This is a preprint, the final article can be found at https://doi.org/10.1016/j.scico.2023.102989

Gaiwan: a Size-Polymorphic Typesystem for GPU Programs

Robbert Gurdeep Singh[a,b,*], Christophe Scholliers[a]

[a] *UGent, Krijgslaan 281-S9, 9000 Gent, Belgium*

[b] *IMEC, Remisebosweg 1, 3001 Leuven, Belgium*

Abstract

General-purpose computing on graphics processing units (GPGPU) is increasingly used for number crunching tasks such as analyzing time series data. GPUs are a good fit for these tasks as they can execute many computations in parallel. To leverage this parallelism, the programmer is forced to carefully divide their input data into data blocks that are then distributed over the many GPU cores. The optimal block sizes are unrelated to the programmers goals, instead, they are based on characteristics of the used GPU and the input data. GPGPU programmers must additionally be wary of introducing race conditions in their programs.

We believe that GPGPU programmers should be able to express GPU transformations without worrying about splitting data or race conditions. For this, we created Gaiwan, a GPGPU programming language with a size-polymorphic type system that only features data race free operations. Programmers can declare the effects of program steps on the sizes of buffers by using affine functions (eg. $\texttt{int}[2n] \twoheadrightarrow \texttt{int}[n+1]$). From a step sequence, Gaiwan derives a set of constraints on the size and shape of valid inputs. Gaiwan guarantees that the program will run for any input satisfying these constraints. This means that one program may analyze both a hundred data points and millions of data points, as long as the input satisfies the constraints.

We prove that our system is sound and show it works with two usage examples. Our benchmarks show that our initial OpenCL-based implementation of Gaiwan scales to handling large programs.

Keywords: GPU, Polymorphism, Type System, OpenCL, Unification

1. Introduction

Parallel hardware such as Graphics Processing Units (GPU) promise great performance gains for those experts capable of programming these devices. Unfortunately, not everyone that wants to reap the benefits of GPUs for general purposes (GPGPU) is such an expert. GPUs have thousands of processing units that all work in parallel. This gives rise to two difficulties. First, the programmer needs to take care not to introduce race conditions in their program. Second, they must also explicitly divide their data into so-called blocks, which are assigned to individual cores [1, 2]. The sizes of these blocks are unrelated to the problem the user is trying to solve. Instead, the sizes are related to the available space in the various memories of the GPU. Improper block sizes can have a detrimental impact on the performance of a GPU program [3]. Selecting the optimal block sizes is no simple task and requires much prior experience with GPU programming. To make matters even worse, the optimal value is device and input dependent.

*Corresponding author

The developer new to GPGPU does not have this experience but may still want to benefit from the performance gains GPUs could offer. We believe that they should be able to express the transformations a GPU needs to carry out on their data without having to worry about data races or block sizes.

Many data processing applications involve time series data. These are arrays of measurement values that are carried out periodically over longer periods. Because of their nature, the sizes of these data sets are not fixed. However, a programmer might want to do a GPU analysis over an arbitrary period without having to rewrite their program. We believe that a program that processes the data for 5 days should also be able to process the data for 5 years. This enables users to test their programs on smaller data sets and have the confidence that they will still work for larger data sets.

Many higher level programming languages such as Accelerate[4, 5], StreamIt[6, 7] and others [8, 9, 10] exist with the aim of lowering the barrier to GPGPU programming. Unfortunately, these languages do not give full visibility into the effects each transformation has on the size of the buffers, especially when these sizes are not known beforehand. This makes it difficult for a novice user to transform their data processing idea into a GPU program that will work on data sets of different sizes.

We see a need for a programming language targeted at non-expert GPGPU users that want to analyze time series data without having to explicitly chop up their data. Such a language should allow users to specify how large a buffer is as a formula, rather than as fixed number. The language should also be capable of efficiently deriving the access patterns of any program. These can then be used to algorithmically select the optimal block sizes based on the size of the input and the available hardware. To make a program in such a programming language safe, we must be able to check its type regardless of the size of the input. This is no straightforward task because a program may, for example, take the average of every five points before continuing its computations. Such a program cannot process a data set of length 234123 as this is not a multiple of five. It is clear that a type checker for a program that works for a variable size of input will need to put some constraints on the length of its inputs, for example that the length should be a multiple of five. A GPGPU language for time-series data should present these constraints to the user. Additionally, the language should ensure that data races are not possible.

In this paper we present Gaiwan, a data race free programming language featuring a size-polymorphic type system. Gaiwan allows users to specify that a function called `flatten`, for example, transforms an array of n pairs into an array of $2n$ elements. This function could be assigned the type $(A \times A)[n] \twoheadrightarrow A[2n]$. Our `flatten` function is parametric in the size of the input n. An input of 42 pairs will yield an output of 84 items. Additionally, the function is parametric in the shape of the inputs elements. Before the arrow, we specified that `flatten` expects a buffer of pairs of two elements of the same shape $(A \times A)$. After the arrow we stated that it returns a buffer of elements of the shape A, where A refers to the A used before the arrow. If we supply pairs of integers (`int` $\times$ `int`) we will get back two `int`s per pair. To support this kind of polymorphism, we created a novel unification system that codifies this behavior.

Our programming language also clearly separates the selection of data and the computation of new values. This aids performance optimization[10]. Users select what data is needed and then specify how to compute a single element of the output in a data race free manner. Our type checker returns the output type of the program as well as a set of constraints on the input buffers. To the best of our knowledge, there are no type systems for GPU programs that allow us to express such transformations and to retrieve a set of constraints on the length of the possible inputs as simple affine functions.

In this paper we first discuss the various constructs of Gaiwan informally in section 2. Then, we formalize our novel type system (section 3) and its unification process (section 4). With that, we can discuss the evaluation rules (section 5) and prove they are sound (section 6). Next we briefly look at implementation considerations in section 7. An evaluation of our work is given in section 8. We look at two usage examples of Gaiwan: one where we analyze GPS traces and one where we compute the dot product. Additionally, we show that our concepts are practically implementable and executable on real hardware with a prototype Gaiwan executor for OpenCL. Finally, we compare our concepts to the state of the art, discuss future work and conclude the paper in sections 9 to 11.

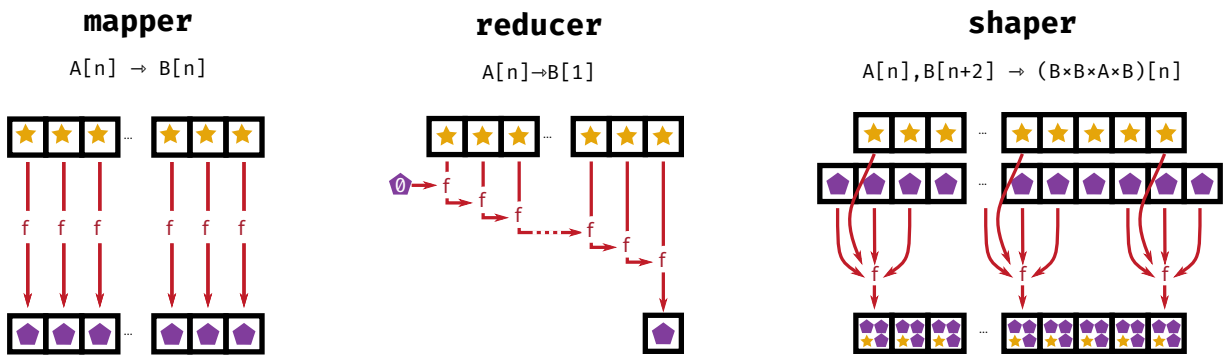


Figure 1: Transformations in Gaiwan

The source code of the prototype implementation of Gaiwan is available at https://github.com/TOPLLab/gaiwan/tree/elsref-2023.

2. Gaiwan Programming Constructs

In this section we lay out the basic constituents of a Gaiwan program. Our language is designed to be as small as possible while still allowing the exploration of our size-polymorphic type system.

Gaiwan programs deal with data buffers (section 2.1) which can be manipulated by transformations (section 2.2). The order in which these transformations are applied is dictated by our coordination plan (section 2.3). This plan is a list of actions where the output of previous action becomes the input to the next.

2.1. Buffers

As in any GPU programming language, data buffers play a significant role in Gaiwan. A data buffer containing the integers 1, 2 and 3 is written explicitly as `(buf`$_{\texttt{int[3]}}$ `1 2 3)`. The type of this buffer is `int[3]`, where `int` is the shape of the elements in the array and 3 is the length. To avoid confusion we use the word "shape" to denote the type of the elements in the buffer. The word "type" will be reserved to describe the type of buffers and transformations on them. In general, the type buffer of length n with elements of shape S is written as $S[n]$.

Interestingly, in Gaiwan, both the n and the S in the type $S[n]$ may contain variables. We allow any number of variables in the shape part of a buffer type. The size, on the other hand, must be an affine function in at most one variable ($a \cdot n + b$, with a and b fixed integers). For example, the type $(A \times A)[2 \cdot n + 1]$ represents a buffer whose elements have a tuple shape where both items are of shape A, that is, there exists a shape A such that the buffers elements are of shape $(A \times A)$. The length of the typed array must be odd, that is there exists an n such that the length of the buffer is $2n + 1$.

2.2. Transformations

Gaiwan has three main data race free transformations that can be executed on buffers: **Mappers** apply the same function to every element in a buffer; **Reducers** combine all values of a buffer into one; and finally **Shapers** reshape one or more buffers to create a new one without inspecting the values of these buffers. All our transformations have a type of the form below.

$$\underbrace{(S_1[a_1 n + b_1], \ldots, S_m[a_m n + b_m])}_{m \text{ input buffers}} \twoheadrightarrow \underbrace{(S_{\text{out}}[a_{\text{out}} n + b_{\text{out}}])}_{1 \text{ output buffer}}$$

```
1 mapper(idx: int, number: float) : float {
2     number+number
3 }
```

Listing 1: A doubling mapper

Each input buffer type $S_i[a_i n + b_i]$ and the output buffer type $S_{\text{out}}[a_{\text{out}} n + b_{\text{out}}]$ have affine lengths in the same variable (in this case n). All free variables in $S_{\text{out}}[a_{\text{out}} n + b_{\text{out}}]$ must occur in at least one of the input buffer types. The type serves two purposes. First, it ensures that the body of the transformation computes a value of the expected shape. Second, it describes a relation between the lengths of the input buffers, and the output buffer.

Having discussed the type of transformations briefly, we now look at the transformations Gaiwan offers.

2.2.1. Mappers: Transforming Values

The mapper is our first kind of transformation. Figure 1 shows a schematic of it on the left. It applies a function to every element in a buffer. A mappers body may only access the value of the element it is modifying to compute its new value. By enforcing this, there are no data dependencies between the computations of different elements. As a consequence, we may safely apply a mappers body to the elements of a buffer in any order and even in parallel. That is, mappers are a data race free construct.

Listing 1 shows an example mapper that doubles the values in the buffer it is given. We provide the formal argument list and the return type on line 1. Every mapper has two arguments: (a) the index of element in the buffer we are currently operating on (here `idx` of shape `int`), and (b) the value of that element (here `number` of shape `float`). The return type, given after the final colon (`:`) on line 1 describes the shape that returned values are expected to have. In this case the body returns a `float` by adding the `float` value `number` to itself.

A mapper only has one input buffer and cannot change lengths. The type of a mapper is therefore always of the form $(S_1[n]) \twoheadrightarrow (S_{\text{out}}[n])$. For a mapper, S_1 will always be the type of the argument, in this case `float`, the shape of `number`. The output buffer has elements of the same shape (S_{out}) as the body returns, here `float`. Listing 1 thus shows a transformation of type $\texttt{float}[n] \twoheadrightarrow \texttt{float}[n]$.

2.2.2. Reduce: Combining Buffer Elements

Apart from transforming single elements it is often also useful to apply a function to all values of a buffer. This is where the reducer, a parallel folder, comes in. A schematic of a reducer can be found in the middle of figure 1. As opposed to mappers, reducers can access multiple elements. Only one element is accessed directly (`num`), the other accessed elements are summarized in an accumulator.

Listing 2 shows the definition of a reducer that sums all the elements in a buffer. This reducer has two arguments: (a) an accumulator `acc` of shape `int` together with its initial value `0`, and (b) the value that is being folded into the accumulator (`num` of shape `float`). The body returns the value the accumulator holds in the next iteration, it must therefore be of the same shape as the accumulator (`int`). In the end, all values will have been folded into the accumulator.

If the operation defined by the body of the reducer is associative, we may group elements together to improve the runtime performance. This is done by and applying the reducer to each pair of elements and subsequently combining the results of the reducers with the reducer. Figure 2 shows an example of this method of execution. The number of iterations is logarithmic in the number of elements, as opposed to linear in the sequential counterpart. For large buffers the parallelism a GPU offers can improve the time complexity of an operation [11]. Gaiwan automatically tries to derive whether the operation is associative, if not it falls back to the (slower) serial computation of the result.

A reducer takes a buffer of arbitrary length and reduces it to a buffer of size one. The type must therefore always be of the form $(S_1[n]) \twoheadrightarrow (S_{\text{out}}[1])$. The input shape S_1 is the shape of the data parameter (in this

```
reducer(acc = 0 : int, num: float) : int {
    acc + num
}
```

Listing 2: reducer computing the sum

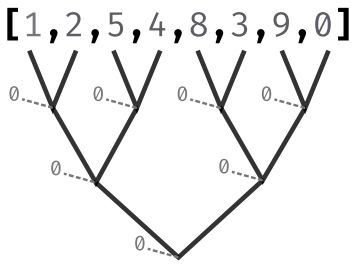


Figure 2: Parallel scheduling of a reducer on a small buffer. The initial accumulator is combined with two elements to compute the values for the next round.

case `float`, the type of `num`). The return shape is the shape of the accumulator (in this case `int`). Our example reducer thus has the type (`float`$[n]$) $\multimap$ (`int`$[1]$).

2.2.3. Shapers: Data Reshaping Transformations

Our final transformation, the shaper, allows users to restructure the data within a buffer. Figure 1 shows a schematic of a shaper on the right. Notice how the pattern of arrows is the same for every output element. A shaper defines these arrows and how the selected elements are combined. Listing 3 shows a shaper that takes a buffer of even length $2n$ and returns a buffer of half the length that contains tuples. The type annotation allows us to change the number of elements by specifying an affine function.

A significant restriction on shaper types is that the input shapes, the S_i's, may not use concrete types (like `int`). Instead, the shapes of the element in the input buffers must be composed of a combination of tuples and shape variables. This limitation ensures that shapers cannot inspect the precise value of a buffer element, they may only reshuffle them. If `buffer` is typed $A[n]$, `buffer[i]` returns a value of type A. Such a value cannot be used in integer computations for example, because these computations only work on `int`s.

When a shaper is applied to a buffer, the free variables in the types are filled in. For example if we supply an `int`$[4]$ buffer (`buf`$_{\texttt{int}[4]}$ 1 2 3 4), the type of an $A[2n] \multimap (A \times A)[n]$ shaper will become `int`$[4] \multimap$ (`int` $\times$ `int`)$[2]$. The shape variable A is replaced by `int` and the n has been derived to be 2 (from $4 = 2n$).

The shape of the elements of the input buffer in Listing 3 is the free shape variable A, the length is an affine function in one variable, in this case the function $2n$. The return type of the shaper is specified as $(A \times A)[n]$, this indicates that the output has length n and is composed of tuples of two values of shape A. The output is half the size of the input, n output values for $2n$ input values. We write $(A[2n]) \multimap (A \times A)[n]$ as the type of the shaper.

The first line of Listing 3 declares that this shaper has two arguments: `outIdx` of type `int` and a `buffer` of type $A[2n]$. The first argument (`outIdx`) specifies the index of the value in the **output** buffer we are currently computing. All other arguments, in this case only "`buffer`" of type $A[2n]$, specify the input buffer(s). In the body of a shaper, we have indexed access to the values of the input buffer by using `[]`. With `data[`i`]` we get the i-th value of `buffer`, which is of shape A. Note that the shaper is the only transformation where indexed access can be used. The other transformations do not have buffers or arrays in their parameter list.

We may also use multiple input buffers as shown in listing 5. In this code snippet we combine the values of two buffers, one of type $A[n]$ and another of type $B[2n]$ to get a result of type $(A \times B \times B)[n]$. Sup-

```
shaper(outIdx: int, buffer: A[2*n]): (A × A)[n] {
  tuple(buffer[2*outIdx], buffer[2*outIdx+1])
}
```

Listing 3: shaper of type $A[2n] \multimap (A \times A)[n]$ making pairs of elements of the first buffer

```
shaper(outIdx: int, bufferA: (A × A)[n]): A[2n] {
  if(outIdx%2 == 0){ bufferA[outIdx/2]._1 } else { bufferA[outIdx/2]._2 }
}
```

Listing 4: shaper of type $(A \times A)[n] \multimap A[2n]$ destructuring a list of pairs into a simple list

```
shaper(outIdx: int, bufferA: A[n], bufferB: B[2n]): (A × B × B)[n] {
  tuple(bufferA[outIdx], bufferB[2*outIdx], bufferB[2*outIdx+1])
}
```

Listing 5: shaper of type $(A[n], B[2n]) \multimap (A \times B \times B)[n]$ making triples of two input buffers

```
1  abstraction vectorLengths(baseX:int,baseY:int):(float[n],float[n])->float[n]{
2      shaper(i:int, x:A[n], y:A[n]){
3          tuple(x[i],y[i])
4      } ;
5      mapper(i: int, d: (float × float)): float{
6          sqrt((d._1-baseX)^2 + (d._2-baseY)^2)
7      }
8  }
9
10 (retrive x newy) ; (call vectorlengths 0 0)
```

Listing 6: Lines 1-9: Abstraction that computes distance from a basepoint to a set of points whos X and Y coordinates are stored in two separate buffers. Line 10: coordination plan that calls the abstractioLine 10: coordination plan that calls the abstraction.

plying the `float`[2] buffer $[1.5, 2.5]$ and the `int`[4] buffer $[5, 6, 7, 8]$ yields the (`float` × `int` × `int`)[2] buffer $[(1.5, 5, 7), (2.5, 6, 8)]$. Shapers are not limited to combining elements: listing 4 shows how a list of pairs can be destructured into a simple list.

As shapers cannot inspect the values in a buffer, the length of the input suffices to determine the permutation a shaper applies. In the simple case of execution on a simple buffer `buf`, a shaper that describes the function f will result in a buffer `res`. Now, `res`$[i]$ = `buf`$[f(i)]$, so we do not really need to compute the concrete buffer `res`. Instead, the next transformation that uses `res` can just use `buf`$[f(i)]$ instead of `res`$[i]$ when accessing data. Shapers cannot introduce data races as they do not write values.

2.2.4. Abstraction

Mappers, reducers and shapers can be combined, parameterized and given a name by placing them in an abstraction. The top part of listing 6 shows an example abstraction of type (`int`, `int`) → (`float`$[n]$, `float`$[n]$) ⊸ `float`. This abstraction combines the functionality of a shaper (lines 2-4) and a mapper (lines 5-7) to transform a list of x coordinates, and a list of y coordinates, into a list of distances to a given point. The arguments of the abstraction (`baseX` and `baseY`) define the location of the given point. Finally, the abstraction is also assigned a name (`vectorLengths`), such that it can be called.

If an abstraction is executed, all occurrences of its scalar variables are substituted for the values with which the abstraction was called. What remains is a list of transformations. These transformations are simply executed one after the other. The output of the previous transformation becomes the input for the next. The output of the final transformation is the output of the call to the abstraction.

