

We will test this hypothesis by (1) designing a refinement type system for Futhark, (2) implementing it in a small compiler that erases the types and produces safe Futhark code without checks, and (3) evaluating its use based on a number of small and medium examples. Moreover, we will evaluate the usability of the refinement type system, for instance by comparing example refinement-type-annotated Futhark programs with their unannotated versions to inspect the added complexity.

1.1 Our approach

To not complicate the types of Futhark too much, we have specified an internal language with fully indexed types and given so-called bidirectional type rules for refinement types in the internal language.

The internal language is verbose and does not resemble Futhark very much, thus we have also specified an external language, which is a subset of Futhark with the addition of allowing a few extra type annotations.

To type check programs in the external language, we have designed a transformation function that transforms programs in the external language to programs in the internal language.

By transforming programs, we also simplify our type rules because they do not have to deal with the quirks of Futhark.

The type rules are inspired by previous work on refinement types [10, 24, 25]. Our rules are very similar to the algorithmic rules of [10] and we change the rules or add our own as needed.

The type rules are deterministic in the sense that for every subderivation at most one rule is applicable, and algorithmic, which means that the types the rules are used on are deterministically determined from the input. This means that rules are directly implementable without any other decisions in the implementation. We have implemented a typechecker in Haskell based on the type rules.

1.2 Report overview

In Section 1, we present the motivation for the work and present the thesis. In Section 2, we briefly cover some preliminaries including, the Futhark language, bidirectional typing, refinement types, dependent types and index refinements. In Section 3, we introduce the syntax and key elements of our internal language. In Section 4, we introduce our external language, which, for demonstration purposes, is a subset of the Futhark language. In Section 5, we present a transformation function that transforms a program in the external language to the internal language representation. In Section 6, we present the bidirectional type rules for typecheking a program in the internal language. In Section 7, we describe and discuss our implementation of the rules presented in Section 6. We also describe the implementation of the parser, the transformation function between the external and internal language, the erasure function and a small solver. In Section 8, we evaluate our implementation, its effect on runtime performance, the usefulness of the solution and the process of the project. In Section 9, we discuss our type rules, the coverage of Futhark, the expressiveness of refinement types, challenges with existential types, type inference and constraint solving. In

Section 10, we present related work. In Section 11, we conclude and discuss future work.

1.3 Methods and tools

In this section, we present how we have approached the development of the implementation. We also introduce the concept of bidirectional typechecking that we have used when developing type judgements and rules.

1.3.1 Development style

When developing the implementation, we have used examples to guide focus and direction. Before writting the implementation of a rule, we wrote an example of its application as a test in the file */test/Spec.hs*.

1.3.2 Bidirectional type rules

Bidirectional typechecking is a technique for designing type systems in which terms in some context either synthesize a type or are checked against a known type. It makes it easy to make the type rules syntax directed or almost syntax directed so that they are easily implemented as recursive functions that pattern match on the syntax. Type annotations only have to be added where they are necessary. It is the middle ground between fully annotated systems, which are easy to design and implement but are annoying to use, and type inference, which is difficult to understand and extend to handle types that are not simple.

Because of these properties, bidirectional typechecking is a popular choice for designing type systems with advanced types [9]. We follow the same approach as has been used in previous systems [9, 10, 24, 25] and present a bidirectional type system for refinement types in Futhark.

1.3.3 Haskell

We have used GHC Haskell (`https://www.haskell.org/`) for our implementation, but we have not used any significant GHC extensions. Monads are used to structure our implementation.

2 Preliminaries

In this section we describe relevant parts of the Futhark language. We also briefly explain refinement types and dependent types as well as index refinements.

2.1 The Futhark language

Futhark [7,12,17] is a a statically typed, data-parallel, and purely functional array language with a higher-order ML-style module system [11] and limited support for higher-order functions [18].

Futhark has a type system featuring parametric polymorphism and type inference.

At the core of Futhark is a number of built-in functions, such as `iota`, `map` and `filter`, the so-called SOACs (second order array combinatores). These combinators express the inherent parallelism of the programs in which they are used and are a key part of Futhark and are the reason that Futhark code can be compiled for parallel execution, as the SOACs have parallel semantics.

To make array programming more ergonomic, arrays in Futhark carry around their size, which can be passed implicitly to functions by the compiler. For example, the `length` function in Futhark is defined as follows:

```
1 let length [n] 't (_: [n]t): i32 = n
```

The function takes an array of size n and returns a signed 32bit integer. The `[n]` here is a size parameter, which makes the compiler pass in the length of the array implicitly and the size variable n can be used in the body of the function. Even though the size is a runtime construct, the size here is syntactically part of the array type, and so, it is also useful for documentation. For example, the `reverse` function returns an array of the same size as the input array, which is expressed in its type:

```
1 let reverse [n] 't (x: [n]t): [n]t = x[::-1]
```

The invariant that the size of the input and the output is the same is checked at runtime (if it cannot be garuanteed by the compiler at compile time). Both functions are polymorphic in the array element type and the type parameter is declared with a `'t`. Thus, size parameters and type parameters are in the same syntactic category. Futhark also supports using program variables as

sizes in array types. For example, the `iota` function returns an array of size n containing values between 0 and n.

```
1 let iota (n: i32): [n]i32 = 0..1..<n
```

One of Futhark's features is that it can be compiled for execution on GPUs, which are notoriously difficult to program in low level languages such as CUDA [1]. As a consequence, array bounds checks in Futhark can be turned off using the keyword `unsafe`. The reason is that there is no good way to deal with out of bounds errors during the execution of the GPU code as many threads are executed in a SIMD fashion and some of them may encounter errors while others may not. Since threads may do synchronization, it is not a good idea just to terminate the threads that go wrong because that may send the program into deadlock. Even if all threads are terminated, it would be difficult to propagate a helpful error message to the user besides that "some array index error happened".

2.2 Refinement types

A refinement type system is used to extend an already existing type system. Specifically, refinement types can express the fact that a value is in a subset of values of a specific type. For instance, the integers that are odd numbers is a refinement on the integer type. These subsets of types are types themselves in a refinement type system. That is, they can be argument types, return types, or variable types.

Refinement types is a conservative extention in the sense that they do not allow more programs to type check than the un-refined type system does [14].

An interesting subset of arrays is the subset of arrays of a specific length. By expressing the length of an array as part of its type, it is possible to reason about which subset of integers may be used to index the array.

2.3 Dependent types

Dependent types are types that depend on a value. In full spectrum dependent types, this value can be any value of the language, even functions. For example, an array type may be indexed by an integer representing the size of the array. The type indices can be quantified over by the Π quantifier, which is a universal quantifer, and the Σ quantifier, which is an existential

quantifier. In a fully dependently typed language, the type checker might go into an infinite loop since a type can contain any term, which in general need not succeed at evaluating to a value.

Examples of programming languages with dependent types include Coq [4], Agda [2, 26], and Idris [5].

2.4 Index refinements

Combining type refinements and dependent types, we arrive at index refinements, where the terms used to index the type are refined. This means that it is possible to form subsets of families of types instead of subsets of sets of values directly. This technique was used to implement a restricted form of dependent types in Dependent ML [24, 25]. We use the same technique. Dependent ML is further discussed in Section 10.

3 Internal Language

In this section, we present the internal language, which is used for type checking. We later present type rules over the syntax of the internal language. All types in the internal language are indexed to support index refinements. The syntax can be seen in Figure 1. The syntax is inspired by the work in [10] and [25].

In the internal language, expressions consist of variables x; lambda expressions $\lambda x.e$; let expressions $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$; function application $e\, s^+$; pairs $\langle e_1, e_2 \rangle$ and annotated expressions $(e : \tau)$.

Instead of the usual application form $e_1\, e_2$, we instead follow [10] and apply a function to a whole sequence of arguments called a spine. Application and spines both use juxtaposition, thus, spines have higher syntax precedence than applications. For example, $e_1\, e_2\, e_3$ is parsed as e_1 applied to the spine $e_2\, e_3$.

Index objects can be index variables a; integers n; pairs $\langle i_1, i_2 \rangle$; function applications $f(i)$ and binary functions $i_1 + i_2$.

Types are indexed by index objects i, which themselves are typed. To avoid confusion, the types of index objects are called sorts and are denoted by γ. For better readability, sorts are written in *italic* and types in `typewriter`.

Also, f is used for interpreted index functions (like $+$), and functions of sort $\gamma \to \mathit{bool}$ are called predicates and are denoted by the symbol p.

Types can be pairs $\tau_1 * \tau_2$; function types $\tau_1 \to \tau_2$; universal types $\forall a : \kappa.\tau$; existential types $\exists a : \gamma.\tau$; guarded types $P \supset \tau$ and asserting types $\tau \triangleright P$.

We use $\forall$ and $\exists$ for universal and existential types instead of the more traditional Π and Σ from dependent types. The quantifiers can quantify over both types and indices so $\forall$ is used for both parametric polymorphism and Π-types.

Guarded types are types that can only be used if the property guarding the type can be proved. Most often they guard function types with the intention that a function cannot be applied unless the argument type satisfies some property. Asserting types are types that establish some property about the type they contain. For example, we can say that some number is positive if it can be assigned the type $\texttt{i32}(n) \triangleright (n > 0)$ for some index n in scope.

We use P for propositions over indices, which are used in guarded and asserting types.

We use δ for base type families (e.g., `i32, i64, bool, array`) and b for base index sorts (e.g., *int, bool*). Types have to be indexed by index objects of the appropriate sort (e.g., `i32` and `array` have to be indexed by objects of sort *int* and so on). Notice that the type families `i32` and `i64`, which are indexed by objects of sort *int*, have a limited range whereas their indices do not.

expressions	e	::=	$x \mid \lambda x.e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid e\, s^+ \mid$ $\langle e_1, e_2\rangle \mid (e : \tau) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e)$
spines	s	::=	$\cdot \mid e\, s$
nonempty spines	s^+	::=	$e\, s$
index objects	i	::=	$a \mid n \mid \langle i_1, i_2\rangle \mid f(i) \mid i_1 + i_2$
types	τ	::=	$\delta(i) \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2 \mid \forall a : \kappa.\tau \mid \exists a : \kappa.\tau \mid$ $P \supset \tau \mid \tau \triangleright P$
index sorts	γ	::=	$b \mid \gamma_1 * \gamma_2$
kinds	κ	::=	$\star \mid \gamma$
index propositions	P	::=	$\top \mid \bot \mid p(i) \mid i_1 \le i_2 \mid P_1 \wedge P_2 \mid P_1 \vee P_2$

Figure 1: Syntax of the internal language.

3.1 Representing unindexed types

Since all types in the internal language are indexed, one might wonder how to opt out of refinement types in parts of the program. It should be possible in the type system to represent good old types such as `i32, i64, []i32, bool`, and so on. Furthermore, it should also be possible to apply functions that expect indexed types to unindexed types. Otherwise we would have to use different functions for indexed and unindexed types. For example, $(+)$ for indexed `i32` has the type:

$$\forall n : \mathit{int}.\forall m : \mathit{int}.\texttt{i32}(n) \to \texttt{i32}(m) \to \texttt{i32}(n + m)$$

We would like to be able to use this one type of $(+)$ for both unindexed and indexed arguments since having multiple types and picking the right one would be overly cumbersome for the programmer.

To represent such unindexed types, we follow [25] and use existentially quantified types so that `i32` becomes a shorthand for $\exists a : \mathit{int}.\texttt{i32}(a)$. Such a type can be interpreted as at type where the index is unknown. The term `i32` can mean several different things; it is used to denote the shorthand described above and the type family of which the type $\texttt{i32}(i)$ is a member (for any i).

For integer types, an unknown index means that the value is unknown because it is a singleton type whereas for arrays, an unknown index means that the size of the array is unknown.

3.2 Existential types

Our type system supports existential types. For example, a monomorphic version of `filter` has the type:

$$\forall n : \mathit{int}.(\texttt{i32} \to \texttt{bool}) \to \texttt{i32}\ \mathit{array}(n) \to \exists m : \mathit{int}.\texttt{i32}\ \mathit{array}(m) \triangleright (m \leq n)$$

The above function uses an existentially quantified type as the return type. The reason is that the size of the returned array is not known; it depends on how many elements satisfy the predicate.

Our existential types are also first-class. For example, they can appear as array element types; without this feature, we would only be able to represent types of arrays that contain identical elements.

Existential types introduce complications into the use of the type system. Values with existential types cannot be used as arguments to functions which universally quantify over the same parameters. Consider the following expression, which is not directly typeable:

```
1 | map (+2) (filter even [1,2,3])
```

Here the type of `map` is

$$\forall n : int. \forall \alpha. \forall \beta. (\alpha \to \beta) \to \alpha\ array(n) \to \beta\ array(n)$$

The array returned from `filter` is packed in an existential type, thus, it cannot be passed to `map` because map expects an array. In a sense, we have en array of unknown length but we need an array of known length. To work around this problem, we can transform the program to the following equivalent form:

```
1 | let arr = filter even [1,2,3] in map (+2) arr
```

The rules for let-expressions allow us to unpack such an existential type.

The typing rules are presented in Section 6.

4 External Language

The external language closely resembles a subset of Futhark with a few extensions that make it possible to decorate programs with refinement type annotations. The language could easily be changed such that the extended type annotations were contained in comments in the style of Liquid Haskell [20, 22].

One of the design goals is for the extended annotations to be as close as possible to the way Futhark's types are already expressed.

The syntax is shown in Figure 2. Expressions can be variables x, lambda expressions $\lambda p^+ \to e$, function application $e_1\, e_2^+$, binary operators $e_1 \oplus e_2$, array indexing $x[e]$, let bindings **let** $x = e_1$ **in** e_2, and let bound functions **let** $x\ tp^*\ p^* : \tau = e_1$ **in** e_2.

Functions can have type-level parameters which can be either type parameters, which are prepended with a single quote, or size parameters, which are surrounded with square brackets. A function parameter can be annotated with a type.

The expression syntax is purely a subset of Futhark.

Types however have been extended. As in Futhark, we have type variables α, integer types `i32`, boolean types `bool` and array types $[]\tau$ and $[i]\tau$. But we have added new types: Indexed integer and boolean types $\texttt{i32}(i)$, $\texttt{bool}(i)$, and property types $\{\tau \mid P\}$.

Type indices can be index variables, integer literals, or boolean literals. As in the internal language, types have to be indexed by indices of the correct sort.

A property type is used to establish some proposition involving one or more indices. If an indexed type is used outside of braces, any index variable has to refer to a variable already in scope, but if it is used inside a pair of braces, any index variables in the type are brought into scope and can be used in the proposition after the pipe.

For example, $\{\texttt{i32}(n) \mid 0 \leq n\}$ is the type of nonnegative integers.

Expressions	e	::=	$x \mid n \mid \lambda p^+ \to e \mid e_1\, e_2^+ \mid e_1 \oplus e_2 \mid x[e] \mid$ **let** $x = e_1$ **in** $e_2 \mid$ **let** $x\ tp^*\, p^* : \tau = e_1$ **in** e_2
type paramater	tp	::=	$'\alpha \mid [a]$
parameter	p	::=	$x \mid (x : \tau)$
types	τ	::=	$\alpha \mid \texttt{i32} \mid \texttt{bool} \mid \texttt{i32}(i) \mid \texttt{bool}(i) \mid []\tau \mid [i]\tau \mid \{\tau \mid P\} \mid \tau_1 \to \tau_2$
type index	i	::=	$a \mid n \mid b$
proposition	P	::=	$P_1 \wedge P_2 \mid i_1 \leq i_2 \mid i_1 < i_2$

Figure 2: Syntax of the external language.

5 Program Transformation

The transformation of programs from the external language (presented in Section 4) to programs in the internal language (presented in Section 3) is split in two transformation functions T_τ and T_e and an auxillary function for function parameters T_ϕ. The functions transform the types of the external language and the expressions of the external language to the internal language types and expressions, respectively.

The transformation of types and expressions from the external language to the internal language can be seen in Figure 3 and Figure 5. The auxillary transformation can be seen in Figure 4.

In the internal language, we use type indices that do not exist in the external language. The type indices therefore have to be generated during the transformation. The base type `i32` is transformed to an existential type $\exists m : int.\texttt{i32}(m)$ in the internal language. This means that the types has the index m but that m is unknown at this time. Likewise with the base type `bool` that can be either true or false. When the actual value is unknown, it is transformed to an existential type $\exists m : bool.\texttt{bool}(m)$.

An array can be declared with or without a size variable (length) in the external language. In the case where it has no length given, the array type $[]t$ will be transformed to an existential type $\exists a : int.T_\tau(t)array(a) \triangleright (0 \leq a)$ and the transform function is called on the type of the array. We use an asserting type here because an array always has a nonnegative length. If a length is given, the array is simply transformed to an array of that length, and like before, the type of the array is transformed recursively. We use $[n]t$ for constant size arrays and $[a]t$ for arrays where the size is expressed with a variable.

Function types in the external language are transformed to function types in the internal languages and the types of arguments are found by calling the transform function recursively.

The function F produces a list of variables ϕ that should be bound in $\forall$ quantifiers. A variable should be bound in a $\forall$ quantifier if it is referenced in other parameters or the return type. The $\text{FV}(rt)$ is the set of free variables in the return type and the condition from before is equivalent to checking if the variable is in that set. We define $\forall\phi.\tau$ inductively according to the following equations:

$$\forall(n : int), \phi.\tau = \forall(n : int).\forall\phi.\tau \qquad\qquad \forall \cdot .\tau = \tau$$

The F function can be seen in Figure 6.

In the internal language array indexing is represented with a built-in function .index which has the type

$$.\text{index} : \forall\alpha : \star.\forall n : int.\alpha\ array(n) \to \texttt{fin}(n) \to \alpha$$

When we transform a function body we prepend it with the `unsafe` keyword.

$$
\begin{aligned}
T_\tau(\texttt{i32}) &= \exists a : \mathit{int}.\texttt{i32}(a) \\
T_\tau(\texttt{bool}) &= \exists a : \mathit{bool}.\texttt{bool}(a) \\
T_\tau(\texttt{i32}(i)) &= \texttt{i32}(i) \\
T_\tau(\texttt{bool}(i)) &= \texttt{bool}(i) \\
T_\tau([\,]t) &= \exists a : \mathit{int}.T_\tau(t)\mathit{array}(a) \triangleright (0 \leq a) \\
T_\tau([a]t) &= T_\tau(t)\mathit{array}(a) \triangleright (0 \leq a) \\
T_\tau([n]t) &= T_\tau(t)\mathit{array}(n) \\
T_\tau(\tau_1 \to \tau_2) &= T_\tau(\tau_1) \to T_\tau(\tau_2) \\
T_\tau(\{\texttt{i32}(a) \mid P\}) &= \exists a : \mathit{int}.(\texttt{i32}(a) \triangleright P) \\
T_\tau(\{[a]t \mid P\}) &= \exists a : \mathit{int}.([a]T_\tau(t) \triangleright P \wedge (0 \leq a)) \\
T_\tau(\alpha) &= \alpha
\end{aligned}
$$

Figure 3: The transformation of types from the external to the internal language.

$$
\begin{aligned}
T_\phi([n]\ tps^*, p^*, rt) &= \forall n : \mathit{int}.T_{\phi,n}(tps^*, p^*, rt) \\
T_\phi(\text{'}a\ tps^*, p^*, rt) &= \forall a : \star.T_\phi(tps^*, p^*, rt) \\
T_\phi(\cdot, (x : t)\ p^*, rt) &= T_\tau(t) \to T_\phi(\cdot, p^*, rt) \\
T_\phi(\cdot, (n : \texttt{i32})\ p^*, rt) &= \begin{cases} \texttt{i32}(n) \to T_\phi(\cdot, p^*, rt) & n \in \phi \\ T_\tau(\texttt{i32}) \to T_\phi(\cdot, p^*, rt) & n \notin \phi \end{cases} \\
T_\phi(\cdot, \cdot, rt) &= T_\tau(rt)
\end{aligned}
$$

Figure 4: The transformation of parameters to types in the internal language.

$$
\begin{aligned}
T(\textbf{let } f\ tp^*\ p^+ : rt = e_1 \textbf{ in } e_2) &= \textbf{let } f = (T(\lambda p^+ \to e_1) : \forall \phi. T_\phi(tp^*, p^+, rt)) \textbf{ in } T(e_2) \\
&\qquad \textbf{where } \phi = F(p^+, rt) \\
T(\textbf{let } x = e_1 \textbf{ in } e_2) &= \textbf{let } x = T(e_1) \textbf{ in } T(e_2) \\
T(\lambda(x : \tau)\, p^* \to e) &= \lambda x.\, T(\lambda p^* \to e) \\
T(\lambda x\, p^* \to e) &= \lambda x.\, T(\lambda p^* \to e) \\
T(\lambda \cdot \to e) &= T(e) \\
T(e_1\, e_2) &= T(e_1)\, T(e_2) \\
T(x) &= x \\
T(n) &= n \\
T(x[e]) &= .\text{index}\, T(e)\, x \\
T(e_1 \oplus e_2) &= T(e_1) \oplus T(e_2)
\end{aligned}
$$

Figure 5: The transformation of expressions from the external to the internal language.

$$
\begin{aligned}
F((n : \tau)\, ps, rt) &= \begin{cases} (n : \tau), F(ps, rt) & n \in FV(ps) \cup FV(rt), \tau \in \{\texttt{i32}, \texttt{bool}\} \\ F(ps, rt) & \text{otherwise} \end{cases} \\
F(\cdot, rt) &= \cdot
\end{aligned}
$$

Figure 6: Helper function that produces a list of variables that needs to be bound by forall quantifiers.

The erasure function that transforms a program in the external language to a Futhark program can be seen in Figure 7.

$$
\begin{aligned}
ER_e(\textbf{let } f\ tp^*\ p^+ : rt = e_1 \textbf{ in } e_2) &= \textbf{let } f\ tp^*\ E_p(p^+) : ER(rt) = ER_e(e_1) \textbf{ in } ER_e(e_2) \\
ER_e(\textbf{let } x = e_1 \textbf{ in } e_2) &= \textbf{let } x = ER_e(e_1) \textbf{ in } ER_e(e_2) \\
ER_e(\lambda p^+ \to e) &= \lambda E_p(p^+) \to ER_e(e) \\
ER_e(e_1\, e_2) &= ER(e_1)\, ER(e_2) \\
ER_e(x) &= x \\
ER_e(n) &= n \\
ER_e(x[e]) &= x[ER_e(e)] \\
ER_e(e_1 \oplus e_2) &= ER_e(e_1) \oplus ER_e(e_2) \\
ER_e(e_1\, e_2) &= ER_e(e_1)\, ER_e(e_2) \\
E_p(x\, p^*) &= x\, E_p(p^*) \\
E_p((x : \tau)\, p^*) &= (x : ER(\tau))\, E_p(p^*) \\
E_p(\cdot) &= \cdot \\
ER(\texttt{i32}) &= \texttt{i32} \\
ER(\texttt{i32}(i)) &= \texttt{i32} \\
ER(\texttt{bool}) &= \texttt{bool} \\
ER(\texttt{bool}(i)) &= \texttt{bool} \\
ER([\,]t) &= [\,]ER(t) \\
ER([a]t) &= [a]ER(t) \\
ER([i]t) &= [\,]ER(t) \\
ER(\tau_1 \to \tau_2) &= ER(\tau_1) \to ER(\tau_2) \\
ER(\{t \mid P\}) &= ER(t) \\
ER(\alpha) &= \alpha
\end{aligned}
$$

Figure 7: The erasure functions

5.1 Some examples

The length function in Futhark is given as:

```
1 let length [n] 't (_: [n]t) = n
```

In classic Futhark, there is no way to express in the return type that the number returned is equal to n; we can at best write

```
1 let length [n] 't (_: [n]t) : i32 = n
```

Since arrays carry their size in their type we extend integers to carry their value in their types so the implementation becomes

```
1 let length [n] 't (_: [n]t) : i32(n) = n
```

Which in the internal language becomes

$$\forall n : int. \forall \alpha : \star. \alpha\ array(n) \triangleright (0 \leq n) \rightarrow \texttt{i32}(n)$$

Notice that we do not know that the size of the array returned by `length` is positive. The transformation function does not capture the fact that the index variable n originates from an array. The property must be established somewhere else.