An abstraction may be composed of multiple transformations. The type of the full abstraction chains the types of these transformations. If an $A[n] \multimap A[2n]$ and a `float`$[m] \multimap$ `int`$[m]$ transformation are chained, the result is `float`$[n] \multimap$ `int`$[2n]$. This result is acquired by first unifying the shapes, the right A of the first type must unify with the left part of the second type. Thus we need $A =$ `float`. Now we can reformulate our

chaining goal to $\texttt{float}[n] \twoheadrightarrow \texttt{float}[2n]$ and $\texttt{float}[m] \twoheadrightarrow \texttt{float}[m]$. The next step is merging $2n$ and m, this can simply be done by setting m to be $2n$. We get $\texttt{float}[n] \twoheadrightarrow \texttt{float}[2n]$ and $\texttt{float}[2n] \twoheadrightarrow \texttt{float}[2n]$. Now, the middle types are the same, and we may conclude that the type of the abstraction is $\texttt{float}[n] \twoheadrightarrow \texttt{int}[2n]$.

Apart from the contained transformations, abstractions may also have arguments. In listing 6, there are two `int` arguments: `baseX` and `baseY`. We will use a different arrow ($\rightarrow$ instead of $\twoheadrightarrow$) to differentiate between scalar arguments to abstractions and inputs of transformation types. If we call the abstraction in listing 6 with two `int`s, we get $(\texttt{float}[n], \texttt{float}[n]) \twoheadrightarrow (\texttt{float}[n])$. Therefore, the full type of the abstraction is $(\texttt{int}, \texttt{int}) \rightarrow (\texttt{float}[n], \texttt{float}[n]) \twoheadrightarrow (\texttt{float}[n])$.

Abstractions cannot be nested and may only have scalar arguments. There are no "higher order" abstractions. All abstractions therefore have a type of the following form.

$$\underbrace{(\ S_{\text{in},1}, \ldots, S_{\text{in},k}\ )}_{k \geq 0 \text{ scalar arguments}} \mapsto \underbrace{(S_1[a_1 n + b_1], \ldots, S_m[a_m n + b_m])}_{m \text{ input buffers}} \twoheadrightarrow \underbrace{(S_{\text{out}}[a_{\text{out}} n + b_{\text{out}}])}_{1 \text{ output buffer}}$$

2.3. Coordination Plan

Gaiwan uses a coordination plan to coordinate in what order and on what data transformations are executed. Data can be thought of going through the program from left to right while being changed by the transformations. The output of the previous transformation step becomes the input to the next. Different steps in the coordination plan are separated with a pipe character (`§`).

We have two constructs that act as sources of buffers: `retrive` (section 2.3.1) and literal buffers (section 2.3.2). Once we have our data, we execute transformations on it with `call`. We store our intermediate results with `letB`.

2.3.1. Retrive: Source of Data

The "standard input" to a Gaiwan program is a memory that contains fixed buffers of numeric data. All buffers have a name and data. As a running example throughout this section, we will use the following, small, memory:

$$\{x \mapsto [1.0, 2.0, 3.0, 4.0, 5.0], y \mapsto [4.0, 2.0, 1.0, 3.0, 7.0], z \mapsto [7, 9]\}$$

We may construct the following memory type for it:

$$\{x \mapsto \texttt{float}[5], y \mapsto \texttt{float}[5], z \mapsto \texttt{int}[2]\}$$

To retrieve data from our memory, we use `(retrive` n_1 $\ldots$ n_m`)`. This action accesses the buffers with names $n_1, \ldots, n_m$. With our example memory, `(retrive` y`)` yields the tuple of one buffer ([4.0, 2.0, 1.0, 3.0, 7.0]). We may also `retrive` multiple buffers. `(retrive` z y z`)` yields a triple of buffers ([7,9],[4.0, 2.0, 1.0, 3.0, 7.0],[7,9]). This construct is the only way in Gaiwan one may obtain a tuple of more than one buffer.

The type of a `(retrive ...)` is the tuple of the types of the retrived buffers. For example `(retrive` z y z`)` yields a $(\texttt{int}[2], \texttt{float}[5], \texttt{int}[2])$. Note that there is no arrow ($\twoheadrightarrow$ nor $\rightarrow$) in the type of a retrive.

2.3.2. Literal Buffers

Sometimes a certain short buffer is inherent to a problem. In these cases, it would not make sense to store this buffer in the memory. Instead, the programmer may define this buffer inline in the program as `(buf`$_{S[m]}$ n_1 $\ldots$ n_m`)` with S a shape and n_1 to n_m values of that shape S. The type of such a construct is the 1-tuple containing the type $S[m]$ with m the number of elements in the literal buffer. Note that the annotation with shape S is required to type buffers of length zero as we cannot infer a type from the (non-exiting) concrete values.

name	type	#in	#out	Inspection	Execution
mapper	$\ldots[n] \to \ldots[n]$	1	1	✓	in-place parallel
reducer	$\ldots[n] \to \ldots[1]$	1	1	✓	in-place tree parallel
shaper	$\ldots[an+b] \to \ldots[cn+d]$	≥ 1	1	×	need not be executed

Table 1: Summary of the available base transformations and their properties.

2.3.3. Call: Applying Transformations

Now that we have our data we wish to apply our mappers, shapers and or reducers to it. Section 2.2.4 has shown how abstractions can group transformations and give them a name. With `(call` r a_1 $\ldots$ a_m`)` we may refer to these names. The last line of listing 6 shows a coordination plan using the `vectorLengths` abstraction. It takes two buffers from memory by using `retrive`. Then, it supplies these buffers to `vectorLengths` by using `call`, the two arguments of the abstraction are filled in with the values 0 and 0.

The type of abstractions that can be called with `(call` r v_1 $\ldots$ v_k`)` have the form $(S_{\text{in},1}, \ldots, S_{\text{in},k}) \to (B_1, \ldots, B_m) \twoheadrightarrow (B_{\text{out}})$. With S_1 to S_k the types of the parameters of the abstraction, $(B_1, \ldots, B_m)$ the types of the input buffers and B_{out} the type of the output buffer of the abstraction named r. The `(call ...)` fills in the parameters, so the remaining type $(B_1, \ldots, B_m) \twoheadrightarrow (B_{\text{out}})$ is the type of the `(call ...)` if the arguments match the shapes $S_{\text{in},1}$ to $S_{\text{in},k}$.

2.3.4. LetB: Naming Data

In our coordination plan data runs from left to right. To define intermediate results independent of this stream, we can use `letB`. This construct assigns a name to the buffer resulting from a computation. The syntax is (`letB` x `=` wl_1 `in` wl_2) with wl_1 and wl_2 programs (work lists). First, wl_1 is executed to get a single value buffer (D^v). Then, wl_2 is executed with a memory that is extended with $x \mapsto D^v$.

As an example, we may wish to compute the lengths of the vectors above after y-translation. Assuming `translate` is an abstraction containing a mapper that adds the value of its argument to each element of the buffer.

```
1  ( letB newY = (retrive y) ; (call translate 2)
2    in (retrive x newy);(call vectorlengths 0 0))
```

The type of (`letB` x `=` wl_1 `in` wl_2) is the type of wl_2 under the memory extended with the type of wl_1. For our example this means that the `letB` construct has the type of `(retrive x newy);(call vectorlengths 0 0)` typed under a memory extended with $\{newY \mapsto \texttt{int}[5]\}$.

2.4. Summary

We have three main transformations: mappers, reducers and shapers. These can be grouped and parameterized by abstractions. The coordination plan describes in what order the abstractions are applied to the memory to compute a final result.

Table 1 summarizes the differences between the main transformations. A mapper gets a single value and computes a new value to put in its place, this process is executed for every element in the input buffer. A reducer takes an accumulator value and combines it with a single value of the input to compute the accumulator value to be used with the next element. The result of the reducer is the buffer containing only the final accumulator. The body of mappers and reducers may inspect at most a single value of their input buffer. The shaper on the other hand accesses multiple elements of multiple buffers, but may not inspect the value of these elements. This restriction ensures that the "shaping" of a buffer only depends on the length of the buffer.

A program's coordination plan describes how the input is transformed using defined abstractions. The plan is a list of "constructs" that are executed one after the other. We separate these constructs with the `;` symbol. The output of the construct before the `;` becomes the input to what comes after it.

Name	Type	#in	#out	Function
`retrive`	Type of the retrived buffers	0	≥ 1	Read from memory
`letB`	Type of the inner work list	0	1	Name a result in memory
`call`	Output type of called abstraction	≥ 1	1	Execute an abstraction

Table 2: Summary of the constructs in the coordination language.

In Gaiwan, the input is a set of named buffers with elements of a fixed shape. We call this set the memory.

The constructs in the coordination language are shown in table 2. The `(retrive ...)` construct retrieves one or more buffers from our input memory. Intermediate results can be added to it temporarily with `(letB ...)`, which assigns a name to a buffer resulting from a sub-plan.

Abstractions form the basis of our computations, they are activated with the `(call ...)` construct. This construct returns at most one buffer. By combining `(retrive ...)` and `(letB ...)` we may construct outputs of more than one buffer.

We have described the constituents of the Gaiwan programming language. It is a small language that is just large enough to allow us to explore the size-polymorphic type system that goes with it.

3. Size polymorphic Type System

The central contribution of Gaiwan is its size polymorphic type system. Up to this point we have mostly looked at the operational aspects of our language. In listing 6 we saw a program that computed the distance from a list of points stored in two separate lists of x and y coordinates. Looking at the code in listing 6, we see that not every input memory can be valid. The program requires that the input memory has buffers named "x" and "y", else `(retrive x y)` would not work. Furthermore, these buffer should have the same length (n).

Our type system infers two properties of a program: (a) the type of the result (b) a list of constraints the input memory should satisfy. If the actual memory supplied to Gaiwan satisfies the constraints (b), the output will be the derived type (a). The program will not start if the memory fails to satisfy the constraints. The results (a) and (b) of our type checker are related. All free variables in the result type occur in the constraints. In the same context as our example, the program `(retrive x y) §(call vectorLengths 0 0)` has the type $\texttt{int}[n]$. The n used here is the n in the constraints: $\{x \mapsto \texttt{int}[n], y \mapsto \texttt{int}[n]\}$. Because the same n is used for the lengths of the buffers x and y they must have the same length.

In this section we will lay out our type system in detail. The main typing relation is $\vdash_{\mathrm{P}}$, which relates programs and parts thereof to their types and constraints. There are two variants of the relation. The first form is for buffer results, which may have constraints; the second is for transformations, which always have the empty set of constraints. Both forms are shown below.

$$\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} \mathit{term} : \overbrace{B|\mathcal{C}}^{\text{Buffer result}} \quad \text{or} \quad \Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} \mathit{term} : \overbrace{T|\emptyset}^{\text{Transformation result}}$$

Here, the variable environment is denoted by Γ, and $\mathcal{A}$ contains a list of abstractions available to *term*. After the colon (`:`) we find the type (B or T) and a set of constraints $\mathcal{C}$ that go with it. For transformations, this set of constraints will always be empty as a transformation does not access the memory directly, it only reads its input buffers. Terms returning buffers types B may have interacted with the memory through `(retrive ...)`, and may therefore have constraints.

In this section we will first describe the form of our types and expressions (section 3.1). Then, the typing rules will be presented. Our typing rules can be split up in four parts. First, we discuss transformation definitions (section 3.2). Second, we describe the coordination language that dictates the order in which these transformations run (section 3.3). Third, we look at the arithmetic expressions used in the body of

transformations (section 3.4). Finally, we combine all these rules to derive the type of instantiated programs (section 3.5).

3.1. Semantic Entities

The inductive definition of our formalization's semantic entities are listed in figure 3. Entities with a bar (e.g. $\overline{e}$) indicate zero or more occurrences of the over-lined non-terminal in a fixed order, a tuple. Most of our expression are S-expressions [12]. These are written between brackets, the first word between brackets signifies the kind of expression we are using. The other elements between the brackets are the arguments. In the examples in the previous section we used the syntax accepted by our implementation, in the formal sections of this paper we will use S-expressions instead.

The first entity in figure 3 is S, which we use to denote shapes. Gaiwan supports two primitive shapes: the shape of integral numbers (`int`) and of floating point numbers (`float`). We can also describe shapes abstractly with a variable name (x), for clarity we will always use uppercase letters for shape variables (e.g. A). Two shapes can be combined to become the shape of a pair by using the $\times$ operator. In our formalization we will distinguish between buffer types (B) and transformation types (T). Buffer types (B) combine a shape (S) with an affine length, we write $S[num \cdot n + num]$. Transformation types (T) relate two tuples of buffers $\overline{B}$ with an arrow $\twoheadrightarrow$. Our buffers types always have a corresponding set of constraints $\mathcal{C}$ going with them. These are mappings from buffer names x to buffer types. A pair of buffer types and constraints will be written as $\overline{B}|\mathcal{C}$. Buffer types and regular shapes can be used in the image of our variable environment Γ.

A data buffer D is written as $(\texttt{buf}_{S[num]}\ \overline{e})$, with $\overline{e}$ the elements of the buffer, S their shape and num the fixed number of elements. The $S[num]$ subscript is required to type the buffer if num is not strictly positive (see section 6.6). Note that we use num and not an affine function, the sizes of buffers D are always fixed integers. If the buffer only contains values (v), it is a value buffer, denoted by D^v. Of course every D^v is a D.

All of our transformations have a transformation type T as their first argument. Their last argument holds an expression containing the body e of the operation. Our three transformers are represented as follows:

- $(\texttt{mapr}\ T\ x_i\ x_d\ e)$ for mappers. The names assigned to in x_i and x_d are variable names to be used in the body e. x_i Contains the variable name for the index of the input buffer we are currently processing. x_d Holds the variable name that will be assigned the value of the data point at the current index.
- $(\texttt{redr}\ T\ x_d\ x_a\ e_0\ e_b)$ for reducers. We have x_d and x_a as variable names for respectively the value and the accumulator. The initial value of the accumulator is given in e_0. The body of the reducer is stored in e_b.
- $(\texttt{shpr}\ T\ x_i\ \overline{x_a}\ e)$ for shapers. Here, x_i denotes the index, and there is one x_a for each input buffer. The change in length is encapsulated in the type T.

We also need an administrative entity `(shpr* ...)` that acts as a marker, keeping track of buffers supplied to a shaper while running the program (see section 5.2). Transformers can be grouped in an abstraction `(abst ...)` that has a unique name r that can be called. Apart from a name, an abstraction may have scalar variables assigned to the names $\overline{x_a}$. Their shapes are integrated in the $\overline{S}$ part of the second argument of the abstractions ($\overline{S} \mapsto T$).

An instantiated program is written as $(\texttt{main}\ P\ \mathcal{M})$. Where P is the program and $\mathcal{M}$ is the memory the program is instantiated with. The memory $\mathcal{M}$ is a mapping from names x to concrete buffers D^v represented as a cons list with a colon ":" as delimiter. $\mathcal{M}$ is the set of elements in the supplied memory of the form $\overline{x \mapsto D^v}$. The program P has a list of abstractions $\mathcal{A}$ and a so-called work list wl.

The work list wl represents the coordination plan that consists of at least one step w. Possible steps include: transformers (t), literal buffers ($\overline{D}$), `call`s to abstractions by their name r and with arguments $\overline{e}$, `retrieve`ing

one more buffers by their name $\overline{x}$ or using a `letB`. A left associative ⨟ symbol fuses every two steps.

$$\texttt{(retrive } a\texttt{)} \fatsemi \texttt{(call } r_1 \ \ldots\texttt{)} \fatsemi \texttt{(call } r_2 \ \ldots\texttt{)} \equiv (\texttt{(retrive } a\texttt{)} \fatsemi \texttt{(call } r_1 \ \ldots\texttt{)}) \fatsemi \texttt{(call } r_2 \ \ldots\texttt{)}$$

The bodies of mappers, shapers and reducers as well as the arguments to `call` are arithmetic expressions e. These expressions are standard arithmetic expressions extended with `let`, `if`, pairs and `index`. `(index` x e`)` Takes the e-th element out of the buffer named x. In the following sections we will see that the type system ensures that `(index ...)` only occurs in shapers.

For sake of simplicity, we will assume that we do not use the same variable name x for different purposes. Any program can be made to adhere to this by using α-conversion [13].

Substitutions (σ) and evaluation context (E) and will be discussed in section 4 and section 5 respectively.

3.2. Transformations

The typing rules for transformations are shown in figure 4. These rules look as shown below.

$$\Gamma \mid \emptyset \vdash_{\text{P}} \texttt{(transformer ...)} : \mathsf{fresh}(T) | \emptyset$$

Note again that all these rules return an empty set of constraints ($\mathcal{C} = \emptyset$). All transformations already have a specified type T as their first argument. These rules thus validate if the declared type matches the actual type of the transformer. Applying fresh to T substitutes all free shape and size variables by fresh names to avoid collisions. There is a typing rule for each transformation. All of them will inspect the type of the body using the typing relation for arithmetic expressions ($\vdash_{\text{E}}$).

TT-Mapr dictates that a mapper has type $S_1[n] \twoheadrightarrow S_2[n]$ if its body e returns a S_2 if the index (x_i) is an `int`, and the value (x_d) is of shape S_1.

TT-Redr specifies how reducers are typed. The body of a reducer e_b takes an accumulator value of shape S_a and a data point of shape S_d to compute a new accumulator value of type S_a. We require that the initial value of the accumulator e_0 also has shape S_a. If the above requirements are met, we return a buffer containing one item (the last accumulator value), the type is $S_d[n] \twoheadrightarrow S_a[1]$

TT-Shpr specifies that the body e of a shaper should return a value of shape S_o under an extended environment. The environment Γ is extended with x_i mapping to `int` and the the x_{v1} to x_{vm} mapping to the argument buffer types. These buffer types are obtained from the type T in the first argument of the `(shpr` T `...)` construct. As motivated in section 2.2.3, shaper types may only contain free variables, this is enforced by *onlyFree* (see Appendix A.1, page 42). The TT-BufferShaper rule will be discussed in section 5.2.

Abstractions are not transformations, but by `call`ing them with arguments of the correct shape, a `(call ...)` construct becomes a transformer. The $\vdash_{\text{A}}$ relation is used to type abstractions, it relates an `(abst ...)` with a mapping from arguments shapes to a transformation type. Abstractions are always typed under an empty environment Γ, we therefore omit the environment in the $\vdash_{\text{A}}$ rules. Constraints are also not required as abstractions always have the empty constraint. The TT-Abstr rule types an abstraction's constituents $\overline{t}$ under the environments built by its argument list. The shapes of the arguments are derived from the second argument of `(abst ...)`. Abstractions cannot call other abstractions, so the constituents are typed with $\mathcal{A} = \emptyset$.

3.3. Coordination Plan

Now that we know how to type transformations, we can combine them using a coordination plan. Figure 5 shows the relevant typing rules. First, we will discuss our individual work item constructs, then we will explain how they are combined with TP-List and TP-LetB.