Functions like `iota` and `replicate` reference parameter names as size variables, and since they are parameter names they, are not declared as type parameters in the front of the function; their types are:

```
1 let iota (n: i32): [n]i32 = 0..<n
2 let replicate 't (n: i32) (x: t): [n]t = map (const x) (iota n)
```

We produce the following types for `iota` and `replicate`:

$$\forall n : int.\texttt{i32}(n) \rightarrow \texttt{i32}\ array(n) \triangleright (0 \leq n)$$
$$\forall \alpha : \star. \forall n : int.\texttt{i32}(n) \rightarrow \alpha \rightarrow \alpha\ array(n) \triangleright (0 \leq n)$$

In a sense we 'promote' some variables to the front of the type and bind them in a $\forall$ if they are used in other parts of the type because then they have to be scoped to the whole type.

Note that that the types of `iota` and `replicate` tell us that the array produced will have a positive length. An arguably better approach would be to instead require that the input is positive by using guarded types. To figure out which types should be guarded requires more complicated transformation rules than those presented here.

6 Type checking the Internal Language

In this section, the elements of type checking are presented. We first give the definition of the context and then we briefly introduce the concept of constraints before giving the type rules. Finally, we present some examples of type derivations.

6.1 Context

The context structure can be seen in Figure 8.

Contexts contain information about type variables, index variables, and program variables.

Type variables and index variables can be universal variables $\alpha : \kappa$ or existential variables $\hat{\alpha} : \kappa$, and existential variables may be solved to a particular type $\hat{\alpha} : \kappa = \tau$.

As type variables and index variables share syntax, they are distinguished from each other by their kind κ, where the kind of types is $\star$ and the kind of an index variable is its sort, like for example *int*.

Program variables are mapped to their types $x : A$. Markers $\blacktriangleright_u$ are used to mark a spot in the context that split the context in two $\Gamma, \blacktriangleright_u, \Delta$ so that we can denote the different parts of the context.

Finally the context also contains subtype constraints $A \leq B$.

A context Γ is ordered and objects in the context may only reference variables to the left of the object itself.

We can use the information in the context to substitute into types using the notation $[\Gamma]A$. For example $([\hat{\alpha} : \kappa = \tau]\hat{\alpha}) = \tau$.

Universal variables	α, β		
Existential variables	$\hat{\alpha}, \hat{\beta}$		
Variables	u	::=	$\alpha \mid \hat{\alpha}$
Contexts	Γ, Δ, Θ	::=	$\cdot \mid \Gamma, \alpha : \kappa \mid \Gamma, \hat{\alpha} : \kappa \mid \Gamma, x : A \mid$ $\Gamma, \hat{\alpha} : \kappa = \tau \mid \Gamma, \blacktriangleright_u \mid \Gamma, A \leq B$

Figure 8: Syntax for the context.

6.2 Constraints

The process of type checking generates, if successful, a program and a set of constraints that has to be solved to find out if the program is correctly typed. The constraints are inequality constraints.

If a type derivation can be constructed for an expression the process will generate a set of inequality constraints. Only if the constraints are proven to be always satisfied are we sure that the program is safe.

The judgement $\Gamma \vdash P\ true$ that is used in Figure 12 and Figure 13 is not derived but used to construct a constraint that is collected into a constraint set that is checked by a solver. To produce the constraint to be satisfied when our derivation includes $\Gamma \vdash P_0\ true$ we use the following relation

$$\begin{aligned}\Phi(P, \Gamma) &= P \supset \Phi(\Gamma) \\ \Phi(\forall a : \gamma, \Gamma) &= \forall a : \gamma.\Phi(\Gamma) \\ \Phi(\cdot) &= P_0 \\ \Phi(\_, \Gamma) &= \Phi(\Gamma)\end{aligned}$$

Using this relation naively we can construct a set of constraint or one big constraints using conjunction: $(\forall a : \gamma.P \supset P_1) \wedge (\forall a : \gamma.P \supset P_2)$. In a practical implementation we should instead construct these constraints so that they can share prefixes and produce constraints on the form $\forall a : \gamma.P \supset (P_1 \wedge P_2)$ which is equivalent to $(\forall a : \gamma.P \supset P_1) \wedge (\forall a : \gamma.P \supset P_2)$.

6.3 Judgements and rules

The rules use the bidirectional type checking approach and as such we have two mutually recursive judgements for type synthesis and type checking. Each syntactic form has either a checking rule or a synthesis rule but not both. The type checking and type synthesis rules are mutually recursive. Also there is a mutually dependent cycle between the check, synthesis and spine rules. Most of the judgements and many of the rules are not new [10].

The connection between the judgements can be seen in Figure 10. There is an arrow from judgement A to judgement B if there is a rule of judgement A that uses judgement B.

6.3.1 Checking rules

The judgement $\Gamma \vdash e \Leftarrow A \dashv \Delta$ means that the expression e is checked against type A in context Γ and the checking produces a context Δ. The rules can be seen in Figure 12.

The Sub rule is identical to that of [10]. It is the rule that allows subtyping to be integrated. We will see later how big an effect that will have. The rule also has the role of changing the direction of type checking when we want to check terms that are synthesizing. The subtype relation is polarized, which will be explained later. It is through the subtyping rules we instantiate existential variables.

Note that the rules $\forall I$, $\exists I$, $\supset I$ and $\triangleright$I unlike the other rules are used for multiple syntactic constructs. Those constructs are the checked introduction forms and e *chk-I* means that expression e is a checked introduction form; right now these are the lambdas and pairs. A let-expression is a checked form but not an introduction form. If the construct is a synthesizing form, the Sub rule is used instead.

6.3.2 Synthesizing rules

The judgement $\Gamma \vdash e \Rightarrow A \dashv \Delta$ means that the type A is synthesized for expression e with output context Δ. The rules are deterministic and typechecking starts either in the checking or the synthesis rules depending on the syntax of the expression. The rules can be seen in Figure 11.

The Anno rule, like the Sub rule, changes direction of type checking. In the Anno rule, we switch from synthesis to checking. The Anno rule supports the intuition that we can tell the type system what type a term should have by giving it a type annotation.

6.3.3 Spine rules

To synthesize a type for function application, we use the spine judgement to check types of function arguments. The judgement $\Gamma \vdash s : A \gg C \dashv \Delta$ means that applying a function of type A to a spine s synthesizes the type C with output context Δ. The rules can be found in Figure 13. In the spine rule $\rightarrow$`Spine`, the arguments are checked using the checking judgement.

$$
\begin{array}{lll}
\mathrm{pol}(\forall\alpha:\kappa.A) & = - & \qquad \mathrm{join}(+,\mathcal{P}) = + \\
\mathrm{pol}(\exists\alpha:\kappa.A) & = + & \qquad \mathrm{join}(-,\mathcal{P}) = - \\
\mathrm{pol}(A \triangleright P) & = + & \qquad \mathrm{join}(\circ,\mathcal{P}) = \mathcal{P} \\
\text{otherwise} & = \circ &
\end{array}
$$

Figure 9: The join and pol functions

6.3.4 Let-binding checking rules

The `let` construct has the role of unpacking existential and asserting types. The idea of limiting the scope of existentially quantified type indices to the scope of the variable of the indexed type is not new [10,25]. It has been done with the `case` construct and the `let` construct.

There are no `case` expressions in the language but let bindings are a special form of `case` with one branch thus we used them for existial unpacking.

Having let bindings in their own judgement aids the readability of both check and let-bindings by seperating them. The judgement $\Gamma \vdash x : A$ **in** $e \ll C \dashv \Delta$ means that expression e is checked against type C in context Γ while x has type A. The rules for let-bindings can be seen in Figure 18.

The Let$\exists$ rule deconstructs an existential type while making sure the (index/type) variable is only scoped in the subderivation, thereby controlling the scope of the existential. The Let$\triangleright$ rule deconstructs an asserting type and in the same way as before making sure the proposition is only assumed in the subderivation. Finally, the LetBase rule is used when there are no more existential or asserting types. The let-bound variable is put into context while the expression is checked against the type. This rule is similar to how a regular let-rule looks like.

6.3.5 Subtyping rules

$\Gamma \vdash A \leq^{\mathcal{P}} B \dashv \Delta$ means that A is a subtype of B in context Γ with output context Δ. The rules can be seen in Figure 14. The subtype relation is polarized, which means that it is split into two mutually recursive subtype relations, $\leq^{+}$ and $\leq^{-}$. When used from the Sub rule, the polarity of the subtype relation is determined by the polarity of the expression $\mathrm{join}(\mathrm{pol}(B), \mathrm{pol}(A))$. The join and pol functions can be seen in Figure 9. The join allows the programmer to chose which subtype relation to use by using a type annotation. The choice of subtype relation determines whether to deconstruct the

universal or the existential quantifiers first. If the $\leq^-$ relation is used, all universal quantifiers are deconstructed first, before the relation is switched to $\leq^+$ if there are any existential quantifiers. The opposite is true for the $\leq^+$ relation, for which all existential quantifiers are deconstructed first. If the types are neither positive or negative, they have to be equivalent according to the equivalence rules.

This notation comes from the fact that existential types are positive types in logic and universal types are negative [10]. Furthermore, asserting types are positive and guarded types are negative. We also give asserting type rules to support cases like $\exists n : \mathit{int}.\mathtt{i32}(n) \triangleright (n < 5) \leq^+ \exists n : \mathit{int}.\mathtt{i32}(n) \triangleright (n < 6)$ to support array indexing.

An exception for subtyping is made for existential variables, for which subtype constraints are put into the context and later checked in their own rules $\lceil \hat{\alpha} \rceil$. We don't know if the variable will later be instantiated for a positive or negative type so we delay subtype checking.

Note that array types do not have a subtype rule. The reason for this is variance. Since we do not want to deal with variance, array types only have an equivalence rule.

6.3.6 Type equivalence and related rules

The judgement $\Gamma \vdash A \equiv B \dashv \Delta$ means that the types A and B are equivalent in context Γ with output context Δ. The rules can be seen in Figure 15.

The $\delta\equiv$ rule reduces type equivalence to index equality $\stackrel{\circ}{=}$. Since index equality is decidable it means that type equivalence is decidable. $S(\delta)$ is the signature of the type family and gives the sort of the the index, e.g. we have $S(\mathtt{i32}) = \mathit{int} \to \star$.

When an existential variable $\hat{\alpha}$ is encountered, the instantiation rules are used to instantiate the variable to the other type.

The judgement $\Gamma \vdash i \stackrel{\circ}{=} j \dashv \Delta$ means that the indices i and j are equal. The rules can be seen in Figure 16. If any of the indices are existential variables, the instantiation rules are used just like for types. Note that we do not have a rule for the case of $f(i) \stackrel{\circ}{=} g(j)$ because we currently do not support the kind of constraints the rule would require us to solve. The case $f(i) \stackrel{\circ}{=} \hat{\alpha}$ and $\hat{\alpha} \stackrel{\circ}{=} f(i)$ are handled by the instantiation rules.

The judgement $\Gamma \vdash \hat{\alpha} := \tau : \kappa \dashv \Delta$ means that the existential variable $\hat{\alpha}$

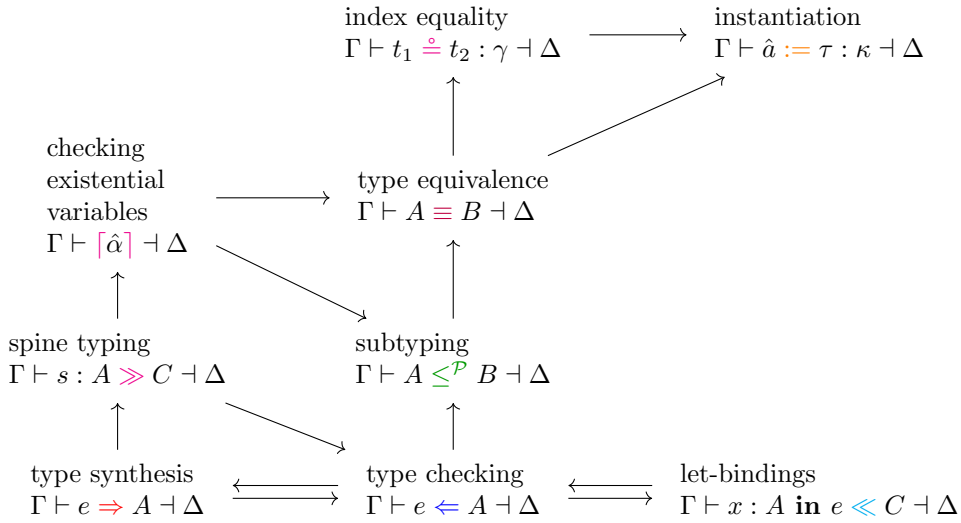

Figure 10: Dependencies between the type judgements.

should be instantiated to the type τ. The rules can be seen in Figure 17.

6.3.7 Existential variable subtype checking rules

The judgement $\Gamma \vdash \lceil \hat{\alpha} \rceil \dashv \Delta$ means that all subtype constraints of the form $A \leq \hat{\alpha}$ and $\hat{\alpha} \leq A$ in context Γ are checked. This judgment is original and is needed to support the case where an existential variable might later be instantiated to an existentially quantified type. The rules can be seen in Figure 19.

If $\hat{\alpha}$ is unsolved then we find the first subtype constraint $\hat{\alpha} \leq A$ or $A \leq \hat{\alpha}$ after the variable is declared and require $\hat{\alpha} \equiv A$, which ends up using the instantiation rules to instantiate $\hat{\alpha}$ to A. If $\hat{\alpha}$ is solved to type A and there still is a subtype relation $\hat{\alpha} \leq B$ in the context, then we must derive $A \leq B$.

$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$ type A is synthesized for expression e with output context Δ.

$$\frac{}{\Gamma \vdash n \Rightarrow \texttt{i32}(n) \dashv \Gamma}\ \text{Num}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow [\Gamma]A \dashv \Gamma}\ \text{Var}$$

$$\frac{\Gamma \vdash A\,\text{type} \qquad \Gamma \vdash e \Leftarrow [\Gamma]A \dashv \Delta}{\Gamma \vdash (e : A) \Rightarrow [\Delta]A \dashv \Delta}\ \text{Anno}$$

$$\frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash s : A \gg C \dashv \Delta}{\Gamma \vdash e\,s \Rightarrow C \dashv \Delta}\ {\to}E$$

Figure 11: Type synthesis rules

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ expression e is checked against type A in context Γ producing context Δ

$$\frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash A \leq^{\text{join}(\text{pol}(B),\text{pol}(A))} B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \text{Sub}$$

$$\frac{v\ \textit{chk-I} \qquad \Gamma, \alpha : \kappa \vdash v \Leftarrow A \dashv \Delta, \alpha : \kappa, \Theta}{\Gamma \vdash v \Leftarrow \forall \alpha : \kappa . A \dashv \Delta}\ \forall\text{I}$$

$$\frac{e\ \textit{chk-I} \qquad \Gamma, \alpha : \kappa \vdash e \Leftarrow [\hat{\alpha}/\alpha]A \dashv \Delta}{\Gamma \vdash e \Leftarrow \exists \alpha : \kappa . A \dashv \Delta}\ \exists\text{I}$$

$$\frac{v\ \textit{chk-I} \qquad \Gamma, \blacktriangleright_p, P \dashv \Theta \qquad \Theta \vdash v \Leftarrow [\Theta]A \dashv \Delta, \blacktriangleright_p, \Delta'}{\Gamma \vdash v \Leftarrow P \supset A \dashv \Delta}\ \supset\text{I}$$

$$\frac{e \text{ not a let} \qquad \Gamma \vdash P \text{ true} \dashv \Theta \qquad \Theta \vdash e \Leftarrow [\Theta]A \dashv \Delta}{\Gamma \vdash e \Leftarrow A \triangleright P \dashv \Delta}\ \triangleright\text{I}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.e \Leftarrow A \to B \dashv \Delta}\ \to\text{I}$$

$$\frac{\Gamma[\hat{\alpha}_1 : \star, \hat{\alpha}_2 : \star, \hat{\alpha} : \star = \hat{\alpha}_1 \to \hat{\alpha}_2], x : \hat{\alpha}_1 \vdash e \Leftarrow \hat{\alpha}_2 \dashv \Delta, x : \hat{\alpha}_1, \Delta'}{\Gamma[\hat{\alpha} : \star] \vdash \lambda x.e \Leftarrow \hat{\alpha} \dashv \Delta}\ \to\text{I}\hat{\alpha}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow A_1 \dashv \Theta \qquad \Theta \vdash e_2 \Leftarrow [\Theta]A_2 \dashv \Delta}{\Gamma \vdash \langle e_1, e_2 \rangle \Leftarrow A_1 \times A_2 \dashv \Delta}\ \times\text{I}$$

$$\frac{\Theta \vdash e_2 \Leftarrow [\Theta]\hat{\alpha}_2 \dashv \Delta \qquad \Gamma[\hat{\alpha}_1 : \star, \hat{\alpha}_2 : \star, \hat{\alpha} : \star = \hat{\alpha}_1 \times \hat{\alpha}_2] \vdash e_1 \Leftarrow \hat{\alpha}_1 \dashv \Theta}{\Gamma[\hat{\alpha} : \star] \vdash \langle e_1, e_2 \rangle \Leftarrow \hat{\alpha} \dashv \Delta}\ \times\text{I}\hat{\alpha}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta \qquad \Theta \vdash x : A \textbf{ in } e_2 \ll C \dashv \Delta}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \Leftarrow C \dashv \Delta}\ \texttt{let}$$

Figure 12: Typechecking rules

$\boxed{\Gamma \vdash s : A \gg C \dashv \Delta}$ applying a function of type A to a spine s synthesizes the type C with output context Δ

$$\frac{\Gamma, \hat{\alpha} : \kappa \vdash e\ s : [\hat{\alpha}/\alpha]A \gg C \dashv \Theta \qquad \Theta \vdash \lceil \hat{\alpha} \rceil \dashv \Delta}{\Gamma \vdash e\ s : \forall \alpha : \kappa . A \gg [\Delta]C \dashv \Delta}\ \forall\text{Spine}$$

$$\frac{\Gamma \vdash e\ s : A \gg C \dashv \Theta \qquad \Theta \vdash [\Theta]P\ \text{true} \dashv \Delta}{\Gamma \vdash e\ s : P \supset A \gg C \dashv \Delta}\ \supset\text{Spine}$$

$$\frac{}{\Gamma \vdash \cdot : A \gg A \dashv \Gamma}\ \text{EmptySpine}$$

$$\frac{\Gamma \vdash e \Leftarrow A \dashv \Theta \qquad \Theta \vdash s : [\Theta]B \gg C \dashv \Delta}{\Gamma \vdash e\ s : A \to B \gg C \dashv \Delta}\ \to\text{Spine}$$

$$\frac{\Gamma[\hat{\alpha}_1 : \star, \hat{\alpha}_2 : \star, \hat{\alpha} : \star = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash e\ s : (\hat{\alpha}_1 \to \hat{\alpha}_2) \gg C \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash e\ s : \hat{\alpha} \gg C \dashv \Delta}\ \hat{\alpha}\text{Spine}$$

Figure 13: Spine typing rules

$\boxed{\Gamma \vdash A \leq^P B \dashv \Delta}$ A is a subtype of B under context Γ

$$\frac{A, B \text{ not headed by } \forall/\exists \text{ or a } \hat{\alpha} \qquad \Gamma \vdash A \equiv B \dashv \Delta}{\Gamma \vdash A \leq^{\mathcal{P}} B \dashv \Delta} \leq\text{Equiv}$$

$$\frac{}{\Gamma[\hat{\alpha} : \star][\hat{\beta} : \star] \vdash \hat{\alpha} \leq^{\mathcal{P}} \hat{\beta} \dashv \Gamma[\hat{\alpha} : \star, \hat{\alpha} \leq \hat{\beta}][\hat{\beta} : \star]} \leq\hat{\alpha}\hat{\beta}\text{L}$$

$$\frac{}{\Gamma[\hat{\alpha} : \star][\hat{\beta} : \star] \vdash \hat{\beta} \leq^{\mathcal{P}} \hat{\alpha} \dashv \Gamma[\hat{\alpha} : \star, \hat{\beta} \leq \hat{\alpha}][\hat{\beta} : \star]} \leq\hat{\alpha}\hat{\beta}\text{R}$$

$$\frac{A \text{ is not } \hat{\beta}}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} \leq^{\mathcal{P}} A \dashv \Gamma[\hat{\alpha} : \star, \hat{\alpha} \leq A]} \leq\hat{\alpha}\text{L}$$

$$\frac{A \text{ is not } \hat{\beta}}{\Gamma[\hat{\alpha} : \star] \vdash A \leq^{\mathcal{P}} \hat{\alpha} \dashv \Gamma[\hat{\alpha} : \star, A \leq \hat{\alpha}]} \leq\hat{\alpha}\text{R}$$

$$\frac{\text{B not headed by } \forall \qquad \Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha} : \kappa \vdash [\hat{\alpha}/\alpha]A \leq^{-} B \dashv \Delta \blacktriangleright_{\hat{\alpha}}, \Theta}{\Gamma \vdash \forall \alpha : \kappa . A \leq^{-} B \dashv \Delta} \leq\forall\text{L}$$

$$\frac{\Gamma, \beta : \kappa \vdash A \leq^{-} B \dashv \Delta, \beta : \kappa, \Theta}{\Gamma \vdash A \leq^{-} \forall \beta : \kappa . B \dashv \Delta} \leq\forall\text{R}$$

$$\frac{\Gamma, \alpha : \kappa \vdash A \leq^{+} B \dashv \Delta, \alpha : \kappa, \Theta}{\Gamma \vdash \exists \alpha : \kappa . A \leq^{+} B \dashv \Delta} \leq\exists\text{L}$$

$$\frac{\text{A not headed by } \exists \qquad \Gamma, \blacktriangleright_{\hat{\beta}}, \hat{\beta} : \kappa \vdash A \leq^{+} [\hat{\beta}/\beta]B \dashv \Delta \blacktriangleright_{\hat{\beta}}, \Theta}{\Gamma \vdash A \leq^{+} \exists \beta : \kappa . B \dashv \Delta} \leq\exists\text{R}$$

$$\frac{\Gamma, \blacktriangleright_P, P \vdash A \leq^{+} B \dashv \Delta, \blacktriangleright_P, \Theta}{\Gamma \vdash A \triangleright P \leq^{+} B \dashv \Delta} \leq\triangleright\text{L}$$

$$\frac{nonpos(A) \qquad \Gamma \vdash P \ true \qquad \Gamma \vdash A \leq^{+} B \dashv \Delta}{\Gamma \vdash A \leq^{+} B \triangleright P \dashv \Delta} \leq\triangleright\text{R}$$

$$\frac{\Gamma \vdash A \leq^{-} B \dashv \Delta \qquad (neg(A) \wedge nonpos(B)) \vee (nonpos(A) \wedge neg(B))}{\Gamma \vdash A \leq^{+} B \dashv \Delta} \leq -+$$

$$\frac{\Gamma \vdash A \leq^{+} B \dashv \Delta \qquad (pos(A) \wedge nonneg(B)) \vee (nonneg(A) \wedge pos(B))}{\Gamma \vdash A \leq^{-} B \dashv \Delta} \leq +-$$

Figure 14: Subtyping rules

$\boxed{\Gamma \vdash A \equiv B \dashv \Delta}$ types A and B are equivalent in context Γ with output context Δ.