Types and constraints

$$
\begin{array}{rcl}
S & ::= & \texttt{int} \mid \texttt{float} \mid S \times S \mid x \\
B & ::= & S[num \cdot x + num] \\
T & ::= & \overline{B} \twoheadrightarrow \overline{B} \\
\mathcal{C} & ::= & (x \mapsto B) : \mathcal{C} \mid \emptyset \\
\Gamma & ::= & (x : B) : \Gamma \mid (x : S) : \Gamma \mid \emptyset
\end{array}
$$

Buffers

$$
\begin{array}{rcl}
D & ::= & (\texttt{buf}_{S[num]}\ \overline{e}) \\
D^v & ::= & (\texttt{buf}_{S[num]}\ \overline{v})
\end{array}
$$

Transformations

$$
\begin{array}{rcl}
\mathcal{A} & ::= & \overline{d} \\
d & ::= & (\texttt{abst}\ r\ (\overline{S} \to T)\ \overline{x_a}\ t) \\
t & ::= & (\texttt{mapr}\ T\ x_i\ x_d\ e) \\
 & \mid & (\texttt{redr}\ T\ x_d\ x_a\ e_0\ e_b) \\
 & \mid & (\texttt{shpr}\ T\ x_i\ \overline{x_a}\ e) \\
 & \mid & (\texttt{shpr*}\ \overline{D}\ \mathcal{M}) \quad \text{admin}
\end{array}
$$

Work lists

$$
\begin{array}{rcl}
wl & ::= & wl \fatsemi w \mid w \\
w & ::= & t \mid \overline{D} \\
 & \mid & (\texttt{call}\ r\ \overline{e}) \\
 & \mid & (\texttt{retrive}\ \overline{x}) \\
 & \mid & (\texttt{letB}\ x\ wl\ wl)
\end{array}
$$

Instantiated program

$$
\begin{array}{rcl}
\mathbb{R} & ::= & (\texttt{main}\ P\ \mathcal{M}) \\
P & ::= & (\texttt{prog}\ \mathcal{A}\ wl) \\
\mathcal{M} & ::= & (x \mapsto D^v) : \mathcal{M} \mid \emptyset
\end{array}
$$

Variables

$$
x ::= \textit{variable not otherwise mentioned}
$$

Arithmetic expressions

$$
\begin{array}{rcl}
e & ::= & (\texttt{+}\ e\ e) \mid (\texttt{-}\ e\ e) \mid (\texttt{let}\ x\ e\ e) \\
 & \mid & (\texttt{8}\ e\ e) \mid (\texttt{/}\ e\ e) \mid (\texttt{if}\ e\ e\ e) \\
 & \mid & (\texttt{tuple}\ e\ e) \mid (\texttt{fst}\ e) \mid (\texttt{snd}\ e) \\
 & \mid & (\texttt{index}\ x\ e) \\
 & \mid & num \mid x \\
v & ::= & (\texttt{tuple}\ v\ v) \mid num
\end{array}
$$

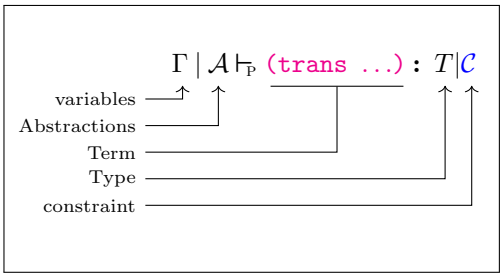
Evaluation contexts

$$
\begin{array}{rcl}
E_w & ::= & (\texttt{main}\ (\texttt{prog}\ \mathcal{A}\ E_{wl})\ \mathcal{M}) \\
E_{wl} & ::= & \cdot \mid E_{wl} \fatsemi w \\
E_R & ::= & E_B \mid (\texttt{shpr*}\ E_B\ \mathcal{M}) \\
 & \mid & \overline{D^v} \fatsemi (\texttt{call}\ r\ (\overline{v}, E, \overline{e})) \\
 & \mid & \overline{D^v} \fatsemi (\texttt{redr}\ T\ x_d\ x_a\ E\ e_b) \\
E_B & ::= & (\overline{D^v}, (\texttt{buf}_{S[num]}\ \overline{v}, E, \overline{e}), \overline{D}) \\
E & ::= & \cdot \mid (\texttt{tuple}\ E\ e) \mid (\texttt{tuple}\ v\ E) \\
 & \mid & (\texttt{+}\ E\ e) \mid (\texttt{+}\ v\ E) \\
 & \mid & (\texttt{-}\ E\ e) \mid (\texttt{-}\ v\ E) \\
 & \mid & (\texttt{*}\ E\ e) \mid (\texttt{*}\ v\ E) \\
 & \mid & (\texttt{/}\ E\ e) \mid (\texttt{/}\ v\ E) \\
 & \mid & (\texttt{let}\ x\ E\ e) \\
 & \mid & (\texttt{fst}\ E) \mid (\texttt{snd}\ E) \\
 & \mid & (\texttt{if}\ E\ e\ e) \mid (\texttt{index}\ x\ E)
\end{array}
$$

Unifiers

$$
\begin{array}{rcl}
\sigma & ::= & \sigma, u \mid u \\
u & ::= & \langle x/S \rangle \\
 & \mid & \langle \lambda(a \cdot x_1 + b)/(f_1(a,b) \cdot x_2 + f_2(a,b)) \rangle
\end{array}
$$

Figure 3: Syntax of Gaiwan. Over lined identifiers (e.g. $\overline{d}$) represent an ordered tuple of zero or more occurrences.



TT-MAPR

$$\frac{(x_i \mapsto \texttt{int}), (x_d \mapsto S_1), \Gamma \vdash_{\text{E}} e : S_2 \qquad T = (S_1[n]) \twoheadrightarrow (S_2[n])}{\Gamma \mid \emptyset \vdash_{\text{P}} (\texttt{mapr}\ T\ x_i\ x_d\ e) : \texttt{fresh}(T)|\emptyset}$$

TT-REDR

$$\frac{(x_a \mapsto S_a), (x_d \mapsto S_d), \Gamma \vdash_{\text{E}} e : S_a \qquad \Gamma \vdash_{\text{E}} e_0 : S_a \qquad T = (S_d[n]) \twoheadrightarrow (S_a[1])}{\Gamma \mid \emptyset \vdash_{\text{P}} (\texttt{redr}\ T\ x_d\ x_a\ e_0\ e_b) : \texttt{fresh}(T)|\emptyset}$$

TT-SHPR

$$\frac{\begin{array}{c}(x_i \mapsto \texttt{int}), (x_{v1} \mapsto S_1[e_1]), \ldots, (x_{vm} \mapsto S_m[e_m]), \Gamma \vdash_{\text{E}} e : S_o \\ T = (S_1[e_1], \ldots, S_m[e_m]) \twoheadrightarrow (S_o[e_o]) \qquad \forall i.onlyFree(S_i)\end{array}}{\Gamma \mid \emptyset \vdash_{\text{P}} (\texttt{shpr}\ T\ x_i\ (x_{v1}\ \ldots\ x_{vm})\ e) : \texttt{fresh}(T)|\emptyset}$$

TT-BUFFERSHPR

$$\frac{\vdash \mathcal{M}_s : \Gamma_s \qquad \Gamma_s \vdash D : B}{\Gamma \mid \emptyset \vdash_{\text{P}} (\texttt{shpr*}\ (D)\ \mathcal{M}_s) : (B)|\emptyset}$$

TT-ABST

$$\frac{(x_1 \mapsto S_1) : \ldots : (x_m \mapsto S_m) \mid \emptyset \vdash_{\text{P}} t : T|\emptyset}{\vdash_{\text{A}} (\texttt{abst}\ r\ ((S_1, \ldots, S_m) \to T)\ x_1\ \ldots\ x_m\ t) : (S_1, \ldots, S_m) \to T}$$

Figure 4: Typing rules for transformations and abstractions. These rules all return an empty set of constraints and may themselves not use abstractions. Note that all variable environments Γ in this picture stem from TT-ABSTR

Let us start with the simplest rule: TP-Buf. This rule specifies how a tuple of literal buffers is typed. The notation $\vdash D_i : B_i$ specifies that the buffer D_i has type B_i under the empty environment (See also TM-Buffer in figure A.13). Expressions in buffers cannot use variables at the top level, and are therefore always typed under the empty environment $\Gamma = \emptyset$. Literal buffers do not interact with the memory, as a consequence, their type has no constraints ($\emptyset$).

We use `(call ...)` to invoke an abstraction on the current buffers. If we `call` an abstraction of type $\overline{S} \mapsto T$ with values of shape $\overline{S}$, we get a transformation of type T as result. Our TP-Call rule codifies this, it validates that the arguments are of the correct shape with $\vdash_{\mathrm{E}}$ and returns the type T. Note that there are no constraints associated with a `call`. There are no interactions with the memory from within calls. We only read the input buffer.

Our `(retrive ...)` construct takes buffers from the memory $\mathcal{M}$. If we say that `(retrive` a`)` has type $(A[n])$ then it must be the case that $\mathcal{M}$ has the key a mapping to the type $A[n]$. To enforce this we return a set of constraints. For every unique returned variable name, we create a mapping to a fresh type B_i. The resulting type is the list of fresh types in the same order as the variables. The mapping we built is returned as the accompanying constraints $\mathcal{C}$. Note how all free variables in the returned type are also in the constraints by construction (lemma 1, page 25).

In section 2.2.4 we briefly touched on typing ⨟-delimited work items. TP-List formalizes this for any work list $wl \fatsemi w$ of length > 1.[1] Note that ⨟ is left associative. The rule requires that the work list wl results in a tuple of buffers $(\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl : \overline{B_1} | \mathcal{C})$ to which the work item w will be applied. This work item w must have a transformation type T. We derive the final type using the join function, defined in TM-Join. It relies on unification, which we will explore in section 4. Our join fuses the buffer types $\overline{B_1}$ and the transformation $T = \overline{B_2} \twoheadrightarrow B_3$. Let us use an example with an input buffer of type $B_1 = (\texttt{int}[3n])$ and a transformation of type $T = A[m] \twoheadrightarrow A[2m]$. First, we unify $\overline{B_1}$ and $\overline{B_2}$, the resulting unifier σ ensures that $\sigma(\overline{B_1}) = \sigma(B_2)$. For the example we get that A should be an `int` and that m is $3n$. We then apply the unifier to the return type of T, and get that $A[2m]$ becomes $\texttt{int}[6n]$.

We can add items to the memory $\mathcal{M}$ in the scope of a `(letB` x wl_1 wl_L`)` construct. TP-LetB shows how this is done. We type the two parts wl_1 and wl_L, and combine the results. First, we analyze wl_1, the work list that will compute the buffer to which the variable x will be bound. It can only return a single buffer, hence the return type is of the form $(B_1)|\mathcal{C}_1$. The constraints $\mathcal{C}_1$ of wl_1 describe requirements for the memory used in wl_1, these all stem from uses of `(retrive ...)` in wl_1. We may now use x in wl_L, but x is neither added to Γ nor to $\mathcal{C}$ when typing wl_L. This diverges from what someone would expect based on other type systems. The environment does not need to be extended to type the body of the `letB`. Instead, the requirement for the existence of x moves upward in the program in the constraints $\mathcal{C}_L$ from the point where it is used (`(retrive` x`)`). Any constraint requiring the existence of x is removed by japply when typing a `(letB` x `...)`.

TT-Japply describes how we merge the constraints of the value (wl_1) and the body (wl_L) of a `(letB` x wl_1 wl_L`)` construct. The result of the action is $\overline{B_1}|\mathcal{C}_1 \cup (\mathcal{C}_L \setminus \{x\})$ to which a unifier σ is applied. We keep all the constraints of wl_1 and wl_L except for the constraints placed on x. These constraints are only applicable inside the `(letB ...)`. Since the value for x is derived from other buffers in wl_1, we can transform the constraints on x to constraints on the variables used in wl_1. This is what the unifier σ does, it ensures $\sigma(\mathcal{C}_L[x]) = \sigma(B_1)$. The buffer type assigned to usages of the buffer named x in wl_L is thus transformed to match B_1, the buffer type that results from wl_1. Additionally, σ ensures that if any other named buffer is used in both wl_1 and wl_L, they are assigned an identical type. By applying σ to the merged constraints, the constraints on x are pushed into the constraints $\mathcal{C}_1$. Let us look at an example term `(letB x = (buf`$_{\texttt{int[6]}}$ `1 2 3 4 5 6) in ((retrive x) ⨟(call get3rds)))`, with `get3rd` an abstraction of type $() \to A[3p] \twoheadrightarrow A[p]$ that selects every third element of the input buffer. The type of wl_1 is $(\texttt{int}[6])|\emptyset$, the body of the `letB` has type $A[n]|\{x \mapsto A[3m]\}$. We will use japply $(x \mapsto \texttt{int}[6], \emptyset, (A[m]), \{x \mapsto A[3m]\})$ to

[1] A work list of length one must be a simple work item and is typed by one of the other rules in figure 5.

derive the type of the whole term. A unifier is obtained from unifying $\{x \mapsto \texttt{int}[6]\}$ and $\{x \mapsto A[3m]\}$ which yields that A should be `int` and that m is 2. The total type is thus $\sigma(A[m]) = \texttt{int}[2]$ and the accompanying constraint is empty as x is removed from the constraints of wl_L.

3.4. Arithmetic Expressions

The actions on the data points in our buffers are described using an arithmetic expression language. This is the final component we need to discuss before we can look at the type of an instantiated program. The typing rules for the arithmetic expression language are the standard typing rules one would expect. Appendix A.3 on page 42 shows them in full. Binary arithmetic operations return values of the same numeric shape as the supplied arguments (TE-BinOp). Tuples, constructed with `(tuple` e_1 e_2`)` have a product shape $S_1 \times S_2$ where S_1 is the shape of the first element and S_2 that of the second. By using `(fst` e`)` and `(snd` e`)` we can respectively get the first (e_1) and second (e_2) back out of a tuple e in their original shape (TE-Fst,TE-Snd). Our `(if` e_c e_t e_f`)` implements an if-expression, if the condition e_c is non-zero the e_t gets executed, otherwise e_f gets executed. The condition must have type `int`, the branches e_t and e_f must have the same type (TE-If).

Expressions may contain variables. These reside in an environment Γ that assigns names to values of shape S and buffers of type $B = S[num_{la} \cdot n_l + num_{lb}]$. The former kind of bindings can be traced back to following sources:

- argument names of abstractions ($x_1 \dots x_m$ in TT-Absr);
- argument names of mappers (x_i and x_d in TT-Mapr);
- argument names of reducers (x_d and x_a in TT-Redr);
- argument names of shapers (x_i in TT-Shpr);
- arithmetic expression let bindings (x TE-Let),

Non-buffer elements of Γ are accessed by placing their assigned name instead of the value, TE-Var will type these names correctly. Variable-buffer bindings only originate from shapers ($x_{v1} \dots x_{vm}$ in TT-Shpr). They are only accessible using the `(index ...)` construct, which selects a single element from a buffer TE-Index.

3.5. Instantiated Programs

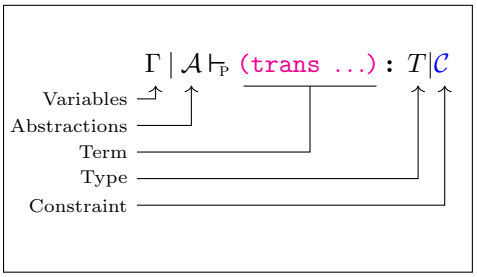
Now that we have typed all the parts of a program, we investigate how a full program is typed. Figure 6 shows the typing rules for the remaining semantic entities.

A full program is written as `(prog` $\mathcal{A}$ wl`)` with $\mathcal{A}$ a list of abstractions and wl a work list. TP-Prog mandates two things for well-typed programs. First, all the abstractions in $\mathcal{A}$ must be well-typed, that is the type indicated in their second argument must correspond with their implementation. Second, the work list must be typeable under the empty environment using the rules from figure 5 with $\mathcal{A}$ as abstractions list. The type of the program is the type of this work list.

If we want to execute a program, we must give it a memory by using the `(main` p $\mathcal{M}$`)` construct. The type of the program p must be a buffer type B_T with a set of constraints $\mathcal{C}_T$. TP-Validate checks if the memory $\mathcal{M}$ adheres to the constraints $\mathcal{C}_T$ and transforms B_T to a concrete buffer type. To do this it requires that $\mathcal{M}$ contains all needed keys ($\mathsf{dom}(\mathcal{M}) \subseteq \mathsf{dom}(\mathcal{C}_T)$). Because $\mathcal{M}$ is concrete, the unifier merging $\mathcal{M}$ and $\mathcal{C}$ will assign concrete values to all free variables in $\mathcal{C}$. By construction (see lemma 2), the free variables of B_T are a subset of those of $\mathcal{C}_T$. Hence, applying the unifier to B_T will yield a concrete type, the type the instantiated program is executed with $\mathcal{M}$.

TP-Buf

$$\frac{\forall i.\ \vdash D_i : B_i}{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} (D_1, \ldots, D_n) : (B_1, \ldots, B_n) | \emptyset}$$



TP-Retrive

$$\frac{\{x'_1, \ldots, x'_m\} = \bigcup_{i=1}^{m} \{x_{\min\{j \mid x_j = x_i\}}\} \qquad \forall i \in \{1, \ldots, m\}. B_{x'_i} \text{ is fresh}}{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} (\texttt{retrive}\ x_1\ \ldots\ x_n) : (B_{x_1}, \ldots, B_{x_n}) \left| \{x'_1 \mapsto B_{x'_1}, \ldots, x'_n \mapsto B_{x'_m}\} \right.}$$

TP-Call

$$\frac{(\texttt{abst}\ r\ ((S_1, \ldots, S_n) \to T)\ \ldots) \in \mathcal{A} \qquad \forall i \in \{1, \ldots, n\}. \Gamma \vdash_{\mathrm{E}} e_i : S_i}{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} (\texttt{call}\ r\ e_1\ \ldots\ e_n) : \mathsf{fresh}(T) | \emptyset}$$

TP-List

$$\frac{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl : \overline{B_1} | \mathcal{C}_1 \qquad \Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} w : T_2 | \emptyset \qquad B | \mathcal{C}_J = \mathsf{join}\,(\overline{B_1} | \mathcal{C}_1\ , T_2)}{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl \fatsemi w : B | \mathcal{C}_J}$$

TM-Join

$$\frac{\mathcal{C} = \sigma(\mathcal{C}_1) \qquad \sigma = \mathsf{unify}(\overline{B_1}, \overline{B_2}) \qquad \overline{B} = \sigma(\overline{B_3}) \qquad FV(\overline{B_3}) \subseteq FV(\overline{B_2})}{\overline{B} | \mathcal{C} = \mathsf{join}\,(\overline{B_1} | \mathcal{C}_1\ , (\overline{B_2} \twoheadrightarrow \overline{B_3}))}$$

TP-LetB

$$\frac{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl_1 : (B_1) | \mathcal{C}_1 \qquad \Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl_L : \overline{B_L} | \mathcal{C}_L \qquad \overline{B} | \mathcal{C} = \mathsf{japply}\,(x \mapsto B_1, \mathcal{C}_1, \overline{B_L}, \mathcal{C}_L)}{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} (\texttt{letB}\ x\ wl_1\ wl_L) : \overline{B} | \mathcal{C}}$$

TT-Japply

$$\frac{\sigma = \mathsf{unify}\,(\mathcal{C}_1 \cup \{x \mapsto B_1\}, \mathcal{C}_L)}{\sigma\,(\overline{B_2} | \mathcal{C}_1 \cup (\mathcal{C}_L \setminus \{x\})) = \mathsf{japply}\,(x \mapsto B_1, \mathcal{C}_1, \overline{B_L}, \mathcal{C}_L)}$$

Figure 5: Typing rules for programs and work lists

TP-Prog

$$\frac{\forall d_i \in \mathcal{A}. \vdash_{\mathrm{A}} d_i : T \qquad \emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl : B | \mathcal{C}}{\Gamma \mid \emptyset \vdash_{\mathrm{P}} (\texttt{prog}\ \mathcal{A}\ wl) : B | \mathcal{C}}$$

TP-Main

$$\frac{\emptyset \mid \emptyset \vdash_{\mathrm{P}} p : B_T | \mathcal{C}_T \qquad B_2 = \mathsf{validate}(\mathcal{M}, B_T | \mathcal{C}_T)}{\vdash (\texttt{main}\ p\ \mathcal{M}) : B_2}$$

TP-Validate

$$\frac{FV(B) \subseteq FV(\mathcal{C}) \qquad \mathsf{dom}(\mathcal{C}) \subseteq \mathsf{dom}(\mathcal{M}) \qquad \sigma = \mathsf{unify}\left(\mathcal{M}^B\big|_{\mathsf{dom}(\mathcal{C})}, \mathcal{C}\right)}{\sigma(B) = \mathsf{validate}(\mathcal{M}, B | \mathcal{C})}$$

Figure 6: Typing rules for instantiated programs. $\mathcal{M}^B$ represents a memory with concrete buffers, $\mathcal{M}$ represents the type of this buffer ($\mathcal{M}^B \vdash \mathcal{M}$).

$$\text{TM-UNIFY}\quad \frac{\sigma = \mathsf{unify}_1(S_L[e_L], S_R[e_R]) \qquad \sigma_O = \mathsf{unify}\,(\mathsf{map}(\sigma, T_{IL}\ldots), \mathsf{map}(\sigma, T_{IR}\ldots))}{\sigma_O, \sigma_O(\sigma) = \mathsf{unify}\,((S_L[e_L], T_{IL}\ldots), (S_R[e_R], T_{IR}\ldots))}$$

$$\text{TM-UNIFYNIL}\quad \frac{}{\emptyset = \mathsf{unify}((),())}$$

$$\text{TM-UNIFY1}\quad \frac{\sigma = \mathsf{unify}_S(S_L, S_R) \qquad \sigma_e = \mathit{solve}(e_L, e_R)}{\sigma_e, \sigma = \mathsf{unify}_1\,(S_L[e_L], S_R[e_R])}$$

$$\text{TM-UNIFYM}\quad \frac{\mathsf{dom}\,(\mathcal{C}_1) \cap \mathsf{dom}\,(\mathcal{C}_2) = \{x_1, \ldots, x_m\} \qquad \sigma = \mathsf{unify}\,((\mathcal{C}_1[x_1], \ldots, \mathcal{C}_1[x_m]), (\mathcal{C}_2[x_1], \ldots, \mathcal{C}_2[x_m]))}{\sigma = \mathsf{unify}(\mathcal{C}_1, \mathcal{C}_2)}$$

Figure 7: Rules for unification, e_L and e_R are shorthand for $num_{L1} \cdot n_L + num_{L2}$ and $num_{R1} \cdot n_R + num_{R2}$ respectively. Composition of 2 unifiers with comma means applying them right to left when they are used.

4. Unification

Up to now we have seen that unification plays an important role in Gaiwan. It is used to join work list items, to type `(letB ...)` constructs, and it is used to get the type of an instantiated program.

Figure 7 shows the rules for unifying tuples of buffer types. We see that TM-UNIFY unifies the shapes and the sizes of the first element in either tuple. Then, it applies the resulting unifier to the rest of the lists and it recursively unifies it. When all elements are unified, TM-UNIFYNIL returns an empty unifier. This is a like Robinson's unification of lists[14].

For shape variables unification works as one would expect. It is implemented in TM-UNIFY1. We will lay out the inner workings of this function in section 4.1. The unification of buffer sizes is non-trivial. It is one of the main contributions of this paper, we have codified it in *solve*. We discuss the definition of *solve* in section 4.2.

4.1. Shapes

We use the notation $\langle A/S \rangle$ to indicate that the shape variable A should be replaced by the shape S. S does not need to be a shape variable itself, it may be a concrete shape, or a compound shape, which in turn may contain variables.

We use the standard recursive unification process to find the most general unifier for the two shapes that are supplied. Many algorithms exist for this problem, such as Robinson unification [14] and derivative works[15]. The result of these algorithms is a substitution σ, a collection of replacements of the form $\langle A/S \rangle$.

If unify_1 fails, σ in TM-UNIFY has no value and thus unify fails and so does anything that depends on it. Note that unify_1 may succeed with the empty set as result, this is not considered a failure.

4.2. Sizes

The sizes of buffers in Gaiwan are affine functions in one variable ($an + b$, with $a, b \in \mathbb{Z}$). When unifying buffers, these sizes must be unified as well. The result of our unification should be a most general unifier that equates the two given affine functions. We will use the notation $\langle \lambda(a \cdot n + b)/(f_1(a,b) \cdot l + f_2(a,b)) \rangle$ to denote a substitution that replaces the affine function $a_c \cdot n + b_c$ with $f_1(a_c, b_c) \cdot l + f_2(a_c, b_c)$ for any fixed a_c and b_c in $\mathbb{Z}$. Whenever such a substitution is applied to a buffer size, it should hold that the valid concrete sizes that satisfy the new affine function also satisfy the old function. We say that a substitution is sound if this is the case.

Definition 1 (Sound size substitution)**.**
A substitution $\langle\lambda(a \cdot n + b)/(f_1(a,b) \cdot l + f_2(a,b))\rangle$ *is said to be sound w.r.t. an affine function* $a_c n + b_c$ *iff it holds that* $\forall l \in \mathbb{Z}.\exists n \in \mathbb{Z}.f_1(a_c,b_c) \cdot l + f_2(a_c,b_c) = a_c n + b_c$

To define *solve* in TM-UNIFY1 we aim to unify two affine functions: $a_1 n + b_1$ and $a_2 m + b_2$. We must find a unifier that makes them identical. Our unifier consists of two substitutions: $\sigma_s = \langle\lambda(a \cdot n + b)/(f_{s1}(a,b) \cdot l + f_{s1}(a,b))\rangle$ and $\sigma_t = \langle\lambda(a \cdot m + b)/(f_{t1}(a,b) \cdot l + f_{t2}(a,b))\rangle$. One for lengths with variable n the other for the lengths with variable m. Both return affine functions in the variable l.

It should thus hold that $\langle\lambda(a \cdot n + b)/(f_{s1}(a,b) \cdot l + f_{s2}(a,b))\rangle$ $(a_1 n + b_1)$ is equal to $\langle\lambda(a \cdot m + b)/(f_{t1}(a,b) \cdot l + f_{t2}(a,b))\rangle$ $(a_2 m + b_2)$, or put differently:

$$f_{s1}(a_1,b_1) \cdot l + f_{s2}(a_1,b_1) = f_{t1}(a_2,b_2) \cdot l + f_{t2}(a_2,b_2)$$

must hold for any l, in particular zero and one, so we may conclude that

$$f_{s2}(a_1,b_1) = f_{t2}(a_2,b_2) \quad \wedge \quad f_{s1}(a_1,b_1) = f_{t1}(a_2,b_2).$$

Our two substitutions must be sound with respect to their original affine functions, so the following should hold with $a_s := f_{s1}(a_1,b_1) = f_{t1}(a_2,b_2)$ and $b_s := f_{s2}(a_1,b_1) = f_{t2}(a_2,b_2)$.

$$\forall l \in \mathbb{Z}.\ \exists n, m \in \mathbb{Z}.\ a_1 n + b_1 = a_s l + b_s = a_2 m + b_2$$

Take arbitrary l, and the corresponding n and m. It holds that $a_1 n + b_1 = a_s l + b_s = a_2 m + b_2$. We will try to express n and m in terms of l and the known values a_1, b_1, a_2 and b_2. With this knowledge we will derive the exact definition of σ_s and σ_t.

First, we isolate m and n:

$$\begin{aligned} m &= (a_1 n + b_1 - b_2)/a_2 \\ n &= \frac{a_s}{a_1} l + \frac{b_s - b_1}{a_1} = ul + v \qquad \text{with } u = \frac{a_s}{a_1} \text{ and } v = \frac{b_s - b_1}{a_1} \end{aligned}$$

Combining the previous two results gives us the following

$$m = \frac{a_1 n + b_1 - b_2}{a_2} = \frac{a_1(ul+v)}{a_2} + \frac{b_1 - b_2}{a_2} = \frac{a_1 u}{a_2} l + \frac{a_1 v + b_1 - b_2}{a_2}$$

This results holds for $l = 0$ and for non-zero l (with possibly different values for m). As m must always be whole, the constant part $\frac{a_1 v + b_1 - b_2}{a_2}$ must be whole as well. The same holds for the value multiplied by l, so $\frac{a_1 u}{a_2}$ must also be whole. We thus need that $a_1 v + b_1 - b_2$ and $a_1 u$ are both divisible by a_2. This first divisibility holds if and only if $\exists v.a_1 v = b_2 - b_1 \mod a_2$. The second requirement is satisfied iff $a_1 u$ is a multiple of a_2. From this we can derive values for u and v.

$$u = \frac{\text{lcm}(a_1,a_2)}{a_1} = \frac{a_1 a_2}{a_1 \cdot \text{gcd}(a_1,a_2)} = \frac{a_2}{\text{gcd}(a_1,a_2)}$$

$$\begin{aligned} v &= \text{the smallest for } v \text{ of } a_1 v = b_2 - b_1 \mod a_2 \\ &= \begin{cases} 0 & b_2 = b_1 \\ a_1^{-1}(b_2 - b_1) & \text{If the multiplicative inverse of } a_1 \text{ modulo } a_2 \text{ exists} \\ \text{error} & \text{otherwise} \end{cases} \end{aligned}$$

Now we have enough information about how n and m relate to l that we can derive f_{s1}, f_{s2}, f_{t1} and f_{t2} by simple substitution.

$$\begin{aligned} a_c n + b_c &= a_c(ul + v) + b_c \\ &= \underbrace{a_c u}_{f_{s1}(a_c,b_c)}\ l + \underbrace{a_c v + b_c}_{f_{s2}(a_c,b_c)} \\ &\implies \boxed{\sigma_s = \langle \lambda(a \cdot n + b)/(au \cdot l + (av + b)) \rangle} \end{aligned}$$

$$\begin{aligned} a_c m + b_c &= a_c \left(\frac{a_1 u}{a_2} l + \frac{a_1 v + b_1 - b_2}{a_2} \right) + b_c \\ &= \underbrace{\frac{a_c a_1 u}{a_2}}_{f_{t1}(a_c,b_c)}\ l + \underbrace{\frac{a_c a_1 v + b_1 - b_2}{a_2} + b_c}_{f_{t2}(a_c,b_c)} \\ &\implies \boxed{\sigma_r = \left\langle \lambda(a \cdot m + b) \middle/ \left(\tfrac{a a_1 u}{a_2} \cdot l + \tfrac{a(a_1 v + b_1 - b_2)}{a_2} + b \right) \right\rangle} \end{aligned}$$

The *solve* function as used in TM-UNIFY1 of figure 7 unifies two affine functions. It is defined as follows. There are four possible cases:

- The affine functions are in different variables, in which case *solve* returns σ_s, σ_t as derived above. The σ_s unifier will transform all the buffer sizes of the first argument. The buffers in the second argument will be transformed by σ_t.
- The affine functions are in the same variable (let it be n) and
  - are identical: *solve* returns $\emptyset$. The elements are already the same, nothing needs to happen.
  - differ and intersect in a point with whole coordinates: *solve* returns $\langle \lambda(a \cdot n + b)/(0 \cdot l + x) \rangle$, with x the value of n at the intersection and l fresh. This effectively replaces all occurrences of n with a numeric value.
  - else: *solve* fails

4.3. Example

We will now look at concrete example of our unification.

$$\mathsf{unify}((A \times \texttt{int})[2 \cdot n_1 + 4], B[4 \cdot n_2])$$

We first unify the shapes with unify_S and get that $\langle B/(A \times \texttt{int}) \rangle$ as most general unifier. Next we look at the lengths, we need to find the result of $solve(2 \cdot n_1 + 4, 4 \cdot n_2)$. From the formulae above we know that $u = 2$ and $v = 0$ thus:

- $\sigma_s = \langle \lambda(a \cdot n_1 + b)/(a \cdot 2 \cdot l + (a \cdot 0 + b)) \rangle = \langle \lambda(a \cdot n_1 + b)/(a \cdot 2 \cdot l + b) \rangle$
- $\sigma_r = \left\langle \lambda(a \cdot n_2 + b) \middle/ \left(\frac{a \cdot a_1 \cdot 2}{a_2} \cdot l + \frac{a(a_1 \cdot 0 + b_1 - b_2)}{a_2} - b \right) \right\rangle$
  $= \left\langle \lambda(a \cdot n_2 + b) \middle/ \left(\frac{a \cdot 2 \cdot 2}{4} \cdot l + \frac{a(4-0)}{4} - b \right) \right\rangle$
  $= \langle \lambda(a \cdot n_2 + b)/(a \cdot l + a - b) \rangle$

We now apply these substitutions to our original sizes to verify that we indeed get the same result.

- $\sigma_s(2 \cdot n_1 + 4) = (2 \cdot 2)l + 4 = 4 \cdot l + 4$

- $\sigma_r(4 \cdot n_2) = \langle \lambda(a \cdot n_2 + b)/(a \cdot l + a - b) \rangle\,(4 \cdot n_2) = 4 \cdot l + 4$

We may also apply these substitutions to other affine functions:

- $\sigma_s(n_1) = \langle \lambda(a \cdot n_1 + b)/(a \cdot 2 \cdot l + b) \rangle\,(n_1) = (1 \cdot 2 \cdot l + 0) = 2l$
- $\sigma_r(n_2) = \langle \lambda(a \cdot n_2 + b)/(a \cdot l + a - b) \rangle\,(n_2) = (1 \cdot l + 1 - 0) = l + 1$

5. Evaluation rules

In this section we explain how programs are evaluated with a memory $\mathcal{M}$. The $\rightsquigarrow_{\text{P}}$ reduction relation defines this process. An instantiated program is written as `(main (prog` $\mathcal{A}$ wl`)` $\mathcal{M}$`)`. The notation below indicates that the instantiated program before the $\rightsquigarrow_{\text{P}}$ arrow is transformed into the program after the arrow in one step.

$$\texttt{(main (prog } \mathcal{A}\ wl\texttt{)}\ \mathcal{M}\texttt{)} \rightsquigarrow_{\text{P}} \texttt{(main (prog } \mathcal{A}\ wl'\texttt{)}\ \mathcal{M}\texttt{)}$$

Most of the rules will be of the form $E_w\,[wl_1] \rightsquigarrow_{\text{P}} E_w\,[wl_2]$ where E_w is an evaluation context that extracts the first few elements from the work list wl in the program. The explicit definition of E_w can be found in figure 3. For any (fixed) $\mathcal{A}$, $\mathcal{M}$ and w_1 to w_n the following equivalence holds. For clarity, we have underlined the matched parts of the work list.

$$E_w\,[\underline{wl_1}] \rightsquigarrow_{\text{P}} E_w\,[\underline{wl_2}] \quad \equiv \quad \begin{array}{l}\texttt{(main (prog } \mathcal{A}\ \underline{wl_1}\ \texttt{;}\ w_1\ \texttt{;}\ w_2\ \texttt{;}\ \ldots\ \texttt{;}\ w_n\texttt{)}\ \mathcal{M}\texttt{)} \\ \rightsquigarrow_{\text{P}} \texttt{(main (prog } \mathcal{A}\ \underline{wl_2}\ \texttt{;}\ w_1\ \texttt{;}\ w_2\ \texttt{;}\ \ldots\ \texttt{;}\ w_n\texttt{)}\ \mathcal{M}\texttt{)}\end{array}$$

Because the same instance E_w occurs on both sides of $\rightsquigarrow_{\text{P}}$, the $\mathcal{A}$, $\mathcal{M}$ and w_1 to w_n remain unchanged. None of our reduction rules $\rightsquigarrow_{\text{P}}$ will ever modify $\mathcal{A}$, these are the defined abstraction of the program, and are not affected by running it. The memory $\mathcal{M}$ remains fixed as well – programs can only query elements from the memory, they do not alter it. The `(letB ...)` construct adds elements to the memory $\mathcal{M}$ by recursively looking if a step can be taken for the program with the extended $\mathcal{M}$ (this will be discussed in detail in section 5.3).

We will go through the evaluation rules from low-level to high-level. First, we look at the expressions in the bodies of our transformations. Then we move on to how our transformations are applied. Finally, we look at how transformations are combined using the coordination plan.

5.1. Arithmetic Expressions

The evaluation rules for transformations and our coordination language rely on the fact that there is some way that non-value expressions in buffers are transformed into values. All the rules we will see in the following sections use D^v to denote buffers of values. They also use v for initial values of accumulators and arguments to `(call .)` In this section we will discuss the reduction rules that transform expressions e into values v and by extension D into D^v.

The E-Other rule (figure 8) applies the arithmetic expression reduction relation $\hookrightarrow_{\text{E}}$ to the first non-valued arithmetic expression of an instantiated program. This rule uses E_w to select the first item(s) of the work list. In turn, E_R selects the first non-value arithmetic expression of these items. As figure 3 shows, E_R has four forms that all select the first non-values arithmetic expression:

- $E_B = (\overline{D^v}, \texttt{(buf}_{S[num]}\ \overline{v}, \cdot, \overline{e}\texttt{)}, \overline{D})$, which selects the left most non-value buffer element in a tuple of buffers.
- `(shpr*` E_B $\mathcal{M}$`)`, which uses E_B to select the first non-value element in the buffer contained in a `shpr*` (see section 5.2).
- $\overline{D^v}$ `;` `(call` r $(\overline{v}, \cdot, \overline{e})$`)`, which selects the first non-valued argument to a `call` applied to a buffer that only contains values.

$$\text{E-Other}\ \frac{e_1 \hookrightarrow_{\text{E}} e_2}{E_w\,[E_R[e_1]] \rightsquigarrow_{\text{P}} E_w\,[E_R[e_2]]} \qquad \text{E-Let}\ \frac{}{\texttt{(let}\ x\ v\ e\texttt{)} \hookrightarrow_{\text{E}} e[x/v]}$$

Figure 8: Non-trivial evaluation rules. The remainder of the rules for evaluation of expressions are straightforward and can be found in Appendix A.4 on page 43.

- $\overline{D^v}\,\texttt{;(redr}\ T\ x_d\ x_a\ \cdot\ e_b\texttt{)}$, which selects the non-valued accumulator of a reducer applied to a buffer that only contains values.

Now that we have the first non-valued arithmetic expression, we use $\hookrightarrow_{\text{E}}$ to take a step.

The arithmetic expression reduction relation $\hookrightarrow_{\text{E}}$ is defined in Appendix A.4 on page 43. These are the standard rules to carry out addition, subtraction and multiplication. Our division and modulo operations are special because we have a special rule that specify that division and modulo by zero is always zero[2]. Selecting the first and second element from a tuple are also implemented how one would expect. The conditional `(if` e_c e_t e_f`)` construct is replaced by e_t if e_c is a non-zero number otherwise, if it is zero, it is replaced by e_t.

Let bindings are implemented using substitution. E-Let is also shown in figure 8. Once we know the value v to assign to x in `(let` x v e`)`, we replace the `(let ...)` by e with all occurrences of x substituted by v.

5.2. Transformations

Figure 9 shows the evaluation rules for mappers, reducers and shapers. These rules work with the two first items of the work list. The first item will contain the buffers on which to execute the transformation in the second item. There may be more elements in the work list after the first two, these are captured by E_w and do not change.

First, we look at **mappers**. E-CallMapper shows how a mapper is applied to a buffer of values (D^v). If the buffer contains non-value expressions the E-Other rule will evaluate them into values. We have $\texttt{(buf}_{S[k]}\ v_1, \ldots, v_k\texttt{)}\,\texttt{;(mapr}\ r\ T\ x_i\ x_d\ e\texttt{)}$ at the front of our work list. Mappers apply the function described by e to all elements in the buffer. The variables x_i and x_d may be used in e to refer to the index and the value in the input buffer respectively. Our rule replaces every value v_k of the input buffer by $e[x_i, x_d/k, v_k]$. The square brackets denote substitution. For the j-th element, every occurrence of x_i is replaced by j, the index of the value. Each x_d in e is substituted by the value v_j. After the application of the mapper, the `(mapr ...)` construct itself is eliminated, only the output buffer D_o remains. Note that the buffer maintains its size k. Both the input buffer D_i and the output buffer D_o have length k in their subscript.

Reducers are codified in two rules, E-CallRedr and E-CallRedr0. The first rule, combines the accumulator v_0 and the first element of the buffer $v_{i,0}$. This value is computed as $e_b[x_d, x_a/v_{i,0}, v_0]$, after applying the rule, the input buffer loses one element. The `(redr ...)` construct remains in place and holds the new accumulator value. Once the input buffer has a size less than one, E-CallRedr0 extracts the accumulator from the `(redr ...)` construct and replaces the start of the work list by a new buffer with the accumulator value as its only element. If the accumulator expression is not a value, E-Other applies.