$$\frac{}{\Gamma \vdash \alpha \equiv \alpha} \equiv\text{Var}$$

$$\frac{}{\Gamma \vdash \hat{\alpha} \equiv \hat{\alpha}} \equiv\text{Exvar}$$

$$\frac{S(\delta) = \gamma \to \star \qquad \Gamma \vdash i \stackrel{\circ}{=} j : \gamma \dashv \Delta}{\Gamma \vdash \delta(i) \equiv \delta(j) \dashv \Delta} \equiv\delta$$

$$\frac{\Gamma \vdash i \stackrel{\circ}{=} j : \mathit{int} \dashv \Theta \qquad \Gamma \vdash [\Theta]A \equiv [\Theta]B \dashv \Delta}{\Gamma \vdash A\ \mathit{array}(i) \equiv B\ \mathit{array}(j) \dashv \Delta} \equiv\text{array}$$

$$\frac{\Gamma \vdash A_1 \equiv B_1 \dashv \Theta \qquad \Gamma \vdash [\Theta]A_2 \equiv [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 \equiv B_1 \to B_2 \dashv \Delta} \equiv\to$$

$$\frac{\Gamma, \alpha : \kappa \vdash A \equiv B \dashv \Delta, \alpha : \kappa, \Delta'}{\Gamma \vdash (\forall \alpha : \kappa . A) \equiv (\forall \alpha : \kappa . B) \dashv \Delta} \equiv\forall$$

$$\frac{\Gamma, \alpha : \kappa \vdash A \equiv B \dashv \Delta, \alpha : \kappa, \Delta'}{\Gamma \vdash (\exists \alpha : \kappa . A) \equiv (\exists \alpha : \kappa . B) \dashv \Delta} \equiv\exists$$

$$\frac{\hat{\alpha} \notin FV(\tau) \qquad \Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} := \tau : \star \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} \equiv \tau \dashv \Delta} \equiv{:=}\text{L}$$

$$\frac{\hat{\alpha} \notin FV(\tau) \qquad \Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} := \tau : \star \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \tau \equiv \hat{\alpha} \dashv \Delta} \equiv{:=}\text{R}$$

Figure 15: Type equivalence rules

$\boxed{\Gamma \vdash t_1 \mathrel{\mathring{=}} t_2 : \gamma \dashv \Delta}$ means that the indices i and j are equal

$$\frac{}{\Gamma \vdash n \mathrel{\mathring{=}} n : int \dashv \Gamma} \mathrel{\mathring{=}}\text{INum}$$

$$\frac{}{\Gamma \vdash a \mathrel{\mathring{=}} a : \gamma \dashv \Gamma} \mathrel{\mathring{=}}\text{IVar}$$

$$\frac{}{\Gamma \vdash \hat{\alpha} \mathrel{\mathring{=}} \hat{\alpha} : \gamma \dashv \Gamma} \mathrel{\mathring{=}}\text{ExIVar}$$

$$\frac{\hat{\alpha} \notin FV(i) \qquad \Gamma[\hat{\alpha} : \gamma] \vdash \hat{\alpha} := i : \gamma \dashv \Delta}{\Gamma[\hat{\alpha} : \gamma] \vdash \hat{\alpha} \mathrel{\mathring{=}} i \dashv \Delta} \mathrel{\mathring{=}}{:=}\text{L}$$

$$\frac{\hat{\alpha} \notin FV(i) \qquad \Gamma[\hat{\alpha} : \gamma] \vdash \hat{\alpha} := i : \gamma \dashv \Delta}{\Gamma[\hat{\alpha} : \gamma] \vdash i \mathrel{\mathring{=}} \hat{\alpha} \dashv \Delta} \mathrel{\mathring{=}}{:=}\text{R}$$

Figure 16: Index equality rules

$\boxed{\Gamma \vdash \hat{\alpha} := \tau : \kappa \dashv \Delta}$ the existential variable $\hat{\alpha}$ should be instantiated to the type τ

$$\frac{\Gamma_0 \vdash \tau : \kappa}{\Gamma_0, \hat{\alpha} : \kappa, \Gamma_1 \vdash \hat{\alpha} := \tau : \kappa \dashv \Gamma_0, \hat{\alpha} : \kappa = \tau, \Gamma_1} :=\text{Solve}$$

$$\frac{\hat{\beta} \in \text{unsolved}(\Gamma[\hat{\alpha} : \kappa][\hat{\beta} : \kappa])}{\Gamma[\hat{\alpha} : \kappa][\hat{\beta} : \kappa] \vdash \hat{\alpha} := \hat{\beta} : \kappa \dashv \Gamma[\hat{\alpha} : \kappa][\hat{\beta} : \kappa = \hat{\alpha}]} :=\text{Reach}$$

$$\frac{\Theta \vdash \hat{\alpha}_2 := [\Theta]\tau_2 : \star \dashv \Delta \qquad \Gamma[\hat{\alpha}_2 : \star, \hat{\alpha}_1 : \star, \hat{\alpha} : \star = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash \hat{\alpha}_1 := \tau_1 : \star \dashv \Theta}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} := \tau_1 \to \tau_2 : \star \dashv \Delta} :=\to$$

$$\frac{f : \gamma_1 \to \gamma_2 \qquad \Gamma[\hat{\alpha}_1 : \gamma_1, \hat{\alpha} : \gamma_2 = f(\hat{\alpha}_1)] \vdash \hat{\alpha}_1 := i : \gamma_1 \dashv \Delta}{\Gamma[\hat{\alpha} : \gamma_2] \vdash \hat{\alpha} := f(i) : \gamma_2 \dashv \Delta} :=\text{IFun}$$

$$\frac{}{\Gamma[\hat{\alpha} : int] \vdash \hat{\alpha} := n : int \dashv \Gamma[\hat{\alpha} : int = n]} :=\text{Num}$$

Figure 17: Instantiation rules

$\boxed{\Gamma \vdash x : A \textbf{ in } e \ll C \dashv \Delta}$ expression e is checked against type C in context Γ while x has type A.

$$\frac{\text{A not headed by } \triangleright \text{ or } \exists \qquad \Gamma, x : A \vdash e \Leftarrow C \dashv \Delta, x : A, \Delta'}{\Gamma \vdash x : A \textbf{ in } e \ll C \dashv \Delta} \text{LetBase}$$

$$\frac{\Gamma, \alpha : \kappa \vdash x : A \textbf{ in } e \ll C \dashv \Delta, \alpha : \kappa, \Theta}{\Gamma \vdash x : (\exists \alpha : \kappa . A) \textbf{ in } e \ll C \dashv \Delta} \text{Let}\exists$$

$$\frac{\Gamma, \blacktriangleright_P, P \dashv \Theta \qquad \Theta \vdash x : A \textbf{ in } e \ll C \dashv \Delta, \blacktriangleright_P, \Delta'}{\Gamma \vdash x : A \triangleright P \textbf{ in } e \ll C \dashv \Delta} \text{Let}\triangleright$$

Figure 18: Rules for let-binding.

$\boxed{\Gamma \vdash \lceil \hat{\alpha} \rceil \dashv \Delta}$ Under context Γ check that all subtype constraints involving $\hat{\alpha}$ in Γ are satisfied

$$\frac{\hat{\alpha} \text{ not followed by } \hat{\alpha} \leq A \text{ or } A \leq \hat{\alpha} \text{ in } \Gamma}{\Gamma \vdash \lceil \hat{\alpha} \rceil \dashv \Gamma} \hat{\alpha}\leq\text{Base}$$

$$\frac{\Gamma[\hat{\alpha} : \star] \vdash A \equiv \hat{\alpha} \dashv \Theta \qquad \Theta \vdash \lceil \hat{\alpha} \rceil \dashv \Delta}{\Gamma[\hat{\alpha} : \star, A \leq \hat{\alpha}] \vdash \lceil \hat{\alpha} \rceil \dashv \Delta} \hat{\alpha}\leq\text{Unsolved}$$

$$\frac{\Gamma[\hat{\alpha} : \star = B] \vdash A \leq B \dashv \Theta \qquad \Theta \vdash \lceil \hat{\alpha} \rceil \dashv \Delta}{\Gamma[\hat{\alpha} : \star = B, A \leq \hat{\alpha}] \vdash \lceil \hat{\alpha} \rceil \dashv \Delta} \hat{\alpha}\leq\text{Solved}$$

Figure 19: Rules for checking subtype constraints for existential variables.

6.4 Example of type derivations

We illustrate the type rules with some type derivations.

6.4.1 replicate

Suppose we have a function `replicate`, defined in a context Γ with the type

$$\forall\alpha : \star.\forall n : int.\texttt{i32}(n) \to \alpha \to \alpha\ array(n) \triangleright (0 \leq n)$$

We expect the expression `replicate 5 0` to be a subtype of $(\texttt{i32}(0))\ array(5)$.

We use the $\to E$ rule, which as a premise uses the spine rules and we show how the type rules derive:

$$\Gamma \vdash (5\,(0\,\cdot)) : A \gg (\texttt{i32}(0))\ array(5) \triangleright (0 \leq 5) \dashv \Gamma, \hat{\alpha} : \star = \texttt{i32}(0), \hat{\beta} : int = 5$$

where A is the type of `replicate`.

First the $\forall$Spine rule is applied and it substitutes α for a new existential variable $\hat{\alpha}$ which represents the instantiation of the universal variable α. The new existential variable is added to the context. We now must derive

$$\Gamma, \hat{\alpha} : \star \vdash (5\,(0\,\cdot)) : \forall n : int.\texttt{i32}(n) \to \hat{\alpha} \to \hat{\alpha}\ array(n) \triangleright (0 \leq n) \gg C \dashv \Gamma_2$$

and construct the derivation $\mathcal{D}_1$ of $\Gamma_2 \vdash \lceil\hat{\alpha}\rceil \dashv \Gamma_3$. The $\forall$Spine rule is used a second time and our new goals are

$$\Gamma, \hat{\alpha} : \star, \hat{\beta} : int \vdash (5\,(0\,\cdot)) : \texttt{i32}(\hat{\beta}) \to \hat{\alpha} \to \hat{\alpha}\ array(\hat{\beta}) \triangleright (0 \leq \hat{\beta}) \gg C \dashv \Gamma_4$$

and the derivation $\mathcal{D}_2$ of $\Gamma_4 \vdash \lceil\hat{\beta}\rceil \dashv \Gamma_2$.

Now the $\to$Spine rule is applied and we derive (details omitted)

$$\Gamma, \hat{\alpha} : \star, \hat{\beta} : int \vdash 5 \Leftarrow \texttt{i32}(\hat{\beta}) \dashv \Gamma, \hat{\alpha} : \star, \hat{\beta} = 5 : int$$

and must now derive

$$\Gamma, \hat{\alpha} : \star, \hat{\beta} = 5 : int \vdash (0\,\cdot) : \hat{\alpha} \to \hat{\alpha}\ array(5) \triangleright (0 \leq 5) \gg C \dashv \Gamma_4$$

since $[\Gamma, \hat{\alpha} : \star, \hat{\beta} = 5 : int](\hat{\alpha}\ array(\hat{\beta}) \triangleright (0 \leq \hat{\beta})) = \hat{\alpha}\ array(5) \triangleright (0 \leq 5)$.

We apply the →Spine rule one more time and we derive (details omitted)

$$\Gamma, \hat{\alpha} : \star, \hat{\beta} : \mathit{int} = 5 \vdash 0 \Leftarrow \hat{\alpha} \dashv \Gamma, \hat{\alpha} : \star, \mathtt{i32}(0) \leq \hat{\alpha}, \hat{\beta} = 5 : \mathit{int}$$

and must derive

$$\Gamma, \hat{\alpha} : \star = \mathtt{i32}(0), \hat{\beta} = 5 : \mathit{int} \vdash \cdot : \hat{\alpha}\ \mathit{array}(5) \triangleright (0 \leq 5) \gg C \dashv \Gamma_4$$

The only spine rule left to apply is the EmptySpine rule, and after applying it we have that

$$C = \hat{\alpha}\ \mathit{array}(5) \triangleright (0 \leq 5)$$

and

$$\Gamma_4 = \Gamma, \hat{\alpha} : \star, \mathtt{i32}(0) \leq \hat{\alpha}, \hat{\beta} = 5 : \mathit{int}$$

We now construct $\mathcal{D}_2$ using the $\hat{\alpha}\leq$Base rule so we derive $\Gamma_4 \vdash \lceil \hat{\beta} \rceil \dashv \Gamma_4$ and so we have $\Gamma_2 = \Gamma_4$.