Finally, to execute **shapers**, we need four rules. The first rule, E-CallShpr, initiates a shaper. It creates a memory $\mathcal{M}_s$ that assigns the name $x_{v,i}$ to the i-th source buffer $D^v_{s,i}$. The rule also computes the length k of the output buffer. This is done by applying the join function to the type of the input buffers, and the

[2]We chose not to include exceptions to our language to focus on our type system. Although tedious, exceptions can be added to our language as described in Types And Programming Languages [13, Chapter 14].

transformation type T of the shaper[3]. Each element of the output buffer of length k is the body of the shaper. Just as with mappers, each occurrence of the variable in x_i in an element is substituted for the index of that element. Contrary to the previous transformations, the other arguments $x_{v,1}, \ldots, x_{v,m}$ are not substituted. Instead, they are made available through $\mathcal{M}_s$ by a `(shpr* ...)` construct wrapping the constructed buffer.

The `(shpr* ...)` construct is used to reduce `(index` x num`)` expressions. These expressions can only occur in the bodies of shapers (and thus also in `(shpr* ...)`). E-INDEX and E-INDEXOUTBOUNDS use the $\mathcal{M}_s$ saved in the `(shpr* ...)`. E-INDEX replaces `(index` x num`)` with the value at index num in the buffer named x in $\mathcal{M}_s$. If the index is out of bounds, E-INDEXOUTBOUNDS places a zero of the correct shape[4] at the position of the `(index ...)`. By combining the rules to evaluate `(index ...)` with E-OTHER we can transform all the resulting elements to values. Once this happens, E-ENDSHPR applies. This rule removes the `(shpr* ...)` wrapper around the buffer such that the program can continue.

We briefly return to typing now that we have explained the `(shpr* ...)` construct. TT-BUFFERSHPR in figure 4 shows that this construct has the same type as the buffer it holds typed under Γ_s. This $\Gamma_s = \{x \mapsto B_x \mid x \in \mathsf{dom}(\mathcal{M}_s) \ \wedge \ \vdash \mathcal{M}[x] : B_x\}$ is a mapping of the variable names of the shaper to the corresponding buffer types (figure A.13, page 42). The elements of Γ_s are only used by the TE-INDEX rule that types `(index ...)` constructs, other interactions with the environment only use scalars.

5.3. Coordination Plan

The coordination plan links calls to transformations and queries the memory to provide input data. For transformers, the work list always started with a tuple of buffers. Here, we will show where these buffers come from, how we extend the memory and how we invoke transformers. Figure 10 shows the reduction rules for the coordination plan.

There are two ways a buffer can get introduced in the work list. First, it can be written literally. We do not need a rule for this case as literal buffers are values. Second, the buffer is obtained from memory with `(retrive ...)`. In this case E-RETRIVE inspects the memory $\mathcal{M}$ contained in E_w. It selects the right buffers D_i^v and composes them into a tuple.

Our `(letB` x wl_1 wl_L`)` construct allows us to add elements to the store for a limited scope. When it is the first element of the work list, we evaluate it. There are three rules involved in this: E-LETB1, E-LETBBUF and E-LETBEND

As long as wl_1 is not a single value buffer, E-LETB1 applies. This rule extracts wl_1 and wraps it into a new instantiated program construct. Then, it recursively applies $\rightsquigarrow_{\mathrm{P}}$ to it and unwraps the result to get the new value of wl_1. As soon as wl_1 is a tuple of one buffer of values, we evaluate wl_L under an extended memory. E-LETBBUF codifies this. This rule extracts wl_L and wraps into a new instantiated program construct with an extended memory $\mathcal{M}'$. $\mathcal{M}'$ is derived by extending[5] the original $\mathcal{M}$ with a new binding mapping the variable name x of the `(letB ...)` with the value buffer computed by wl_1. We recursively apply $\rightsquigarrow_{\mathrm{P}}$ on the wrapped value and unwrap the result to get the new value of wl_L. Once both wl_1 and wl_L only contain values, there is no need for the `(letB ...)` construct anymore. E-LETBEND removes it, and only keeps the buffers resulting from wl_L.

In our semantics `(letB ...)` constructs serve as a tracker. They can be nested and keep track of the names and buffers that should be added to the store when evaluating the second argument (wl_L). Other programming languages typically use a stack for this purpose. We could also have used a stack, but the type system, the language and the corresponding proof grew hand in hand, and we arrived at this formalization.

[3] For a well-typed program, this will always succeed. The length k will be a concrete integer because all the input buffers were concrete and thus had a fixed integral length.

[4] ($0_{\texttt{int}\times\texttt{float}}$ = `(tuple 0 0.0)`)

[5] Note that we have assumed that all names in the program are unique

E-CallMapr

$$\frac{D_i^v = (\texttt{buf}_{S[k]}\ \ v_0, \ldots, v_{k-1}) \qquad D_o = (\texttt{buf}_{S'[k]}\ \ e[x_i, x_d/0, v_0], \ldots, e[x_i, x_d/k-1, v_{k-1}])}{E_w\,[(D_i^v)\,\fatsemi\,(\texttt{mapr}\ \ r\ \ T\ \ x_i\ \ x_d\ \ e)] \rightsquigarrow_{\mathrm{P}} E_w\,[D_o]}$$

E-CallRedr

$$\frac{\begin{array}{c} D_i^v = (\texttt{buf}_{S[k]}\ \ v_{i,0}, v_{i,1}, \ldots, v_{i,k-1}) \\ k > 0 \qquad D_o^v = (\texttt{buf}_{S[k]}\ \ v_{i,1}, \ldots, v_{i,k-1}) \qquad e_1 = e_b[x_d, x_a/v_{i,0}, v_0] \end{array}}{E_w\,[(D_i^v)\,\fatsemi\,(\texttt{redr}\ \ r\ \ T\ \ x_d\ \ x_a\ \ v_0\ \ e_b)] \rightsquigarrow_{\mathrm{P}} E_w\,[(D_o^v)\,\fatsemi\,(\texttt{redr}\ \ r\ \ T\ \ x_d\ \ x_a\ \ e_1\ \ e_b)]}$$

E-CallRedr0

$$\frac{k \leq 0}{E_w\,[((\texttt{buf}_{S[k]}\ \ ))\,\fatsemi\,(\texttt{redr}\ \ r\ \ T\ \ x_d\ \ x_a\ \ v_0\ \ e_b)] \rightsquigarrow_{\mathrm{P}} E_w\,[(\texttt{buf}_{S'[1]}\ \ v_0)]}$$

E-CallShpr

$$\frac{\emptyset \mid \emptyset \vdash_{\mathrm{P}} \overline{D_s^v} : \overline{B_s^v}|\emptyset \qquad S_o^c[k] = \mathsf{join}\,(\overline{B_s^v}|\emptyset\,, T) \qquad \mathcal{M}_s = (x_{v,1} : D_{s,1}^v) : \ldots : (x_{v,m} : D_{s,m}^v)}{\begin{array}{c} E_w\,[\overline{D_s^v}\,\fatsemi\,(\texttt{shpr}\ \ r\ \ T\ \ x_i\ \ (x_{v,1}\ \ \ldots\ \ x_{v,m})\ \ e)] \\ \rightsquigarrow_{\mathrm{P}} E_w\,[(\texttt{shpr*}\ \ ((\texttt{buf}_{S_o^c[k]}\ \ e[x_i/0], \ldots, e[x_i/(k-1)]))\ \ \mathcal{M}_s)] \end{array}}$$

E-Index

$$\frac{x_1 \mapsto (\texttt{buf}_{S[m]}\ \ v_0 \ldots v_{m-1}) \in \mathcal{M}_s \qquad 0 \leq num < m}{E_w\,[(\texttt{shpr*}\ \ E_B[E[(\texttt{index}\ \ x_1\ \ num)]]\ \ \mathcal{M}_s)] \rightsquigarrow_{\mathrm{P}} E_w\,[(\texttt{shpr*}\ \ E_B[E[v_{num}]]\ \ \mathcal{M}_s)]}$$

E-IndexOutBounds

$$\frac{x_1 \mapsto (\texttt{buf}_{S[m]}\ \ v_0 \ldots v_m) \in \mathcal{M}_s \qquad \neg(0 \leq num \leq m)}{E_w\,[(\texttt{shpr*}\ \ E_B[E[(\texttt{index}\ \ x_1\ \ num)]]\ \ \mathcal{M}_s)] \rightsquigarrow_{\mathrm{P}} E_w\,[(\texttt{shpr*}\ \ E_B[E[0_S]]\ \ \mathcal{M}_s)]}$$

E-EndShpr

$$\frac{}{E_w\,[(\texttt{shpr*}\ \ (D^v)\ \ \mathcal{M}_s)] \rightsquigarrow_{\mathrm{P}} E_w\,[(D^v)]}$$

Figure 9: Evaluation rules for transformations in coordination plans. For the sake of clear presentation we have omitted the premises to define S' in E-CallMapr, E-CallRedr0. The premise to be added to each rule is: $S'[num_{la} \cdot n_l + num_{lb}]|\mathcal{C} = \mathsf{join}\,(\overline{B_i^v}, T)$ with $\overline{B_i^v}$ the concrete type of $\overline{D_i^v}$ (or $(D_1^v, \ldots, D_m^v)$ in the case of E-CallShpr). Note that the notation $v_a, \ldots, v_b$ used in buffers is empty if $b < a$.

$$\textsc{E-Retrive}\quad \frac{E_w = \texttt{(main (prog } \mathcal{A}\ E_{wl}\,[\cdot]\texttt{)}\ \mathcal{M}\texttt{)} \qquad \forall i.(x_i \mapsto D_i^v) \in \mathcal{M}}{E_w\,[\texttt{(retrive } x_1\ \ldots\ x_m\texttt{)}] \rightsquigarrow_{\mathrm{P}} E_w\,[(D_1^v \ldots D_m^v)]}$$

$$\textsc{E-LetBEnd}\quad \frac{}{E_w\,[\texttt{(letB } x\ D_1^v\ \overline{D_2^v}\texttt{)}] \rightsquigarrow_{\mathrm{P}} E_w\,[\overline{D_2^v}]}$$

$$\textsc{E-LetB1}\quad \frac{E_w = \texttt{(main (prog } \mathcal{A}\ E_{wl}\,[\cdot]\texttt{)}\ \mathcal{M}\texttt{)} \qquad \texttt{(main (prog } \mathcal{A}\ wl_1\texttt{)}\ \mathcal{M}\texttt{)} \rightsquigarrow_{\mathrm{P}} \texttt{(main (prog } \mathcal{A}\ wl_2\texttt{)}\ \mathcal{M}\texttt{)}}{E_w\,[\texttt{(letB } x\ wl_1\ wl_3\texttt{)}] \rightsquigarrow_{\mathrm{P}} E_w\,[\texttt{(letB } x\ wl_2\ wl_3\texttt{)}]}$$

$$\textsc{E-LetBBuf}\quad \frac{E_w = \texttt{(main (prog } \mathcal{A}\ E_{wl}\,[\cdot]\texttt{)}\ \mathcal{M}\texttt{)} \qquad \begin{array}{l}\texttt{(main (prog } \mathcal{A}\ wl_2\texttt{)}\ \mathcal{M}'\texttt{)} \\ \rightsquigarrow_{\mathrm{P}} \texttt{(main (prog } \mathcal{A}\ wl_3\texttt{)}\ \mathcal{M}'\texttt{)}\end{array} \qquad \mathcal{M}' = (x \mapsto D^v) : \mathcal{M}}{E_w\,[\texttt{(letB } x\ \texttt{(}D^v\texttt{)}\ wl_2\texttt{)}] \rightsquigarrow_{\mathrm{P}} E_w\,[\texttt{(letB } x\ \texttt{(}D^v\texttt{)}\ wl_3\texttt{)}]}$$

$$\textsc{E-Call}\quad \frac{E_w = \texttt{(main (prog } \mathcal{A}\ E_{wl}\,[\cdot]\texttt{)}\ \mathcal{M}\texttt{)} \qquad \texttt{(abst } r\ \texttt{(}\overline{S} \to T\texttt{)}\ \overline{x_a}\ t\texttt{)} \in \mathcal{A}}{E_w\,[\overline{D^v}\ \texttt{;}\ \texttt{(call } r\ \overline{v}\texttt{)}] \rightsquigarrow_{\mathrm{P}} E_w\,[\overline{D^v}\ \texttt{;}\ t[\overline{x_a}/\overline{v}]]}$$

Figure 10: Evaluation rules for coordination plans

Our final rule, E-Call, invokes an abstraction by name. To do this, it looks it up the definition in $\mathcal{A}$. From the definition it derives the variable names $\overline{x_a}$ and body t. The `(call ...)` construct is replaced by $t[\overline{x_a}/\overline{v}]$ where $\overline{v}$ are the values of the arguments.

6. Soundness

We have made some bold claims about our size-polymorphic type systems. In this section we will provide more evidence for the claims we made. First, we will discuss a lemma that formalizes an intuition for why our type system is correct. Then, we will use this lemma to prove progress and preservation. Finally, we will show that:

For any run configuration $\texttt{(main } p\ \mathcal{M}\texttt{)}$ *it holds that if*

- $\forall B \in \overline{B}.\ \exists S.\ \exists k \in \mathbb{Z}.\ B = S[0 * x + k]$ *and* *(k fixed)*
- $\vdash \texttt{(main } p\ \mathcal{M}\texttt{)} : \overline{B}$

then, there exists a tuple of value buffers $\overline{D^v}$ *of type* $\overline{B}$ *such that*

$$\texttt{(main } p\ \mathcal{M}\texttt{)} \rightsquigarrow_{\mathrm{P}}{}^{*} \texttt{(main (prog } \mathcal{A}\ \overline{D^v}\texttt{)}\ \mathcal{M}\texttt{)}$$

This section contains an abbreviated version of our proofs. The full proofs can be found in Appendix B (starting on page 45).

6.1. Definitions

We will use the term "constructed type" for types that could have been derived for a program. The terminology "inhabited" is sometimes also used to refer to this concept.

Definition 2 (Constructed)**.** *we say a pair* $B|\mathcal{C}$ *is constructed if there exists a* wl, $\mathcal{A}$ *and a* Γ *such that* $\mathcal{A} \mid \Gamma \vdash_{\mathrm{P}} wl : B|\mathcal{C}$

When we type an instantiated program we apply validate to the type and constraints of the program (TP-Validate). These are actually the buffer type and constraints derived for the work list in the program (TP-Prog). A work list is typed by repeatedly using TP-List, which uses join. We define two new

functions $\mathsf{join}^\star$ and $\mathsf{validate}^\star$ to make it more convenient to derive properties about chained application of TP-List.

Definition 3 (Repeated join and validate)**.** *We define* $\mathit{join}^\star(\ldots)$ *and* $\mathit{validate}^\star(\ldots)$ *inductively as follows:*

$$\begin{aligned}\mathit{join}^\star(B|C) &= B|C\\ \mathit{join}^\star(B_1|C_1, T_1, T_2, \ldots, T_n) &= \mathit{join}^\star(\mathit{join}(B_1|C_1, T_1), T_2, \ldots, T_n)\\ \mathit{validate}^\star(\mathcal{M}, B_1|C_1, T_1, \ldots) &= \mathit{validate}(\mathcal{M}, \mathit{join}^\star(B_1|C_1, T_1, \ldots))\end{aligned}$$

We can immediately write the following lemmas:

Lemma 1 (Join preserves containment of free variables)**.** *If* $FV(\overline{B}) \subseteq FV(\mathcal{C})$ *then for any* $T_1, \ldots, T_n$ *(also* $n = 0$*):* $\overline{B^\star}|\mathcal{C}^\star = \mathit{join}^\star(\overline{B}|\mathcal{C}, T_1, \ldots, T_n) \implies FV(\overline{B^\star}) \subseteq FV(\mathcal{C}^\star)$

Proof. By induction on n. See Appendix B.1. □

Lemma 2. *For any constructed pair* $B|\mathcal{C}$*, it holds that* $FV(B) \subseteq FV(\mathcal{C})$

Proof. By well-founded induction on the derivation of $\mathcal{A} \mid \Gamma \vdash_{\mathrm{P}} wl : B|\mathcal{C}$. See Appendix B.2. □

$\mathcal{M}$ is the set of elements in the supplied memory of the form $\overline{x \mapsto D^v}$. We will sometimes write this set as $\mathcal{M}^E$. Each of the D^v in $\mathcal{M}^E$ has a corresponding type derived by TP-Buf. We may now define a related mapping from names to the types of buffers associated with them:

$$\mathcal{M}^B := \left\{ x \mapsto B \;\middle|\; x \in \mathsf{dom}(\mathcal{M}) \wedge \emptyset \mid \emptyset \vdash_{\mathrm{P}} \mathcal{M}[x] : B|\emptyset \right\}$$

In this paper we will omit the superscript when it is clear from context what $\mathcal{M}$ is used.

6.2. Validation Lemma

Informally, we say that a substitution σ is consistent with a store $\mathcal{M}$ and a set of constraints $\mathcal{C}$ if it replaces some free variables in $\mathcal{C}$ by concrete types or numbers derived from the actual shapes and sizes of the (concrete) buffers in $\mathcal{M}$.

Definition 4 (Store-consistent substitution)**.** *A substitution* σ *is said to be consistent w.r.t. a store* $\mathcal{M}$ *and* $\mathcal{C}$ *iff* $\sigma \subseteq \mathsf{unify}(\mathcal{M}\big|_{\mathit{dom}(\mathcal{C})}, \mathcal{C})$

Now we can define the following lemma that captures an important intuition. Note that it works in two directions. Informally, the lemma teaches us that we may replace a shape variable or size by the concrete value it will get when a memory is supplied without disturbing the result of $\mathsf{validate}^\star$ ($\Rightarrow$). Interestingly, we may also apply in it the opposite direction. We may replace concrete shapes and numbers in the constraints $\mathcal{C}_1$ by variables as long as these variables will be filled in with the same value when validated with $\mathcal{M}$ and Ct_1.

Lemma 3 (Validation is preserved under store consistent substitution)**.** *For any substitution* σ *consistent with* $\mathcal{M}$ *and* C_1*, and any* B_1 *such that* $FV(B_1) \subseteq FV(\mathcal{C}_1)$*:*

$$B = \mathit{validate}^\star(\mathcal{M}, B_1|\mathcal{C}_1, T_1, \ldots, T_n) \iff B = \mathit{validate}^\star(\mathcal{M}, \sigma(B_1|\mathcal{C}_1), T_1, \ldots, T_n)$$

Proof. By induction on n, see Appendix B.4 on page 46. □

6.3. Progress

To prove progress of our language we need to show that our program is either a value, or a step can be taken. It suffices to show that we can always execute a step at the front of our work list (captured by E_{wl}). Note that we need the `(prog ...)` construct to do this, as it contains the memory needed to evaluate `calls` and `retrives`. Using E_{wl} simplifies the induction needed to prove progress for `(letB ...)` constructs. The special case of $E_{wl} = \cdot$ gives us preservation for the full language.

Lemma 4 (Progress of work lists)**.** *For any wl such that*

- $\emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl : \overline{B_T}|\mathcal{C}_T$ *and*
- $\exists \overline{B}.\overline{B} = \mathit{validate}(\mathcal{M}^B, \overline{B_T}|\mathcal{C}_T)$

it holds that either

- $\exists wl'.$ `(main (prog` $\mathcal{A}$ $E_{wl}[wl]$`)` $\mathcal{M}^E$`)` $\rightsquigarrow_{\mathrm{P}}$ `(main (prog` $\mathcal{A}$ $E_{wl}[wl']$`)` $\mathcal{M}^E$`)` *for any E_{wl} ("a step can be taken") or*
- $wl = ($`(buf`$_{S_1[k_1]}$ $\overline{v}$`)`$, \ldots,$ `(buf`$_{S_n[k_n]}$ $\overline{v}$`))` *("wl is a value")*

Proof. By induction on the typing derivation, see Appendix B.8 on page 57. □

Lemma 5 (Progress)**.** *For any run configuration* `(main` p $\mathcal{M}$`)` *with* `(main` p $\mathcal{M}$`)` $\vdash B$ *it holds that there is a p' such that* `(main` p $\mathcal{M}$`)` $\rightsquigarrow_{\mathrm{P}}$ `(main` p' $\mathcal{M}$`)` *or p is an value program*

Proof. `(main` p $\mathcal{M}$`)` $\vdash B$ could only have been derived by TP-Main, TP-Validate and TP-Prog as follows:

$$\text{TP-Main}\ \frac{\text{TP-Prog}\ \dfrac{\forall d_i \in \mathcal{A}.\vdash_{\mathrm{A}} d_i : T \qquad \emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl : B_T|\mathcal{C}_T}{\Gamma \mid \emptyset \vdash_{\mathrm{P}} \texttt{(prog}\ \mathcal{A}\ wl\texttt{)} : B_T|\mathcal{C}_T} \qquad \frac{FV(B) \subseteq FV(\mathcal{C}_T) \qquad \mathsf{dom}(\mathcal{C}_T) \subseteq \mathsf{dom}(\mathcal{M}) \qquad \sigma = \mathsf{unify}\left(\mathcal{M}|_{\mathsf{dom}(\mathcal{C}_T)}, \mathcal{C}_T\right)}{\sigma(B_T) = \mathsf{validate}(\mathcal{M}, B_T|\mathcal{C}_T)}\ \text{TP-Validate}}{\vdash \texttt{(main (prog}\ \mathcal{A}\ wl\texttt{)}\ \mathcal{M}\texttt{)} : \sigma(B_T)}$$

We now know that the (canonical) form of p must be `(prog` $\mathcal{A}$ wl`)`. And we have that $\emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl : B_T|\mathcal{C}_T$ for some B_T and $\mathcal{C}_T$ such that $B = \mathsf{unify}(\mathcal{M}, \mathcal{C}_T)(B_T)$. We may apply lemma 4 and obtain that wl is either a value (and p is a value program) or a step of the form below can be taken.

$$\texttt{(main (prog}\ \mathcal{A}\ E_{wl}[wl]\texttt{)}\ \mathcal{M}\texttt{)} \rightsquigarrow_{\mathrm{P}} \texttt{(main (prog}\ \mathcal{A}\ E_{wl}[wl']\texttt{)}\ \mathcal{M}\texttt{)}$$

for any E_{wl} including "$\cdot$". This proves the lemma. □

6.4. Preservation

Lemma 6 (Preservation)**.** *For any run configuration* `(main` p $\mathcal{M}$`)` *it holds that if*

- $B = S[0 * x + n]$ *(output has concrete type, n is fixed)*
- $\vdash$ `(main` p $\mathcal{M}$`)` $: B$
- `(main` p $\mathcal{M}$`)` $\rightsquigarrow_{\mathrm{P}}$ `(main` p' $\mathcal{M}$`)`

then,

- $\vdash$ `(main` p' $\mathcal{M}$`)` $: B$

Proof. The full proof can be found in Appendix B.6 on page 50.

Intuitively, we can use lemma 3 to convince ourselves that preservation holds. As the full program is being executed, the first items of the work list are replaced by concrete values. The reduction rules will work on

the value buffers, and the buffers will be of the right type. This is similar to how nested function application works in other languages.

If there are `(retrive ...)` constructs they must be at the start of a work list. Lemma 3 tells us that the output buffer type B adheres to following rule, for any σ consistent with the store and the constraints.

$$B = \mathsf{validate}^{\star}(\mathcal{M},\, B_1|\mathcal{C}_1\,, T_1, \ldots, T_n) \implies B = \mathsf{validate}^{\star}(\mathcal{M}, \sigma\,(B_1|\mathcal{C}_1\,), T_1, \ldots, T_n)$$

Let us assume that the work list starts with `(retrive` x`)`, which would be typed $X[n_x]|\{x \mapsto X[n_x]\}$. Now take a memory $\mathcal{M}$ that satisfies the left hand side of the implication above. Any store consistent substitution replaces some shape variables in $B_1|\mathcal{C}_1$ by concrete values in accordance with the memory. If $\mathcal{M}[x] = \texttt{int}[3]$, a store consistent unifier would be $\sigma = \langle X/\texttt{int}\rangle\, \langle\lambda(a \cdot n_x + b)/(0 \cdot l + 3)\rangle$. The right-hand side of the implication above now becomes:

$$B = \mathsf{validate}^{\star}(\mathcal{M}, (\,\texttt{int}[3]|\{x \mapsto \texttt{int}[3]\}\,), T_1, \ldots, T_n)$$

If we look at TP-VALIDATE, we see that the constraints are used for two things: (a) validate if the memory adheres to them (b) fill in type variables in the output buffer type based in the unification of the final and the $\mathcal{M}$. Because the program successfully type checked before, we know that both conditions are true. Applying σ to $\mathcal{C}_1$ fills in some of the free variables in $\mathcal{C}_1$ but does not change the result of validate. We simply applied a part if the substitution validate would apply anyway. In fact, validate derives no substitutions from concrete requirements in $\mathcal{C}_1$, so we may drop these. This is not a problem because (a) was already fulfilled for this memory. We arrive at: $B = \mathsf{validate}^{\star}(\mathcal{M}, (\,\texttt{int}[3]|\emptyset\,), T_1, \ldots, T_n)$. The type is preserved. □

6.5. Soundness

Combining the progress lemma with the preservation lemma gives us following theorem.

Theorem 7. *For any run configuration* `(main` p $\mathcal{M}$`)` *it holds that if*

- $\forall B \in \overline{B}.\ \exists S.\ \exists k \in \mathbb{Z}.\ B = S[0 * x + k]$ *and* *(k fixed)*
- $\vdash$ `(main` p $\mathcal{M}$`)` $: \overline{B}$

then, there exists a tuple of value buffers $\overline{D^v}$ *of type* $\overline{B}$ *such that*

$$\texttt{(main}\ p\ \mathcal{M}\texttt{)} \rightsquigarrow_{\mathrm{P}}{}^{*}\ \texttt{(main (prog}\ \mathcal{A}\ \overline{D^v}\texttt{)}\ \mathcal{M}\texttt{)}$$

Put informally, starting a program with a memory that adheres to the constraints derived for the program, will eventually[6] result in `(main (prog` $\mathcal{A}$ $\overline{D_v}$`))`. And it will be the case that the type of D_v will be the type $\overline{B}$ which we got by typing the program before execution.

6.6. Practical Implications

The soundness theorem above indeed ensures that we will get an output of the expected shape and size if we run our program. Nevertheless, it is possible that some buffers in the program get a negative size during execution. This may for example happen if we supply a $\texttt{int}[5]$ buffer to a $A[n] \twoheadrightarrow A[n-9]$ shaper. At first sight this might seem to be a counterexample for soundness. But this is not the case. The result of our example will be a $\texttt{int}[-4]$ buffer which we write as $(\texttt{buf}_{\texttt{int}[-4]}\ )$. This is a valid expression in our language, typed $\texttt{int}[-4]$ by TM-BUFFER in figure A.13, because the ranges $0, \ldots, -4$ and $e_0, \ldots, e_{-4}$ are empty. Both the typing and reduction rules of mappers, shapers and reducers can handle negative buffer sizes. The `(index ...)` construct executed on buffers of negative size will always be resolved by E-INDEXOUTBOUNDS.

[6] All of the constructs in Gaiwan are guaranteed to terminate, so a result will be reached. Proof can be made by induction on the worklists.

We will thus only read zeros from these buffers, this is an appropriate value for reading outside the bounds of a buffer when using techniques like sliding windows.

The implementation of Gaiwan can issue a warning to the users if a buffer's size will become negative during the execution. We can detect this when we compute the concrete sizes of the buffers will be using for execution by filling in the holes in our program. If a negative buffer size crops up, we can then inform the user about where in the program this occurs.

7. Evaluator

Gaiwan's evaluator is built in a modular way and consists of three parts: a static analyzer, an intermediate code generator and an executor. The static analyzer validates the syntax and types of the program. Its result is a set of constraints and a typed program with types that may refer to the variables in the constraints. Our intermediate code generator then transforms the typed program into a series of typed execution steps, which again may refer to shape and size variables in the constraints. The intermediate code is identical for every execution of the program, regardless of the input size or shapes. To speed up evaluation of the program for different data sets, we may store the intermediate code. With a concrete memory that adheres to the constraints, we can ask an executor to compute the result. The executor generates concrete code for a targeted platform, such as OpenCL or CUDA, and executes it. Apart from running the code on a GPU, an executor could also be used to generate an SVG schematic of the program's execution.

7.1. Static Analyzer

To validate a program we first type the abstractions and then validate the coordination plan. Abstractions always have a simple transformation type with no constraints ($\overline{B} \twoheadrightarrow \overline{B}|\emptyset$). They can be independently validated. With this we can type the coordination plan.

The coordination plan is typed in two passes. First, we recursively type the program. Each base part of the plan gets a type and a set of constraints. These are then combined to type any composite step (such as `letB`). Once an entire sequence of steps is typed, the second pass pushes the obtained information back into the types of the constituents. For example, if a merged work item has the transformation type $(C \times \texttt{int})[2n] \twoheadrightarrow C[n]$ obtained from combining $A[m] \twoheadrightarrow A[3m+1]$ and $(B \times \texttt{int})[6k+1] \twoheadrightarrow B[k]$. The constituents are now retyped $(C \times \texttt{int})[2n] \twoheadrightarrow (C \times \texttt{int})[6n+1]$ and $(C \times \texttt{int})[6n+1] \twoheadrightarrow C[n]$. Individual mappers, shapers and reducers also get an updated type. By doing this, we can quickly derive the final types of each piece of the program once the memory is supplied in the execution phase.

7.2. Intermediate Code Generator

We cannot generate concrete executable code without knowing the size and shape of the input. We can, however, already generate intermediate code that describes the steps a GPU should perform to execute the program once these details are known. Steps include allocating buffers, reading input data into buffers, extracting output data from buffers, creating kernels[7] and executing them. A call to a shaper of type $(A[n], B[n]) \twoheadrightarrow A[n]|\emptyset$ could, for example, be implemented as a kernel with three buffer arguments: two input buffers and one output buffer. The list of steps we generate in this phase ensures that the required buffers are allocated and filled before the kernel is called and that the data of the result buffer is extracted afterwards. If `letB` bindings are used, the intermediate code will derive the value of the binding before executing its body. The intermediate code generator queries the selected executor to get hints on what optimizations it may carry out. This way the intermediate code may be instructed not to use certain features that are not present on all platforms (e.g. global barriers). As a consequence, the generated intermediate code only works for the executor it was generated for.

[7] A kernel is a routine that can be invoked on the GPU by OpenCL. It can interact with the buffers given to it as arguments.

and then →	mapper	reducer	shaper
mapper	chain	rewrite access	concertize
reducer	keep	2nd reducer becomes mapper	concertize
shaper	rewrite access	rewrite access	chain

Table 3: Equational reasoning on transformations. This table shows what can be done when one transformation type (left) is followed by an other (top).

7.2.1. Equational Reasoning

If there are multiple work items, there may be multiple kernels. Because there is an overhead to calling and creating kernels, we try to fuse subsequent steps by using equational reasoning[16]. Table 3 summarizes the possible combinations and how we deal with them in a datarace-free way. We highlight some of the combinations below.

Mapper-Mapper. Whenever two mappers are executed after each other they can be combined into a single mapper that executes both tasks. If a first mapper multiplies by 14 and the second mapper multiplies by 3, we may use a mapper that simply multiplies by 42 once. In the case that a mapper returns a tuple, we have a formula for every entry of the tuple. We can then look at the tuple access in the second mapper and do the composition at that point.

Shaper-Mapper, Shaper-Reducer and Shaper-Shaper. Executing any transformation after a shaper can be reformulated by changing the access pattern to access a different value. The fused transformations are of the same kind as the last transformation.

Mapper-Shaper. We do not have a generally applicable optimization for applying a shaper after a mapper. In theory, we could substitute the index in the mapper for the new value that would be computed by the shaper. The resulting transformation is not a real mapper nor a shaper (nor a reducer).

As a shaper may select the same value multiple times, fusing may lead to computing the same value multiple times. Applying static analysis techniques to the body of the shaper could learn us if merging a mapper and a shaper is useful on a case by case basis.

In our current implementation we determine the complexity[8] of the mapper's output. If the complexity is high we concertize the result of the mapper and apply the shaper on the stored value. When the complexity is low me merge the mapper and the shaper.

Associative Reducers. If there are data dependencies between the calls of a reducer, it must be executed serially. Such dependencies may be inherent to the problem. Often however a reducer represents an associative operation, which can be sped up by using the tree-based reduction pattern described in section 2.2.2. This pattern has a lower algorithmic complexity than the classic serial method, but requires more costly operations such as multiple kernel calls or introducing barriers. The performance gains of a lower algorithmic complexity outweigh the cost of these operations for sufficiently large inputs.

Gawain attempts to statically prove that the operation is associative by using a rewrite system. Although this system may have false negatives, it is able to correctly detect common associative operations such as addition of all elements.

7.2.2. Simplification and Common Sub-Expression Lifting

The fusing operations described above may lead to kernels with complex expressions in their body. As stated before, a mapper following a shaper that creates tuples may for example only use a part of the tuple. The

[8]This value is the amount of leafs in the simplified abstract syntax tree of the expression describing the mappers output.

fused kernel may therefore contain sub-expressions like `tuple(a,b)._1`), which can be simplified to just `a`. To keep the kernels as simple as possible, we simplify the expressions after every fusing operation.

We additionally look for common sub-expressions and extract these into a let binding. This reduces the amount of duplicate computations, and further improves performance.

7.3. Executor

The intermediate code works for any memory that adheres to the constraints of the type system. Once we have a concrete memory, we can derive the values of the shape and size variables mentioned in the intermediate code. If we get a memory $\{a \mapsto \texttt{(buf}_{int[4]}\ 1\ 2\ 3\ 4\texttt{)}\}$ for the constraints $\{a \mapsto A[2n]\}$, we may derive that $A = \texttt{int}$ and $n = 2$. These values can then be filled in the intermediate code and execution can start.

Our current implementation has an initial version of an OpenCL executor. This executor translates the expression in the intermediate code into OpenCL kernels. Buffers are read from files, `(retrive a)` will read the values from the file `a.int.gw`. Our executor takes care of setting up the GPU and loading the data and the OpenCL kernels onto the device. Once these are loaded, the kernels are called in the order dictated by the intermediate code.

With knowledge of the concrete types of the program we can carry out more optimizations and determine the optimal way to store data. Our current implementation simply converts the intermediate code into OpenCL without any additional OpenCL specific optimizations.

8. Evaluation

Our implementation of Gaiwan contains the full type system, and a prototype implementation of an OpenCL executor. While the executor is not yet fully optimized for performance, it can execute any well typed Gaiwan program (for valid inputs).

To evaluate our work we will showcase two usage examples and a benchmark. First we show how we can use Gaiwan to analyze GPS traces (section 8.1). Then we take a closer look at the kernel generated for a Gaiwan program that computes a dot product. Finally, we illustrate the scalability of Gaiwan with growing program sizes, and we compare its performance to that of a hand optimized implementation of Bitonic Sort.

8.1. Usage Example: Analyzing GPS data

Gaiwan can be used to analyze time series data such as GPS traces. We show the start of a GPS data set below. Each three consecutive numbers represent a single measurement of the longitude, latitude and altitude at a moment in time. A new measurement is made every minute.

`data=`51.3463379, 4.2859038, 2.98, 51.3463377, 4.2859039, 47.939, 51.3463377, 4.2859039, 47.939, 51.3463376, 4.2859039, 47.969, 51.3463374, 4.285904, 47.989, 51.3463374, 4.285904, 47.989, 51.3463373, 4.285904, 48.009, 51.3463372, 4.2859041, 48.019, 51.3463372, 4.2859041, 48.019, 51.3463371, 4.2859041, 48.029, 51.346337, 4.2859042, 48.039, 51.346337, 4.2859042, 48.039, 51.346337, 4.2859042, 48.039, 51.3463369, 4.2859043, 48.039, 51.3463369, 4.2859043, 48.039, 51.3463369, 4.2859043, 48.039, 51.3463369, 4.2859043, 48.029,...

In this section we will show how we can use Gaiwan to determine the maximum distance traveled in any 3 minute interval. Additionally, we also want to determine the the highest reached altitude.

Let us first look at the simplest task: computing the maximal altitude. To do this we need to extract every third number from the data set and then find the maximal value.

The former sub-task is a job for a shaper, one that will select every third element. Lines 2-3 in listing 7 fulfill this task. These lines define a shaper of type $A[3n] \twoheadrightarrow A[n]$. Conceptually, the body of the shaper carries out `out[i] = data[3*i+2]`, making the output a buffer of altitudes. We give our shaper a name, `getAlt`, by wrapping it in an abstraction (lines 1-3).

```
1  abstraction getAlt(): A[3n] -> A[n]{
2      shaper(i:int, data:A[3n]): A[n]{
3          data[3*i+2] } }
4
5  abstraction findMax(): float[n] -> float[n] {
6      reducer(acc:float  = -inf, distance:float) : float{
7          if(distance > acc){ distance }
8          else { acc } } }
9
10 abstraction analyze(): float[3n] -> float[n]{
11     shaper(i:int, data:A[3n]): (A, A)[n]{
12         (data[i*3+0],data[i*3+1]) } ⨟
13     shaper(i:int, data:B[n+1]): B[n] {
14         (data[i], data[i+1]) } ⨟
15     mapper(i:int, positions: ((float,float),(float,float))) : float {
16       // Haversine formula: two coordinates -> sphere distance
17       // implementaion in listing 11
18     } ⨟
19     shaper(i:int, distances:C[n+2]):(C×C×C)[n] {
20         (distances[i], distances[i+1], distances[i+2]) } ⨟
21     mapper(i:int, distances:(float×float×float)) : float ⨟ {
22         distances._1 + disances._2 + distances._3 }
23
24 (letB maxDist = ((retrive data) ⨟ (call analyze) ⨟ (call findMax)) in
25   (letB maxAlt = ((retrive data) ⨟ (call getAlt) ⨟ (call findMax)) in
26     (retrive maxDist maxAlt)))
```

Listing 7: A program computing the distance

Now, we still need to find the maximum value of our altitude buffer. Lines 6-8 in listing 7 define a reducer of type `float`$[n] \multimap$ `float`$[1]$ that implements the required max transformation. The accumulator, `acc` is initialized with 0. The body specifies that the accumulator should be updated with any larger value in the input buffer. Eventually, the accumulator will hold the maximal altitude. The result of a reducer is a list of length one containing the last accumulator value. We name this reducer `findMax` by wrapping it in an abstraction (lines 5-8).

Given `getAlt` and `findMax` we can find the maximal altitude. To do this we use the following coordination plan: `(retrive data) ⨟(call getAlt) ⨟(call findMax)`. First we access our GPS-positions. With `(retrive data)` we retrieve the coordinates. The output of that action is then provided to `getAlt` by using `call`. Finally, the output of `getAlt` becomes the input of `findMax` by using a second `call`.

For our second part, we determine the maximal distance traveled over 3 minutes. To do this we sum the distances between every three subsequent coordinates. This is done by a combination of multiple mappers and shapers. The abstraction `analyze` groups these together. There are five steps. First, a shaper (lines 11-12) extracts the longitude and latitude, similar to how `getAlt` worked. Then, a second shaper (lines 13-14) takes every two subsequent coordinates (with overlap). The type of this shaper is $B[n+1] \multimap B[n]$. If we have a list of 10 coordinates $(n+1)$ we will get 9 pairs of coordinates (n). The length reduces by one. Third, we use a mapper that computes the sphere distance between two points using the Haversine formula. Fourth, we combine every three subsequent distances with overlap in a tuple. Now we have an overlap of two, so the list shortens by two. The type of the fourth step is thus: $C[n+2] \multimap (C \times C \times C)[n]$. Finally, we compute the sum of these three distances with a mapper.