To construct $\mathcal{D}_1$ we use the $\hat{\alpha}\leq$Unsolved rule and derive (details omitted)

$$\Gamma, \hat{\alpha} : \star, \mathtt{i32}(0) \leq \hat{\alpha}, \hat{\beta} = 5 : \mathit{int} \vdash \mathtt{i32}(0) \equiv \hat{\alpha} \dashv \Gamma, \hat{\alpha} : \star = \mathtt{i32}(0), \hat{\beta} = 5 : \mathit{int}$$

and use the $\hat{\alpha}\leq$Base rule to derive

$$\Gamma, \hat{\alpha} : \star = \mathtt{i32}(0), \hat{\beta} = 5 : \mathit{int} \vdash \lceil \hat{\beta} \rceil \dashv \Gamma, \hat{\alpha} : \star = \mathtt{i32}(0), \hat{\beta} = 5 : \mathit{int}$$

so $\Gamma_3 = \Gamma, \hat{\alpha} : \star = \mathtt{i32}(0), \hat{\beta} = 5 : \mathit{int}$

Finally we substitute $[\Gamma, \hat{\alpha} : \star = \mathtt{i32}(0), \hat{\beta} = 5 : \mathit{int}](\hat{\alpha}\ \mathit{array}(5) \triangleright (0 \leq 5)$ and get $\mathtt{i32}(0)\ \mathit{array}(5) \triangleright (0 \leq 5)$.

6.4.2 segmented_replicate

We now turn our attention to the main example of this thesis. Checking the type of `segmented_replicate` involves using a refined array element type with `map` and making use of the refinement inside of the functional argument to `map`. The most interesting part is the derivation of

$$\Gamma \vdash \mathit{map}\,(\lambda i \to \mathit{vs}[i])\ \mathit{idxs} \Rightarrow C \dashv \Delta$$

of which the main part is the derivation of

$$\Gamma \vdash (\lambda i \to \mathit{vs}[i])\ \mathit{idxs} : A \gg C \dashv \Delta$$

where A is the type of `map`

$$\forall n : int.\forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha\ array(n) \to \beta\ array(n)$$

We also assume that we have a variable vs and a variable $idxs$ defined in the context Γ, with the types $\texttt{i32}\ array(n)$ and $\texttt{fin}(n)\ array(l)$ where $\texttt{fin}(n)$ is a shorthand for $\exists m : int.\texttt{i32}(m) \triangleright (0 \leq m \wedge m < n)$. The $\forall$Spine is used to replace the universal variable with existential ones and turn the type of map into a function type. Then $\to$Spine is used and as the first premise we construct a derivation

$$\dfrac{\dfrac{\mathcal{D}_1 \qquad\qquad \mathcal{D}_2}{\Gamma_1, i : \hat{\alpha} \vdash vs[i] \Rightarrow \texttt{i32} \dashv \Gamma_3 \qquad \Gamma_3 \vdash \texttt{i32} \leq \hat{\beta} \dashv \Gamma_2}\,\text{Sub}}{\dfrac{\Gamma_1, i : \hat{\alpha} \vdash vs[i] \Leftarrow \hat{\beta} \dashv \Gamma_2, i : \hat{\alpha}, \Delta}{\Gamma_1, \vdash \lambda i \to vs[i] \Leftarrow \hat{\alpha} \to \hat{\beta} \dashv \Gamma_2}\,\to\text{I}}$$

where $\Gamma_2 = \Gamma_3[\hat{\alpha} : \star, \texttt{i32} \leq \hat{\beta}]$. with subderivations $\mathcal{D}_1$ and $\mathcal{D}_2$. We turn our attention to $\mathcal{D}_1$, since $vs[i]$ is just syntactic sugar for .index $vs\,i$ we construct the derivation

$$\dfrac{\dfrac{\mathcal{D}_3}{\Gamma_1 \vdash (vs\,(i\,\cdot)) : A \gg \texttt{i32} \dashv \Gamma_3}}{\Gamma_1, i : \hat{\alpha} \vdash .\text{index}\ vs\,i \Rightarrow \texttt{i32} \dashv \Gamma_3}\,\to\text{E}$$

With subderivation $\mathcal{D}_3$ and where A is the type of .index which we assume is defined in the context

$$.\text{index} : \forall\alpha : \star.\forall n : int.\alpha\ array(n) \to \texttt{fin}(n) \to \alpha$$

We omit the subderivation $\mathcal{D}_3$ here but note that $\Gamma_3 = \Gamma_4[\hat{\alpha} : \star, \hat{\alpha} \leq \texttt{fin}(n)]$ where $\Gamma_3 = \Gamma_1, i : \hat{\alpha}$.

Crucially the subtype constraint is checked after $\hat{\alpha}$ is solved to $\texttt{fin}(n)$ by unification with the element type of the idxs array.

6.5 Properties

If a program typechecks in the internal language, the erasure of the corresponding external language program typechecks in Futhark.

Let e_x be a program in the external language. If $T[e_x] = e$ and $\cdot \vdash e \Leftarrow \texttt{i32} \dashv \Delta$ and the contraints produced in the type checking can be solved by

a solver, then there exists a Futhark program e_f such that $\text{ER}[e_x] = e_f$ and $\vdash_f e_f : \texttt{i32}$.

If a program typechecks in the internal language, the erasure of the corresponding external language program never accesses an array out of bounds.

Let e_x be a program in the external language. If $T[e_x] = e$ and $\cdot \vdash e \Leftarrow \texttt{i32} \dashv \cdot$ and the contraints produced in the type checking can be solved by a solver, then if $e_f \hookrightarrow_f v$ then $v \neq \textit{indexError}$. We do not give the semantics of Futhark here, but refer the reader to previous work [18].

7 Implementation

In this section, a Haskell implementation of the type rules is presented. This typechecker is divided into four parts: a parser of the external language, a function that transforms a program in the external language to a program in the internal language, a checker that typechecks a program in the internal language and a pretty printer that outputs a Futhark program where all the refined type annotations are erased. These parts are implemented in distinct modules.

7.1 Imports

Our implementation supports `import` statements to make it easier to use code written in Futhark together with code with refinement types.

When finding an `import` statement, the Futhark compiler looks for a file with the .fut extension. Our tool smilarly looks for a file, but with an .rfut extension. The reason for this is that it enables us to give type definitions for functions even though our implementation does not support the Futhark implementation of the function body. Also we can give type definitions with extended annotations in the .rfut file that the Futhark compiler cannot understand. It is up to the programmer to make sure the type annotations in the .rfut file matches the names of the functions in the .fut file.

7.2 How to use the tool

The tool can be compiled and installed using stack (`https://www.haskellstack.org`), by executing the command

```
stack install
```

in the root directory of the project. The source code is available at `https://github.com/akiehn/refinedfuthark`.

Our program takes a file as input, and if type checking succeeds, outputs a Futhark program to standard output. If type checking fails, an error message is written to standard error. The resulting Futhark program can also be written to a file using the `-o` flag. Assuming the current directory contains a file *segrep.rfut* containing

```
import "repiota"

let segmented_replicate [n] (reps : [n]i32) (vs : [n]i32): []i32 =
  let idxs = replicated_iota reps
  in map (\i -> vs[i]) idxs

let main [n] (reps : [n]i32) (vs : [n]i32): []i32 =
  segmented_replicate reps vs
```

and a file *repiota.rfut* containing

```
let map 't [n] 'x (f: t -> x) (as: [n]t): [n]x = undefined

let replicated_iota [n] (reps : [n]i32) : []{i32(i) | 0 <= i && i < n} = undefined
```

type checking and Futhark code extraction can be done by executing the following command:

```
refinedfuthark segrep.rfut -o segrep.fut
```

The file *segrep.rfut* has an `import "repiota"` statement at the top and the tool will use the declarations in *repiota.rfut* when checking *segrep.rfut* . If the Futhark compiler and OpenCl has been installed, and the current directory contains a file *repiota.fut* containing

```
let segmented_scan [n] 't (op: t -> t -> t) (ne: t)
                          (flags: [n]bool) (as: [n]t): [n]t
                            =
  (unzip (scan (\(x_flag,x) (y_flag,y) ->
                 (x_flag || y_flag,
                  if y_flag then y else x `op` y))
           (false, ne)
           (zip flags as))).2

let replicated_iota [n] (reps:[n]i32) : []i32 =
```

```
10  let s1 = scan (+) 0 reps
11  let s2 = map (\(i:i32) -> if i==0 then 0 else unsafe s1[i
        -1]) (iota n)
12  let tmp = scatter (replicate (unsafe s1[n-1]) 0) s2 (iota
        n)
13  let flags = map (\(x:i32) -> 0 < x) tmp
14  in segmented_scan (+) 0 flags tmp
```

the generated Futhark file *segrep.fut* can be compiled with the command

```
1 futhark-opencl segrep.fut
```

As an alternative the file can be compiled to C code with the command

```
1 futhark-c segrep.fut
```

Finally the generated executable can be executed

```
1 ./segrep [1,2,3] [4,5,6]
```

The files mentioned in this section are available in the folder */examples/* of the project.

7.3 The parser

To implement the parser, we have studied the Futhark parser, which is implemented using the happy library of parser generators [7]. We have used the parsec library of parser combinators instead. Since we only work with a subset of the Futhark language, large parts of the Futhark syntax is not supported. We have made a few extensions to the syntax to support refinement type annotations. The extended type annotations syntax is $\{t \mid P\}$ and `i32`(n). The parser takes a Futhark program and returns an abstract syntax tree of the program. The module with the source code for the parser is available in the file
/src/RefinedFuthark/Parser.hs.

7.4 The transform function

The transformation function is implemented as three recursive functions `transformExp`, `transformType` and `transformParams` in the file */src/RefinedFuthark/Transform.hs*. The implementation directly corresponds to the functions described in Section 5.

7.5 The checker

The source code for the checker can be found in */src/RefinedFuthark/TypeCheck.hs*.

We export one function from the module `RefinedFuthark.TypeCheck` that has the following type `typeCheck :: Program -> Either Error [Constraint]`.

The implementation of the type checker follows closely the type cheking rules described in Section 6. Each judgement roughly corresponds to a function and each rule roughly corresponds to a pattern match case. We have not implemented all the rules and we prioritized the rules that would help us check our examples.

The type checker uses a monad internally to keep track of state and to report errors. The interface exported by the module is monad-free and returns either an error or a set of constraints. The constraints are also checked by a very simple checker before the function returns. The internal state of the type checker consists of the context, a generator of fresh variable names, and the set of constraints produced so far.

The context Γ, which is used by all judgements, is represented by a list of context items and a map that maps existential type and index variables to their solutions. A context item is represented by a data type and includes term variable bindings and type and index variable declarations.

The function `typeCheck` is simply a wrapper for passing the initial environment and unpacking the monadic result of type checking, which is computed by the function `checkProgram`. The `checkProgram` function calls `checkDec` on each top-level declaration and then gets the generated constraints from the internal state.

Finally the `checkDec` function calls `checkType`, which corresponds to the checking judgement from the type rules as described in Figure 12.

```
1  checkType :: Exp -> TypeExp -> TypeM ()
2  checkType e t = case e of
3    Lambda p ta e1 -> checkLambda p ta e1 t
4    Let binds e'-> checkLet binds e' t
5    Var _ -> checkSub e t
6    Apply _ _ -> checkSub e t
7    IntLit _ -> checkSub e t
8    Index _ _ -> checkSub e t
9    _ -> throwError $ "Cannot yet check syntactic form "
10                     ++ prettyExp e
```

If the expression to be checked is a synthesizing form, the function `checkSub` is called, which first synthesizes a type using the `synthType` function, which implements the synthesizing judgement from Figure 11. The function `checkSub` then continues by checking if the synthesized type is a subtype by calling the function `isSubtypeOf`, which implements the subtype judgement described in Figure 14.