With this abstraction, and the `findMax` from before, we can find our result with: `(retrive data)⨟(call analyze)⨟(call findMax)`.

To get both results out of the GPU, we use `letB` to combine the results. On line 24 of listing 7 we assign the name `maxDist` to the list of one element that contains the maximal distance. The result of the altitude coordination plan is stored in `maxAlt` on line 25. We use `retrive` to return both values by writing `(retrive maxDist maxAlt)`.

8.2. Usage Example: Dot Product

The hyper parallel architecture of a GPU lends itself well to execution of mathematical operations such the dot product. A dot product computes the sum of the pair-wise multiplication of two vectors of equal length.

```
1  abstraction dot_product() {
2          shaper join(i,a:C[n],b:C[n]) : tuple(C,C)[n] {
3                  tuple(a[i],b[i])
4          } ;
5          mapper mul(i, a:tuple(int,int)) : int {
6                  a[[0]]*a[[1]]
7          } ;
8          reducer sum(i,acc: int , d : int) : int (0){
9                  d + acc
10         }
11 }
12 (retrive a b)  ; dot_product()
```

Listing 8: A program computing the dot product of two vectors a and b

```
1    void kernel kernel0(global int int_array0[LEN_123_1_0],
2                        global int int_array1[LEN_123_1_0],
3                        global uint* intermediateLEN,
4                        global int* intermediate){
5      int int_i = 2*get_global_id(0);
6      int int_acc = 0;
7      int_acc = (((int_array0)[(int_i)])*((int_array1)[(int_i)]))+(int_acc);
8
9      int_i++;
10     if(int_i < LEN_123_1_0){
11       int_acc = (((int_array0)[(int_i)])*((int_array1)[(int_i)]))+(int_acc);
12
13     }
14     intermediate[get_global_id(0)] = int_acc;
15
16   };
17   void kernel kernel1(global int int_array2[LEN_123_0_1],
18                       global uint* stepsizePtr,
19                       global uint* intermediateLEN,
20                       global int* intermediate){
21     int stepsize = *stepsizePtr;
22     int int_i = (2*get_global_id(0))*stepsize;
23     if(int_i + stepsize < *intermediateLEN){
24       int int_v1 = intermediate[int_i];
25       int int_v2 = intermediate[int_i+stepsize];
26       int int_acc; int_acc = (int_v2)+(int_v1);
27       intermediate[int_i] = int_acc;
28
29       if(int_i==0){
30         int_array2[int_i] = int_acc;
31       }
32     }
33   };
```

Listing 9: The generated kernels for a Gaiwan program computing the dot product. With manually added white space to aid understanding.

The Gaiwan implementation of this operation features all our transformations: a mapper, a sharper and a reducer.

In Gaiwan this mathematical operation is implemented as shown in listing 8. At the bottom (line 13) we first fetch two input buffers named `a` and `b` and pass them on to the abstraction `dot_product` defined on lines 1-11. This abstraction consists of three steps. First, we use a shaper to combine the i-th values of the input vectors into a pair (lines 2-3). The type of the shaper guarantees that both input buffers are of the same size and that the output is buffer of pairs that is as long as the input. Second, each pair is processed by a mapper that multiplies the pair's values (lines 5-7). The resulting buffer has the same length as the input buffer because a mapper does not impact buffer lengths. Finally, a reducer (lines 8-10) sums the products yielding the final dot product. It combines all elements of the input buffer into one value. As the combination that is used here is assosiative (a simple addition), Gaiwan will use with the tree-like reduction pattern (section 2.2.2).

The OpenCL kernels Gaiwan generates for this example are shown in listing 9. There are two kernels, which we explain in the rest of this section. Although the generated kernels are not yet fully optimized, we will

show that thanks to equational reasoning many intermediate steps are prevented. To keep the explanation brief we assume the reader has some familiarity with openCL or GPU programming. Readers unfamiliar with this may want to jump ahead to section 8.3.

Kernel functions are executed in parallel as many times as output elements are needed[9]. Each parallel execution has a numeric identifier it can obtain by calling the OpenCL function `get_global_id(0)`. This identifier allows each parallel execution to determine what part of a task is must carry out.

The first kernel is defined on lines 1 to 16, it computes the multiplication of the i-th elements of both input buffers (`int_array0` and `int_array1`) and sums the result as prescribed by the reducer. Rather than simply computing the multiplication and storing the result in a buffer of the same lenght, the two multiplication results are combined, and the result is stored in a buffer named `intermediate` that is just half the size of the original buffer. In effect, the kernel does the shapper, mapper and reducer all at the same time thanks to equational reasoning (see section 7.2.1). This is most clearly visible on lines 7 and 11 highlight. These lines both have the form `d + acc` (the body of the original reducer). However, instead of `d` we have a piece of code of the form `a[[0]]*a[[1]]` (the body of the mapper). The tuple values of the mapper have been translated into direct accesses into the buffer as dictated by our shaper. On line 14 `int_acc` contains the result of applying the mapper, shaper and reducer to the elements at index $2i$ and $2i + 1$ of the input buffers. This result is stored in the i-th element of the `intermediate` buffer.

The second kernel (lines 17-33) combines the accumulator results stored at index $2 \cdot i \cdot s$ and $2 \cdot (i + 1) \cdot s$ and places the result at position $2 \cdot i \cdot s$. By doing this repeatedly with s assigned to increasing powers of two (1,2,4,8,16,...) we will have stored the final reduction value at position 0 of the intermediate vector. This value is then copied to the output buffer (line 30). Combining two values is done with the body of the reducer as shown on line 26. The rest of the function prevents that we read values out of bounds.

8.3. Performance Evaluation: Bitonic Sort

In this section we will show that our executor can handle larger programs. To this end, we will choose an artificial Gaiwan program that we can easily make more complex in terms of program size: Bitonic Sort. This parallel sorting algorithm requires $\mathcal{O}(n^2)$ parallel steps to sort 2^n elements. Gaiwan was designed for programs where the amount of transformations does not change with varying input sizes. Although outside our intended use cases, Bitonic Sort does enable us to demonstrate the performance impact of executing larger programs with more transformation. An additional benefit of using Bitonic Sort is that OpenCL implementations of it are readily available in hand optimized form.

Bitonic Sorting repeatedly combines bitonic sequences to arrive at a monotonically increasing sequence. A bitonic sequence is a sequence of values that first increases monotonically and then decreases monotonically[10]. To sort 2^n values, the process works in n stages. Figure 11 shows a schematic of the Bitonic Sort principle for 16 values (4 stages). Each element of the input is assigned to a horizontal line on the left side of the image. The first stage makes bitonic sequences of length four, two increasing values, two decreasing values. Dark blue regions in the image result in increasing values, light green in decreasing values. Every next stage transforms a bitonic sequences into a monotonic sequence in such a way that every two subsequent results are bitonic. Transforming a bitonic sequence is done in i steps in the i-th stage (red blocks in the image containing arrows). Arrows between two lines swap the values such that the arrow points to the highest value. The first of these steps combines 2^i values (2^{i-1} arrows), the second 2^{i-1}, and so on. The last stage transforms a bitonic sequence of size 2^n into a monotonically increasing sequence of size 2^n, the result.

Listing 10 shows a Gaiwan implementation of Bitonic Sort for 2^{30} values. The coordination plan on the bottom (lines 28-33) reflects the steps we have described above. First we read the input buffer on line 28. Then, we execute two nested loops[11], one for the stages and one for the steps in the stages. The individual

[9] Limited by number of parallel entities available on the GPU

[10] For completeness, a bitonic sequence may also be a circular shift of such a sequence

[11] Loops are syntactic sugar for repeating steps in the coordination plan, Appendix D.1 described this construct formally

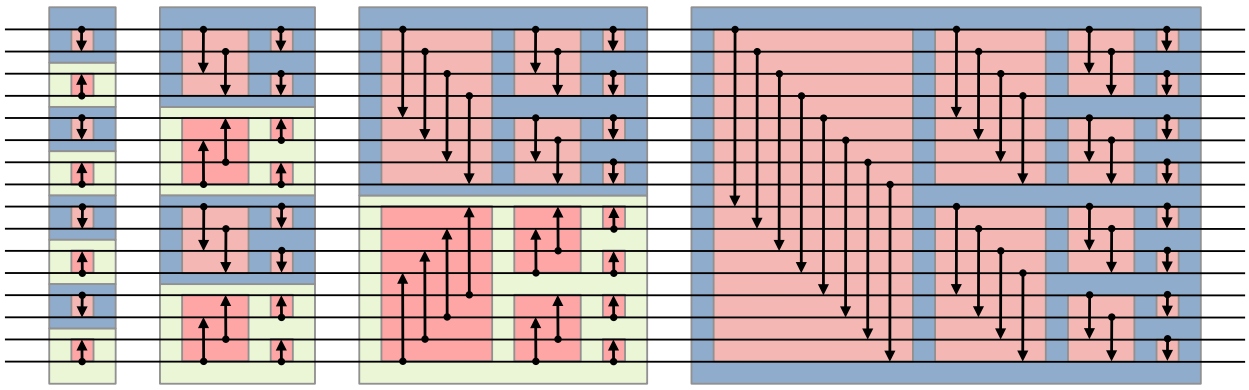


Figure 11: A schematic of a Bitonic Sorter for 16 values. Each element of the input is assigned to a horizontal line on the left side of the image. Arrows between to lines swap the values such that the arrow points to the highest value. The elements of the buffer 1 be sorted on the right of the image. Image adapted from Wikimedia Commons.

steps are implemented in the `bitonic_step` abstraction. It takes two arguments, the stage number and the number of arrows per block (red boxes in figure 11). This abstraction first transforms the data into a list of arrows (tuples) with a shaper. Each tuple of two elements is then sorted in the mapper on lines 9-15. Depending on the index of the arrow, the values are sorted increasingly or decreasingly. Now that our arrows are sorted, we need to decompose them. Lines 16 to 25 define a shaper that does this. For every output index `i`, the corresponding arrow is determined, and the correct part of it is extracted.

Note that the program must be altered to process different buffer sizes, as the number of stages changes for different input sizes. The type of the `bitonic_step` operation is $\texttt{int}[2n] \multimap \texttt{int}[2n]$, Gaiwan enforces that the input buffer has an even size. This is indeed all that is needed for this abstraction to work, `bitonic_step` also works on an $\texttt{int}[10]$ buffer even though 10 is not a power of two. The semantics that the size of the input must be a power of two cannot (yet) be expressed in Gaiwan. Here, we use Bitonic Sort to inspect Gaiwan's behavior when executing large programs. In future work we will also allow for inspect buffer lengths in the coordination plan such that the program does not need to be altered.

An implementation of Bitonic Sort in OpenCL can be found on the website of Intel[12]. It is not contained in this document due to copyright restrictions. The handwritten OpenCL kernel code is 46 lines long after removing all comments and empty lines[13]. The C++ code loading the kernel and sending the data to the GPU is another 40 lines[14]. Both the C++ and OpenCL code are complex and feature things like bit masks, bit operations, buffer alignment computations, memory pointer types, computation contexts and queues. These complexities are not inherent to the bitonic sort problem, they exist only to exploit certain hardware characteristics of the GPU. The actual logic of the bitonic sort is even split over two files, the C++ file and the OpenCL kernel file. By comparison, our Gaiwan program is easier to understand, is contained in one file, and can be explained in a paragraph, as we did above.

We use Gaiwan generated OpenCL program and Intel's handwritten OpenCL program to sort buffers of 32 bit integers with lengths up to 2^{30}. These programs run on an OpenCL C version 1.2 capable GPU, the NVIDIA A100. The log-log graph in Figure 12 shows the execution times of both implementations as solid lines. The measured times include the time taken to upload the data buffer and OpenCL program to the GPU and to run the kernels. We see that the OpenCL code generated by Gaiwan is much slower than the handwritten OpenCL code of Intel. We see that this overhead decreases to 217 times for buffers

[12] https://www.intel.com/content/dam/develop/public/us/en/downloads/intel_ocl_bitonic_sort.zip
[13] Lines only including brackets and white space are also not counted. In total 64 lines were not counted.
[14] Excluding empty lines, lines that only contain brackets, error handling and argument parsing code. 289 lines were not counted.

```
abstraction bitonic_step(stage:int , arrPerBlock:int): int[2m] -> int[2m]  {
    shaper(i:int,d:C[2*n]) : tuple(C,C)[n] { -- split in tuples repesening arrows
        let blockid = i/arrPerBlock in
        let blockstart = blockid * arrPerBlock * 2 in
        let blockoffset = i % arrPerBlock in
        let pos = blockstart + blockoffset in
        tuple(d[pos],d[pos+arrPerBlock])
    } ;
    mapper(i:int, a:(int × int)) : (int × int) { -- swap if needed
        if((i%(2^(stage+1))) < (2^stage)){ -- upper half (increasing sort)
            if(a[[0]] < a[[1]]) {a} else {tuple(a._2,a._1)}
        } else {                           -- lower half (decreasing sort)
            if(a[[0]] < a[[1]]) {tuple(a._2,a._1)} else {a}
        }
    } ;
    shaper(i:int,d:(B × B)[n]) : B[2n] { -- take arrows apart
        let arrowBlock = i/(2*arrPerBlock) in
        let arrowBlockStart = arrowBlock * arrPerBlock in
        let currentArrow = arrowBlockStart + (i % arrPerBlock) in
        if(arrowBlockStart*2+arrPerBlock < i+1){
            d[currentArrow]._1
        }else{
            d[currentArrow]._2
        }
    }
}

(retrive b) ;
for stage in 0..(30-1) {
    for step in 0..stage {
        (call  bitonic_step  stage  (2^(stage - step)))
    }
}
```

Listing 10: An implementation of Bitonic Sort in Gaiwan

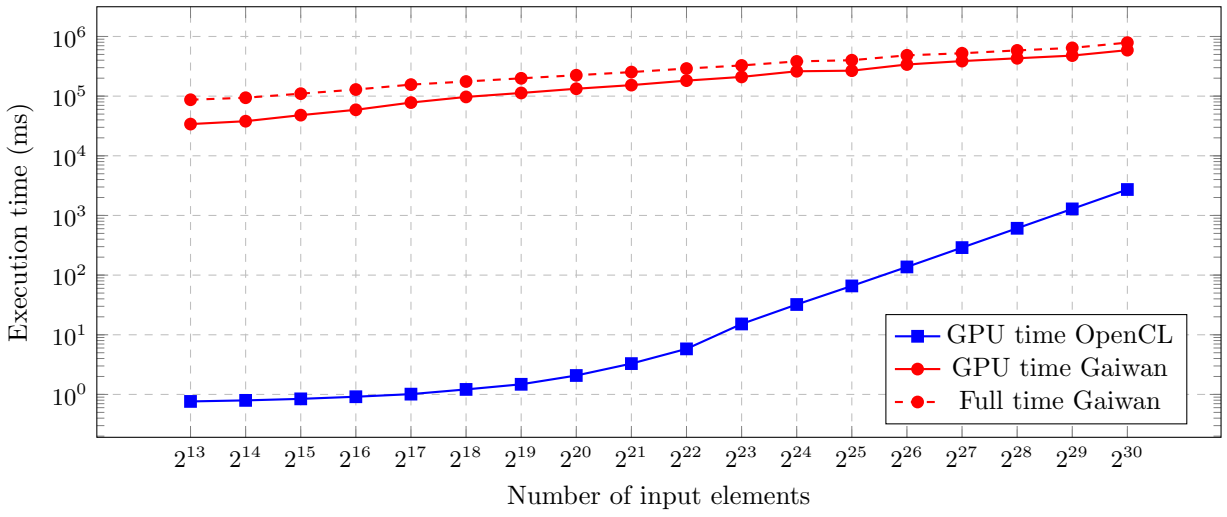


Figure 12: Log-log plot of the execution time of the OpenCL actions of Gaiwan (solid,circle marks) and the hand written Intel OpenCL code (solid, square marks). This measurements includes sending the data buffer and the OpenCL program to the GPU The dashed line represents the full execution time of Gaiwan, which additionally includes the time to type check and optimize the code.

of size 2^{30}. It must be noted that at the time of writing, the loops in the Gaiwan program are unrolled. As a consequence, the size of the program and the number of kernels grows quadraticly with the logarithm of the input size. This has a significant impact on the performance. The GPU needs to process many different kernels, instead of only one. This explains the discrepancy in speed. In future work we plan to address this by allowing kernels that can be parameterized.

The full execution time of the Gaiwan program including type checking, generating an internal representation of the program, optimizing that representation and converting it to OpenCL is shown as a dashed line in Figure 12. As the input size grows, the program grows, and the time taken to optimize it increases as well. The full results of the benchmarks can be found in Appendix E on page 61.

9. Related Work

The work we presented here is part of two domains: on the one hand it exists in the realm of GPU programming and on the other hand our type system is related to dependent typing.

The most important feature of Gaiwan is its size polymorphic type system. It allows users to define a function that describes the change in length of a buffer when a transformation is applied. We can get similar functionality by using dependent type systems such as the one featured in Idris[17]. In these type systems we may define a datatype $Buffer\ a\ b\ S$ to represent a Gaiwan buffer type $S[an + b]$. Transformations may then be represented as a function $Buffer\ a_1\ b_1 \to Buffer\ a_2\ b_2$. We may combine such functions by using a function that converts the types. Our novel size-polymorphic type system is thus likely implementable in Idris. A sketch of such an implementation can be found in Appendix F on page page 62. Gaiwan could have been implemented as an Idris library, we chose to implement Gaiwan in Haskell. Without a Gaiwan-like library the programmer is required to fulfill the proof obligations manually. This means that when they create a function of type $Buffer\ a_1\ b_1 \to Buffer\ a_2\ b_2$ themselves, they must also write a proof for the compiler that their implementation indeed exhibits the declared effect. We chose to implement Gaiwan in Haskell.

Hughes et al. [18] introduced "sized types" to encode the effects of streams in reactive systems with the end goal of proving termination. Their type system makes users express the effect that their operations have on the size of the output as a function. The type checker then validates that these types are correct. Annotating more complex functions with "size types " can be difficult if these combine multiple functions that all have their own size types. As opposed to this, Gaiwan only requires type annotations at the lowest level. At this level, the types have a meaning chosen by the developer directly (e.g. how many output elements should be created per input element of a shaper). The effects of larger parts of the program are automatically inferred by Gaiwan. Additionally, all operations in our system are guaranteed to terminate.

Accelerate[4] is a programming language embedded in Haskell that aims to speed up array computations by using the GPU. Similar to our work, they allow users to declare the shape of the data they are working on. They even allow users to specify that they are working on an n-dimensional matrix that has different lengths along every axis. As with Gaiwan, multiple buffers can be combined with various operators. Accelerate will automatically derive the size of the output at Runtime. Gaiwan derives constraints on the input buffers and gives a formula describing the length of the output buffer during typing. Additionally, Gaiwan informs the user about the size of the input that is required for the program to work. This reduces the chance of unexpected behavior that may occur when buffers of unequal size are `zip`ped, for example.