Several other functions correspond to judgments, for example, let expressions are handled by the function `checkLet` and implements the let-rules described in Figure 18.

```
checkLet :: [(Ident, Maybe TypeExp, Exp)] ->
            Exp -> TypeExp -> TypeM ()
checkLet [] e2 t2 = checkType e2 t2
checkLet ((ident,ta,e1):binds) e2 t2 = do
  t' <- case ta of
    Nothing -> synthType e1
    Just t'' -> checkType e1 t'' >> return t''
  c <- getContext
  m <- getExVarMap
  checkLet1 (replaceContext c m t')
  where
    checkLet1 t = case t of
      AssertingType t' p -> do
        v <- newMarker
        consCond p
        checkLet1 t'
        removeAfter v
      SigType ident' _sort t' -> do
        consContext (UVar ident')
        checkLet1 t'
        removeAfter ident'
      _ -> do
        addVar ident t
        c <- getContext
        m <- getExVarMap
        checkLet binds e2 (replaceContext c m t2)
        removeAfter ident
```

The function handles a whole list of let-bindings at once. We could also have nested the let-expressions first so that we only had to handle one binding at the time but a list of bindings is closer to how the program looks in the source code. If the expression is annotated with at type, it is checked against a type; otherwise a type is synthesized for the expression.

The type is then unpacked if it is an existential or asserting type and finally it is bound to the variable before the other bindings are checked.

7.6 Type erasure

The implementation of the erasure function is based on the function described in Figure 7. The code for the function can be found in */src/RefinedFuthark/-Transform.hs*.

The erasure function takes a program written in the internal language and removes all the extended type information by recursivly going through the types.

7.7 The pretty printer

The pretty printer takes an AST of a program written in the external language and transforms it to a Futhark program where the `unsafe` keyword is added. The code for the pretty printer is in */src/RefinedFuthark/Pretty.hs*. We have only implemented enough of the pretty printer for our examples to work.

7.8 The constraint solver

We have implemented a very simple constraint solver that checks the constraints before reporting that type checking succeeded. It is limited to checking constant inequalities and inequalites that are part of the premises.

The constraint solver lives in the type checking module and consists of the function `solveConstraint`.

8 Evaluation

In this section, we will evaluate to which degree we have obtained the goals set forth in the introduction.

8.1 Testing

To test the correctness of our implementation and find bugs, we have developed a test suite. We have also manually applied the main executable to the programs with extension .rfut found in /examples/. The test suite

is implemented in Haskell, in the file */test/Spec.hs*, using the 'tasty' testing framework library.

To run the tests execute the command
`stack test`
in the root folder.

We tested both the parser and the type checker separately. We have used positive white-box testing of the parser and both positve and negative white-box testing of the type checker.

8.1.1 Parser tests

For each test case, the test framework parses a program and fails the test case if the program could not be parsed. We have also implemented some test cases that parse a program and then pattern match against an expected syntactic structure to check if the produced syntax tree is of the expected form. The test case fails if the syntax tree is of the wrong form because of pattern match failure. This technique is used to make the test cases simpler.

We have not used negative tests for the parser which would be an obvious improvement. Another improvement to testing the parser would be to use randomized testing with quickcheck with the predicate
`pred p = p == parse (pretty p)`.

8.1.2 Type checker tests

For each test case, the test framework parses, transforms the program, and then type checks the input program. The test case succeeds if the program is correctly typed.

We can think of the syntax and type rules as a kind of test specification in that we can use that to plan the coverage of our test cases. For example, we should have a test case for each syntactic construct to check that there is a rule for each construct. Then, we should also have a test case for each rule to check that the rule is implemented. Further tests should be designed to test that different rules correctly interact.

Since the test output space is so simple (the program is correctly typed or it is not) it is hard to be confident of the corectness of the implementation just by the kind of testing mentioned above.

Positive tests only check that correctly typed programs can be type checked. On the other hand, negative tests check that incorrectly type programs are rejected. Failure to reject bad programs can go completely unnoticed since the programs can be executed just as correctly typed ones. This probem is enhanced by the fact that type checking includes collecting and solving constraints and forgetting a constraint can make the type checker accept a bad program. Lastly our subtype checking of existential type variables involves adding subtype constraints to the context and checking them later, which again can lead to forgetting them and accepting bad programs. Because of the problems above, we have also employed negative tests.

Even though we have negative tests, it is hard to be confident that our implementation is correct in general. It is difficult to enumerate badness and due to the problems discussed above, we feel like a lot of things can go wrong.

Apart from having tests covering all of the positive cases outlined above, as well as more negative tests, we could improve our testing by testing with a bigger output space since our type checker generates constraints. We have not explored this option.

In any case much more extensive testing is needed before we can be confident that our implementation is correct and never accepts incorrect programs.

8.1.3 Known bugs

There are at least two bugs that we know of. The first is the combination of existential variable subtype checking together with the $\rightarrow I\hat{\alpha}$ rule. The rule splits an existential variable into two new existential variables, which it then can solve separately. The problem is that our existential variable subtype constraints do not currently track these newly created variables. These variables are tricky since they are created in the middle of the context as opposed to the tail. This pattern of creating variables in the middle of the context is used in several different rules and we are yet to find a way to handle them. For now we have disabled the $\rightarrow I\hat{\alpha}$ rule.

The other bug is name clashing as several different universal type variables with the same name can be in the context at the same time. We should generate a unique tag for them to distinguish between them just as we generate fresh names for existential variables. They should keep their names as they

are referenced in the program and could be used in error messages to the user.

8.2 Runtime improvement

If we require that array access is safe and accept that our type checker is correct, then the performance benefits are the difference between sequential and parrallel execution. We will not explore here how large the speedup is of parallel programs compared to their sequential equivalents. But if one regards unsafe array access as acceptable then there are no performance benefits but instead a greater assurance of correctness.

We can say that our type system introduces no slowdown as opposed to other potential type systems because the types can be erased at compile time. The main goal of the project was to enable safe array indexing on parallel architectures and not to speed up sequential programs.

8.3 Usefulness

We have shown that refinement types can express the conditions necessary for safe array indexing. For example the function

```
1 let myget [n] 't (x : { i32(m) | m < n && 0 <= m})
2                  (arr : [n]t) : t = arr[x]
```

encapsulates array indexing in a function. We can even check that array indexing is safe when the index is drawn from an array, which is not uncommon in Futhark.

```
1 let segmented_replicate [n] (reps : [n]i32)
2                             (vs : [n]i32): []i32 =
3   let idxs = replicated_iota reps
4   in map (\i -> vs[i]) idxs
```

This example also illustrates that we can use refinement types inside of function arguments to higher order functions like `map`. Also a common occurrence in Futhark. Furthermore the `map` function is polymorphic so we have shown the ability to support polymorphism together with refinement types and are even able to instantiate polymorphic type variables with existential types.

8.4 The user experience

A drawback of our system is that the users sometimes have to write the program using let expressions in order for existential and asserting types to be unpacked. However, Futhark programs are often already written in let heavy style so it is not dissimilar to how Futhark code is normally written.

As we have shown in Section 4, we have only introduced a few new type annotations and have used existing Futhark type annotations where possible, which we transform to refinement type annotations. The type annotations are fairly lightweight and do not complicate the reading of the program.

Another benefit of our approchs is that if library maintainers check their library code using refinement types, a user of these libraries can use the checked code without having to use refinement types themselves.

Without adding any aditional type annotations, other than those already in Futhark libraries, our system allows static type cheking of array sizes. The type annotations also serve as documentation.

8.5 The process

We had decided to use the work presented by Xi and Pfenning [23, 25] and base our solution on the rules presented there. After reading and designing some rules we started implementing the solution in Haskell to get a more hands on experience of the problems we might encounter.

We had started implementing universal dependent types but had only started on existential dependent types and found that quite difficult. We also discovered that implementing polymorphism would require essential changes to the system. Finally, the system required that the program was on A-normal form.

We discovered the work of Dunfield and Krishnaswami [10] that was similar in many regards and that handled the above mentioned problem as well as had other advantages. As mentioned, the rules described how to deal with existential variables and also polymorphism in a more clean way. We did, however, later discover that it did not solve the necessity of A-normal forms. We decided to change framework and instead based our rules on the new work.

The original work by Xi and Pfenning was some of the first that used refinement types and we felt it gave us a good understanding of refinement types.

Even though we spend time on implementing part of the solution in the first framework and did not end up using it, the understanding we gained from studying and implementing the work by Xi and Pfenning was valuable.

Implementing the type rules was not too difficult but extending the system proved more complicated. We lifted the restriction that type variables could only be instantiated to monotypes but this involved creating rules for existential variable subtype checking, which do not feel very satisfying or easy to work with. We think it is because they do not have a solid theoretical foundation.

The algorithmic type rules, which we have adapted, are guided by a set of declarative type rules and existential variable instantiation models ordinary unification. Dunfield and Krishnaswami mention that to extend this notion to polytypes an approach would be to use nominal unification instead. We lack knowledgde in proof theory and logic to truly understand the underpinings of their theory so we did not know how to develop rules based on nominal unification.

8.6 Conclusion

The purpose of this thesis was to discover if extending the Futhark type system with refinement types would allow the programmer to refine a program using types in such a way that array bounds checking can be avoided at runtime.

We said we would test this hypothesis by (1) designing a refinement type system for Futhark, (2) implementing it in a small compiler that erases the types and produces safe Futhark code without checks, and (3) evaluating its use based on a number of small and medium examples. Moreover, we said we would evaluate the usability of the refinement type system, for instance by comparing example refinement-type-annotated Futhark programs with their unannotated versions to inspect the added complexity.

We have shown that, at least for a limited language, it is possible to verify the safety of array indexing at compile time. Refinement types can be used together with higher-order functions and polymorphism.

Users sometimes have to write more type annotations since unification based type inference is no longer used, this is discussed in Section 9.5. Existential types complicate things, and sometimes expressions have to let-bound to make programs type correct Section 8.4.

It is difficult to determine if our findings can be generalized to more realistic programs in a more realistic setting since we have only covered a small subset of Futhark and have therefore not been able to apply our type checker to more example programs. More exploration would be needed to really be sure how practical refinement types would be in larger programs, but we think they have potential.

What was maybe most encouraging was how reinterpreting array size annotations as array type indices led us to develop the syntax $\{t(x) \mid P(x)\}$, a sort of pattern matching on type indices, which we think is very neat and can be used both for arrays and integers. The usual syntax is $\{x : t \mid P(x)\}$ which is neat in some cases but does not work very well for arrays. For example the type of map in Liquid Haskell is `(a -> b) -> xs : List a -> { ys : List b | len xs == len ys }` [1], on the other hand `{i32(n) | 0 < n}` is not much worse than `{n : i32 | 0 < n}` We imagine the syntax could also be generalized for pair types `{(i32(n),i32(m)) | n < m}` and index pairs `{sometype(n,m) | n < m}`.

9 Discussion

In this section we discuss our work, what we have learned, and what we would have done differently.

9.1 The type rules

Even though our goals are more alike DML, that is to refine the type indices of some base types which can be erased, we have chosen to base our rules mostly on the work by Dunfield and Krishnaswami [10].

The type rules in DML have some drawbacks, the constraints generated do not only contain inequalities in the integer domain, but also the equality constraints produced by type variable instantiation. This means that instantiation failure cannot be detected and reported until the solver is applied to

[1] `https://stackoverflow.com/questions/56326273`

constraints, or atleast not during the application of the rules themselves. Of course the implementation can be modified to catch some failures earlier but the rules do not reflect that.

There is also a separate step of eliminating existential variables from the constraints which further seperates constraint solving from rule application. These steps are an integrated part of the rules by Dunfield and Krishnaswami [10].

9.2 Coverage of Futhark

We have only covered enough of Futhark to support array indexing and our examples. Notable omissions are: Multidimensional array indexing, modules, loops, algebraic datatypes and uniqueness types.

We do however support polymorphism, higher-order functions and let expressions.

We cannot parse most real Futhark programs because we cannot parse of check most of Futhark, but we think that we support most of the diffucult and most illustrative constructs.

But the intention from the start was exactly to support a small core functionality only for the purposes of exploring refinement types for use of checking array indexing.

We think that this is a good starting point to explore refinement types in larger programs, especially because of the composable nature of types. The composibility means that we can check the parts of the program that we want. We can also let types express properties that we cannot check and check the rest of the program under the assumption that the property holds. This is also useful if we want to define refinement types for built-in functions and/or define different refinement types for the same function in different parts of the program.

We can check the refinement types of a small library of Futhark functions defined in *examples/lib.rfut*. We can express refinement types of yet more functions defined in *examples/prelude.rfut*. These types can be used in other rfut files. This is nessesary in order to express/override the types of the SOACs that are built into Futhark.

9.3 Expressiveness of Refinement Types

Our refinement type system is an example of a form restriced dependent types.

In our approch it is possible to erase the types, but this is not always possible when using fully dependent types. This means that refinement types are a zero cost abstraction, that is, they do not slow down the execution of the program which is important in Futhark as speed is an important feature.

That the types have to be erasable creates limitations. In general this means that it is not possible to type programs that are ill-typed in Futhark, the extension is conservative. Dependent types are an example of an extension that is not conservative but so is for example polymorphism. When a language incorporates polymorphism the implementors have to chose one of several strategies of implementation, all of which have implications for code generation or runtime representation.

There are properties that we cannot express, for example it is not possible to express or prove that the elements of an array are sorted. In general, we can only express position-independent properties such as all numbers are positive and so on. It is not clear how to even express relations between elements in an array in our case, since the maintaining of such properties are traditionally tied to the value construtors of the type, which is not how arrays are created.

Because of subtyping we can gracefully handle properties that are implied by stronger properties for example:

```
1 let x : {i32(n) | n < 5} = 4
2 let y : {i32(n) | n < 6} = x
```

Which is close to how humans would reason about programs. The type annotations can also be thought of as fairly uninstrusive documentation of the program.

9.4 Existential types

Existential types are really difficult to deal with but they are very powefull. They make it possible to use the functions with dependent types on regular types.

An original feature of our type rules is the ability to substitute existential types for type variables. The fact that we can instantiate polymophic type variables with existential types allows us to use the combinators with their standard types together with refinement types which is powerfull since we do not have to make new combinator types or understand them, the programmer can simply use the combinators they are used to. To represent an array of positive integers we simply use a regular array type that contains existentially quantified integers that are positive.

```
1 let as : [5]{i32(n) | 0 < n} = iota 5
```

Even though the machinery is complicated the end result is very simple to understand and hopefully therefore easy to use.

9.5 Type inference

Since Futhark has type unfication and our system does not make sense to ask how much difference there is in terms of type annotations needed. It is good practice to write the full type of at least top-level functions so we can focus on local declarations.

The syntactic forms are all either checked or synthesizing forms, could we make them all synthesizing? If so, it would mean that we have the same functionallity as unification.

We can probably not achieve full type inference in the face of existential types but maybe we can achieve it for the subset that is Futhark. So the programmer only pays with extra type annotations when they use the features.

A pair type can be synthesized if both of the types of the types of the componenets can be synthesized.

An interesting case is the lambda expression but a synthesizing rule for lambda expressions in bidirectional typing has been shown before [10].

$$\frac{\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \vdash e \Leftarrow \hat{\beta} \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Delta' \qquad \vec{\hat{\varepsilon}} = \text{unsolved}(\Delta')}{\Gamma \vdash \lambda x.e \Rightarrow \forall \vec{\alpha}.[\vec{\alpha}/\vec{\hat{\varepsilon}}][\Delta'](\hat{\alpha} \to \hat{\beta}) \dashv \Delta}$$

So this is actually possible. Let expressions could be synthesizing if not for the fact that they have to unpack existential and asserting types, so this meets our expection that it is the existential types that stand in the way of type inference.

9.6 Constraints and the constraint solver

We have implemented a simple constraint solver discussed in Section 7. Ideally we would like to use an SMT solver like Z3 [3, 6] since they are fast and can solve many kinds of constraints.

This is also the aproach chosen by Liquid Haskell [20, 22].

Another option would be to use constraint solver we had developed ourselves, this is the option chosen by DML. This has the advantages that the user does not have to install a solver, however the users that have to install the solver are limited to library maintainers that use refinement types or users that want to check refinement types themselves. Users can still use pre-checked libraries without installing a solver.

However, even with our very simple solver we were suprised with the programs that could be checked, this is due to the fact that the type system substitutes index variables so many of the inequalities are either constant or identical to inequalities in the premises.

This variable solving by the rules reduces the need for equality constraints in our constraint set but there is still an equlity case that our type rules cannot handle and that is the case where we have $f(i) \stackrel{\circ}{=} g(j)$. This could for example be the linear constraint we $x + y \stackrel{\circ}{=} a + b$. Right now this form does not have a rule but if the solver supported equality constraints the above form could be handled by adding the equality constraint to the constraint set.

10 Related Work

Refinement types first appered in work by Freeman and Pfenning [14]. Later Phenning and Xi developed restricted dependent types [23–26] where they added indices to types for type refinements. This work resulted in DML (Dependent ML). Dunfield distinguised between datasort refinements, introduced as refinment types by Freeman and Phenning, and index refinements (the technique used to implement retricted dependent types by Xi and Pfenning) and combined the two in his dissertation [8].

DML extends a subset og the programming language ML with a restricted form of dependent types where the index domain is linear arithmetic. In Xi's dissertation he first extends a monomorphic subset of ML with Π-types and

afterwards Σ-types before adding polymorphism and effects. He introduces three sets of rules for internal type checking, elaboration and constraint generation. The type rules are bidirectional. To handle existential types all programs are transformed to A-normal form which in turn makes some programs not typable.

Our type rules borrow most heavily from the work of Dunfield and Krishnaswami [10], who formulated indexed types in terms of concepts from logic and proof theory. Their aim was to reduce GADTs to these concepts.

Our goals are rather different and match the goals of DML much more closely.

Other work on refinement types include Liquid Types [20, 22], which extend Haskell with refinement types. It differs from previously mentioned work by refining base types with (some restricted set of) program terms instead of indices. Refinement type annotations are expressed as Haskell comments. Because Haskell is lazy the variables can be bound to abitraty terms instead of values, using their refinments can result in inconsitency. Therefore, terms are checked to see if they diverge, converge to a value in head-normal form or converge to a finite value. Only refinments on converging terms can be used to build constraints. The constraints are solved using the Z3 SMT solver.

Other work where array types annotated with sizes are used when targeting Futhark include Tail2Futhark [16, 19].

Dependent types have been used to address array bounds checking for a functional array calculus [21]. Integer vectors of staticly unknown length are used to index array types. Constraints on the vectors are resolved to integer scalars.

11 Conclusion and Future Work

In this section we will present some ideas for future work and conclude.

11.1 Future work

There is a lot of work still to be done before refinement types in the context if array bounds checking in Futhark is sufficiently explored. More of Futhark has to be covered by the external language in order to type check larger and

more realistic Futhark programs. With support for more of Futhark we could maybe type check `replicated_iota` or `qsort` from `https://github.com/diku-dk/sorts` which uses existential dependent pairs, stored in an array, to represent segments. This would also serve to test our implementation better. An obvious place to extend the external language would be to add conditionals.

Proofs of soundness, completeness, progress and preservation would help a lot to verify that the type checking rules are properly designed.

There is a lot that can be done to improve the usability of the implementation. Type abbreviations could be added to shorten the longer types. For example

```
type fin(m) = {i32(n) | 0 <= n && 0 < m}
```

makes the type of `replicated_iota` shorter

```
let replicated_iota [n] (reps: [n]i32): []fin(n) = undefined
```

Existential types can be cumbersome to use. A rule that unpacks existential without having to use let-bindings would diminish how much programs have to be changed to use refinement types.

One feature we hoped bidirectional type checking would enable are descriptive error messages and for that we would like to add location information to the syntax tree so we could display the line the error occured on.

The index domain could be extended, for example $i_1 + i_2$ would allow us to type `concat`

```
let concat 't [n] [m] (xs: [n]t) (ys: [m]t): [n+m]t = xs ++ ys
```

We could also include index pairs $\langle i_1, i_2 \rangle$.

Finally we would like to use a real solver like Z3 to solve constraints or implement a solver for linear constraints like Xi and Pfenning.

11.2 Closing remarks

In this thesis we have shown that it is possible to add refinement types to a subset of Futhark and leverage refinement types to check at compile-time the safety of array accesses. We have presented bidirectional type rules for refinement types in a subset of Futhark and implemented a type checker based on these rules. The type checker is part of a compiler that, after type

checking, erases the types and outputs Futhark code that is known to be safe and does not contain any runtime bounds checks.

To simplify the type rules while keeping the source language close to Futhark, we have presented two languages. The first language is an external source language, which is a small subset of Futhark with a few extra kinds of type annotations. The type annotations guide the type checker but are also useful as documentation and are not so big as to be instrusive. The second language is an internal language, which is designed to simplify the type rules. To complete the type checking process we have specified a transformation function, which transforms programs in the external language to programs in the internal language. We have evaluated the type checker by running it on some example programs.

References

[1] Nvidia developer. `https://developer.nvidia.com/nvidia-developer-zone`. [Online; accessed 27-August-2019].

[2] Ana Bove and Peter Dybjer. Language engineering and rigorous software development. chapter Dependent Types at Work, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009.

[3] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Coq development teem. The coq proof assistant. [Online, Accessed 27-August 2019].

[5] Idris development teem. Idris - a language with dependent types. [Online, Accessed 27-August 2019].

[6] The Z3 development teem. The z3 theorem prover. [Online, Accessed 27-August 2019].

[7] DIKU. The futhark programming language. `https://futhark-lang.org/docs.html`. Accessed: January 2019.

[8] Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, August 2007. CMU-CS-07-129.

[9] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. *SIGPLAN Not.*, 48(9):429–442, September 2013.

[10] Joshua Dunfield and Neelakantan R. Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *Proc. ACM Program. Lang.*, 3(POPL):9:1–9:28, January 2019.

[11] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. Static interpretation of higher-order modules in futhark: Functional gpu programming in the large. *Proc. ACM Program. Lang.*, 2(ICFP):97:1–97:30, July 2018.

[12] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark*. DIKU, November 2018.

[13] Martin Elsman, Troels Henriksen, and Niels Gustav Westphal Serup. Data-parallel flattening by expansion. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, pages 14–24, New York, NY, USA, 2019. ACM.

[14] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.

[15] Futhark Library Maintainers. lib/github.com/diku-dk/segmented/segmented. `https://futhark-lang.org/pkgs/github.com/diku-dk/segmented/0.2.1/doc/lib/github.com/diku-dk/segmented/segmented.html`, 2019. [Online; accessed 27-August-2019].

[16] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. Apl on gpus: A tail from the past, scribbled in futhark. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, FHPC 2016, pages 38–43, New York, NY, USA, 2016. ACM.

[17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 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, pages 556–571, New York, NY, USA, 2017. ACM.

[18] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. High-performance defunctionalisation in futhark. In Michał Pałka and Magnus Myreen, editors, *Trends in Functional Programming*, pages 136–156, Cham, 2019. Springer International Publishing.

[19] Anna Sofie Kiehn and Henriks Urms. Compiling tail to futhark–an adventure in compiling functional data-parallel constructs, June 2015.

[20] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

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

[22] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, 2014. ACM.

[23] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Pittsburgh, PA, USA, 1998. AAI9918624.

[24] Hongwei Xi. Dependent ML an approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

[25] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 249–257, New York, NY, USA, 1998. ACM.

[26] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.