StreamIt (Software Pipelined Execution of Stream Programs on GPUs) is a programming model that allows users to specify data transformations in terms of so-called filters[6]. Each filter can `pop()` one or more elements from the input, compute a value and `push()` that value to the output. There are also provisions to have a `peek()` at the elements that come next without needing to pop them. In StreamIt, the developer must set a push and pop data rate: the amount of elements that are `pushed` and `popped`ed (or `peek`ed) in each iteration. Multiple filters can be composed in a pipeline, split-join or feedback loop. These composites can be nested but every construct has at most one input and one output. Udupa et al.[7] adapted the StreamIt

compiler to work for GPUs. They use the specified data rate to compute the ideal size for buffers and to cleverly coalesce memory access. Gaiwan's transformations are similar to StreamIt's filters. While StreamIt's filters can only access the next element in the data, Gaiwan's shapers provides access to all elements in the buffer. The data rates given on each filter correspond to the transformation type $A[r_{\texttt{pop}}n] \multimap B[r_{\texttt{push}}n]$ in Gawain, if there are no peeks. Gaiwan gives users more fine-grained control over which data elements they want to inspect and allows any affine function to be used as size. The type system of Gaiwan also informs the developer of the size of the input the program expects. Additionally, Gaiwan has provisions for working with multiple named buffers while StreamIt only works with one data stream.

Halide[10] is a programming language for performing high performance image processing on GPUs[10]. Their central insight is that there must be a separation of concerns between *what* is computed and the *ordering* of these computations. The former concern is expressed in a simple language for defining the algorithm. The latter part is expressed in a language that allows the specification of access patterns (like row-major vs column-major and parallel vs vectorized). Experts can use Halide to iteratively look for the best ordering by altering only that part and leaving the *what* intact. In Gaiwan we have made the same separation of concerns by only providing means to express *what* is computed. The ordering is derived automatically using various heuristics. By separating out shapers as a primitive, we have a good view on the access patterns to derive a good *ordering*. The ordering is thus defined in the implementation of Gaiwan itself. We have made the choice to disallow specifying the order because we Gaiwan is aimed at non-expert GPU users. Gaiwan's main contribution is its advanced size-polymorphic type system, Halide features a more standard type system.

The LIFT-project[9, 19] aims to create "performance-portable" code for heterogeneous programming environments, such as the ones with GPUs. They noticed that GPU code that is optimal for one device may have bad performance on another device. The LIFT language is a high level functional programming language with functions like *map* and *reduce*. These allow users to specify the operations they want to execute on their input. A rewrite system is the used to transform the program to many possible implementations in LIFT's high-level intermediate representation (IR). In turn each IR program is rewritten to all possible OpenCL implementations. Then a heuristic process selects the best of the implementations. Similar to Gaiwan, the LIFT IR can derive the length of buffers that are *divided* into a number of parts, they even allow buffer length defined as an expression in multiple variables. Unfortunately, LIFT cannot inform the developer about the broadest acceptable range of input sizes , like Gaiwan does, it can only accept or reject specific sizes [19].

Futhark[20] is a functional data-parallel array language for OpenCL. It aims to seek a middle ground between functional and imperative languages. The language supports writing the size of buffers in signatures of functions that operate on arrays. The size written in the return type will be dynamically checked at runtime. Unfortunately, Futhark only supports simple names or constants as array sizes, they do not support size expressions such as $2n + 1$.

Chapel[8] aims to generalize parallel programming to make it portable and to easily scalability while maintaining simple code. The language has various convenient constructs to express both task and data parallelism. Chapel 1.25 has preliminary support for GPUs of the NVIDIA brand with CUDA. The language has a partitioned global address space, allowing users and the language itself to decide where to store data. This is especially useful to experts in the context of GPU programming as GPUs have many different memories that work at very different speeds and are accessible to different processing units. At the time of writing, users do not yet have full control of the exact place where data will reside in a GPU with Chapel, but they can specify that certain buffers should exist on the GPU. Gaiwan is aimed at non-expert GPU programmers, therefore we abstract over the different memory layouts and the allocation of buffers required to store intermediate results.

NVIDIA, a company that designs and sells GPU hardware, has various projects to help developers get the most our of their hardware. Their Trust[21] library provides various functions for carrying out common GPU tasks efficiently with C++ CUDA. They also have the NOVA[22] language and compiler. Like the

LIFT project, NOVA provides various high-level functional programming constructs such as *map* and *reduce*. These constructs can then be compiled to run efficiently on NVidia GPUs. Unfortunately, both frameworks require knowledge of the input sizes beforehand.

The MapReduce[11] programming model also features "mapper" and "reducers". The main difference with Gaiwan is that here, they may produce any number of outputs depending on the input values. This is inconvenient for GPUs as they do not feature dynamically sized arrays. He et al. overcomes this issue with a two-pass system that brings the MapReduce model to GPUs [23]. Their first pass executes a mapper or reducer and stores the required output size rather than the actual output. Then they allocate buffers of the computed sizes and start the second pass which runs the operation anew and stores the output. Gaiwan offers a more structured type system that allows programmer to specify beforehand the number of output elements per input element. This allows us to execute our operations in one pass.

GPUs are often used for machine learning, especially for deep neural networks [24]. As the formulae used in neural nets are typically simple, and allow heavy optimization by using polyhedral compilation[25, 26]. Polyhedral compilation restructures loop nests for optimal performance. One example of the application of this technique can be found in the work on Tensor Comprehensions by Vasilache et al.[25]. Gaiwan may also use polyhedral compilation in the future to optimize the generated code once we know the size of the input.

10. Future Work

The design of Gaiwan forces users to separate data access from computation. This helps users understand what data is needed for each step by explicitly writing it down. This separation is also useful to language implementers because they no longer needs to extract access patterns to find performance optimization possibilities. In future iterations of Gaiwan we will shift our focus from the formalization of the type system to efficient implementation. We also aim to generate parameterize kernels, which will improve the speed of loops in the coordination plan by reducing the total amount of kernels.

The size-polymorphic type system of Gaiwan paves the way for research toward GPU resource tracking[27]. We could use this to find the optimal size to split data at and to determine if we need to run the program multiple times to not exceed the memory limit of the current card.

Visualization of programs can greatly aid the understanding of the execution and help thwart bugs [28]. The modular design of our evaluator allows generating images that represent the execution. With this, users could visualize the execution on a specific buffer to find errors in their programs.

Gaiwan explores how to create transformations that are size polymorphic. The coordination plan uses this polymorphism by chaining operations. Sometimes, however, it might also be useful for the coordination plan to directly access the length of a buffer to select what transformations to execute. Further research is needed to encode this non-polymorphic use buffer sizes in the coordination plan.

11. Conclusion

In this paper we presented the size-polymorphic type system of Gaiwan, a programming language aimed at non-expert GPGPU programmers. Existing GPU languages force developers to divide their data into blocks for processing. These sizes are often unrelated to the problem at hand. Unfortunately, these sizes do have a significant impact on the performance of a GPU program [3].

With Gaiwan, programmers only need to specify buffer sizes relevant to their problem. They can declare the effects of a transformation on the sizes of buffers by using **affine functions** in one variable. This gives them the flexibility of using the same program for analyzing both a hundred data points and millions of data points. The type system ensures that the size of the supplied input is in the image of the affine function set as expected input size. Additionally, even without providing input data, our type system provides a set

of **constraints** on the input. When multiple input buffers are combined in the program, the constraints reflect this. For example, two input buffers that must have related lengths will share the variable in their affine size functions The developer can rest assured that any input that adheres to the constraints will work. Even more, without running the program they can predict the output size.

The parametricity of the types is not limited to sizes. Shape variables can be used to abstract over the shape of the elements in a buffer as well. This further facilitates reuse of created abstractions. Again, even without knowing what the input will be, the type system can prescribe constraints for the shape of elements in the input buffers.

Gaiwan features three core building blocks for specifying transformations, all with data race free semantics. **Mappers** apply the same function to every element in a buffer. They preserve the length of the buffer. By design, mappers can only access one element of the input to compute an element of the output, reducing data dependencies and enabling more parallelism. **Reducers** fold all values of a buffer into an accumulator with an associative operation. The result of these operations is a buffer of length one containing the final accumulator value. Reducers can be made parallel by executing them in a tree structure. Finally, **shapers** reshape one or more buffers to create a new output buffer without inspecting the values of these buffers. These are the only operations that can change buffer lengths. Length changes are entirely driven by the type annotations on the shaper. By prohibiting value inspection, shapers have predictable access patterns that can be exploited to improve performance.

Transformations are enacted on the input in the order dictated by a coordination plan. When multiple transformations are applied (`call`) to buffers, the output of the previous transformation becomes the input to the next. This plan specifies which named buffers to read from the environment (`retrive`). New named buffers can also be introduced in the scope of a `letB` binding.

As previously stated, the type system validates the coordination plan and the definitions of the transformations. The output is a list of constraints and an output buffer type. As long as no input memory is given, the output buffer type may contain free variables referring to free variables in the constraints. Once a concrete memory is given, the variables in the constraints can be filled in. If these variables are also used in the output buffer type, we can substitute them to obtain the concrete result buffer type of program. We have proved that our main contribution, the size-polymorphic type system and the accompanying unification system are sound.

To evaluate our work we show a two usage examples, and we perform a small benchmark. Gaiwan is not yet optimized for performance, it is slower than a hand optimized implementation of Bitonic Sort, however its code is much more simple and does not contain complicated constructs (bitmasks, buffer alignment, memory types, ...) which are device dependent. The language constructs in Gaiwan minimize data dependencies and facilitate easy determination of accessed values. In future work, we aim to leverage this information to optimize Gaiwan's performance.

Acknowledgments

Robbert Gurdeep Singh received funding from the Special Research Fund (BOF) of Ghent University under grant number BOF18/DOC/327. We were gracefully allowed to use the A100 of IMEC vzw to run our experiments. We would like to thank Toon Bayens for helping us efficiently solve the constraints on buffer sizes.

References

[1] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, D. Schaa, Heterogeneous Computing with OpenCL, 1st Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[2] J. Cheng, M. Grossman, T. McKercher, Professional CUDA C Programming, John Wiley & Sons, 2014, google-Books-ID: q3DvBQAAQBAJ.

[3] R. Alur, J. Devietti, N. Singhania, Block-Size Independence for GPU Programs, in: A. Podelski (Ed.), Static Analysis, Vol. 11002, Springer International Publishing, Cham, 2018, pp. 107–126, series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-319-99725-4_9.
URL http://link.springer.com/10.1007/978-3-319-99725-4_9

[4] R. Clifton-Everest, T. L. McDonell, M. M. T. Chakravarty, G. Keller, Streaming irregular arrays, in: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, ACM, Oxford UK, 2017, pp. 174–185. doi:10.1145/3122955.3122971.
URL https://dl.acm.org/doi/10.1145/3122955.3122971

[5] F. M. Madsen, R. Clifton-Everest, M. M. T. Chakravarty, G. Keller, Functional Array Streams, in: FHPC '15: The 4th ACM SIGPLAN Workshop on Functional High-Performance Computing, ACM, 2015, pp. 23–34.

[6] W. Thies, M. Karczmarek, S. Amarasinghe, StreamIt: A Language for Streaming Applications, in: G. Goos, J. Hartmanis, J. van Leeuwen, R. N. Horspool (Eds.), Compiler Construction, Vol. 2304, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 179–196, series Title: Lecture Notes in Computer Science. doi:10.1007/3-540-45937-5_14.
URL http://link.springer.com/10.1007/3-540-45937-5_14

[7] A. Udupa, R. Govindarajan, M. J. Thazhuthaveetil, Software Pipelined Execution of Stream Programs on GPUs, in: 2009 International Symposium on Code Generation and Optimization, IEEE, Seattle, WA, USA, 2009, pp. 200–209. doi:10.1109/CGO.2009.20.
URL http://ieeexplore.ieee.org/document/4907664/

[8] T. Carneiro, N. Melab, A. Hayashi, V. Sarkar, Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators, in: HPCS 2020 - The 18th International Conference on High Performance Computing & Simulation, Proceedings of HPCS 2020 - The 18th International Conference on High Performance Computing & Simulation, Barcelona / Virtual, Spain, 2021, p. 9.
URL https://hal.archives-ouvertes.fr/hal-03149394/file/final_hpcs2020.pdf

[9] M. Kristien, B. Bodin, M. Steuwer, C. Dubach, High-level synthesis of functional patterns with Lift, in: Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ACM, Phoenix AZ USA, 2019, pp. 35–45. doi:10.1145/3315454.3329957.
URL https://dl.acm.org/doi/10.1145/3315454.3329957

[10] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, F. Durand, Halide: decoupling algorithms from schedules for high-performance image processing, Communications of the ACM 61 (1) (2017) 106–115. doi:10.1145/3150211.
URL https://doi.org/10.1145/3150211

[11] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.
URL https://doi.org/10.1145/1327452.1327492

[12] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, Communications of the ACM 3 (4) (1960) 184–195. doi:10.1145/367177.367199.
URL https://doi.org/10.1145/367177.367199

[13] B. C. Pierce, Types and programming languages, MIT Press, Cambridge, Mass, 2002.

[14] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, Journal of the ACM 12 (1) (1965) 23–41. doi:10.1145/321250.321253.
URL https://doi.org/10.1145/321250.321253

[15] A. Martelli, U. Montanari, An Efficient Unification Algorithm, ACM Transactions on Programming Languages and Systems 4 (2) (1982) 258–282. doi:10.1145/357162.357169.
URL https://dl.acm.org/doi/10.1145/357162.357169

[16] G. Hutton, Programming in Haskell, Cambridge University Press, 2016, google-Books-ID: 75C5DAAAQBAJ.

[17] J. de Muijnck-Hughes, E. Brady, W. Vanderbauwhede, Value-Dependent Session Design in a Dependently Typed Language, Electronic Proceedings in Theoretical Computer Science 291 (2019) 47–59. doi:10.4204/EPTCS.291.5.
URL http://arxiv.org/abs/1904.01288v1

[18] J. Hughes, L. Pareto, A. Sabry, Proving the correctness of reactive systems using sized types, in: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96, Association for Computing Machinery, New York, NY, USA, 1996, p. 410–423. doi:10.1145/237721.240882.
URL https://doi.org/10.1145/237721.240882

[19] Lift Contributors, Lift documentation (Sep. 2018).
URL https://lift-project.readthedocs.io/en/latest/

[20] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, C. 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, ACM, Barcelona Spain, 2017, pp. 556–571. doi:10.1145/3062341.3062354.
URL https://dl.acm.org/doi/10.1145/3062341.3062354

[21] N. Bell, J. Hoberock, Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA, in: W.-m. W. Hwu (Ed.), GPU Computing Gems Jade Edition, Applications of GPU Computing Series, Morgan Kaufmann, Boston, 2012, pp. 359–371. doi:10.1016/B978-0-12-385963-1.00026-5.
URL https://www.sciencedirect.com/science/article/pii/B9780123859631000265

[22] A. Collins, D. Grewe, V. Grover, S. Lee, A. Susnea, NOVA: A Functional Language for Data Parallelism, in: Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ACM, Edinburgh United Kingdom, 2014, pp. 8–13. doi:10.1145/2627373.2627375.

URL https://dl.acm.org/doi/10.1145/2627373.2627375

[23] B. He, W. Fang, Q. Luo, N. K. Govindaraju, T. Wang, Mars: a MapReduce framework on graphics processors, in: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, PACT '08, Association for Computing Machinery, New York, NY, USA, 2008, pp. 260–269. doi:10.1145/1454115.1454152.
URL https://doi.org/10.1145/1454115.1454152

[24] Y. J. Mo, J. Kim, J.-K. Kim, A. Mohaisen, W. Lee, Performance of deep learning computation with TensorFlow software library in GPU-capable multi-core computing platforms, in: 2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN), IEEE, Milan, 2017, pp. 240–242. doi:10.1109/ICUFN.2017.7993784.
URL http://ieeexplore.ieee.org/document/7993784/

[25] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, A. Cohen, Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions, arXiv:1802.04730 [cs]ArXiv: 1802.04730 (Jun. 2018).
URL http://arxiv.org/abs/1802.04730

[26] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, F. Catthoor, Polyhedral parallel code generation for CUDA, ACM Transactions on Architecture and Code Optimization 9 (4) (2013) 54:1–54:23. doi:10.1145/2400682.2400713.
URL https://doi.org/10.1145/2400682.2400713

[27] J. Hoffmann, K. Aehlig, M. Hofmann, Multivariate amortized resource analysis, in: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 357–370. doi:10.1145/1926385.1926427.
URL https://doi.org/10.1145/1926385.1926427

[28] L. Merino, M. Ghafari, C. Anslow, O. Nierstrasz, A systematic literature review of software visualization evaluation, Journal of Systems and Software 144 (2018) 165 – 180. doi:10.1016/j.jss.2018.06.027.
URL http://www.sciencedirect.com/science/article/pii/S0164121218301237

Appendix A. Omitted rules

This appendix contains rules that were omitted for brevity.

Appendix A.1. Meta functions

$$\text{TM-OnlyFree}\ \frac{onlyFree(S_1) \qquad onlyFree(S_2)}{onlyFree(S_1 \times S_2)} \qquad \text{TM-OnlyFreeBase}\ \frac{}{onlyFree(x)} \qquad \text{TM-Fresh}\ \frac{\sigma = \{S \mapsto \text{new fresh shape} | S \in FV(T)\}}{\sigma(T) = \mathsf{fresh}(T)}$$

Appendix A.2. Rules for Buffers

A buffer has a type if all its elements have the right shape and the length is correct. The explicit typing rules are shown in figure A.13.

$$\text{TM-Buffer}\ \frac{\forall i \in \{0, \ldots, n-1\}.\Gamma \vdash_{\text{E}} e_i : S}{\Gamma \vdash (\texttt{buf}_{S[n]}\ e_0, \ldots, e_{n-1}) : S[n]} \qquad \text{TM-BufferMult}\ \frac{\forall i \in \{1, \ldots n\}.\Gamma \vdash B_i : T_i}{\Gamma \vdash (B_1, \ldots, B_n) : (T_1, \ldots, T_n)}$$

$$\text{TM-StoreType}\ \frac{\Gamma = \{x \mapsto B_x \mid x \in \mathsf{dom}(\mathcal{M}) \ \wedge\ \vdash \mathcal{M}[x] : B_x\}}{\vdash \mathcal{M} : \Gamma}$$

Figure A.13: Rules for buffers and store types.The notation $a, \ldots, b$ or $e_a, \ldots, e_b$ is empty if $b < a$

Appendix A.3. Typing rules for expressions $\vdash_E$

The omitted rules for typing arithmetic expressions are shown below.

$$\text{TE-Var}\ \frac{x \mapsto S \in \Gamma}{\Gamma \vdash_{\text{E}} x : S} \qquad \text{TE-Int}\ \frac{n \text{ is an integer}}{\Gamma \vdash_{\text{E}} n : int} \qquad \text{TE-Float}\ \frac{n \text{ is an floating point number}}{\Gamma \vdash_{\text{E}} n : float}$$

$$\text{TE-Index}\ \frac{\Gamma \vdash_{\text{E}} e_i : \texttt{int} \qquad x \mapsto S[num_{la} \cdot n_l + num_b] \in \Gamma}{\Gamma \vdash_{\text{E}} (\texttt{index}\ x\ e_i) : S}$$

$$\text{TE-BinOp}\ \frac{S \in \{\texttt{int}, \texttt{float}\} \qquad \Gamma \vdash_{\text{E}} e_1 : S \qquad \Gamma \vdash_{\text{E}} e_2 : S \qquad binOp \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash_{\text{E}} (binOp\ e_1\ e_2) : S}$$

$$\text{TE-BinComp}\ \frac{S \in \{\texttt{int}, \texttt{float}\} \qquad \Gamma \vdash_{\text{E}} e_1 : S \qquad \Gamma \vdash_{\text{E}} e_2 : S \qquad binOp \in \{\texttt{>}, \texttt{<}, \texttt{==}\}}{\Gamma \vdash_{\text{E}} (binOp\ e_1\ e_2) : \texttt{int}} \qquad \text{TE-Fst}\ \frac{\Gamma \vdash_{\text{E}} e_1 : S_1 \times S_2}{\Gamma \vdash_{\text{E}} (\texttt{fst}\ e_1) : S_1}$$

$$\text{TE-Snd}\ \frac{\Gamma \vdash_{\text{E}} e_1 : S_1 \times S_2}{\Gamma \vdash_{\text{E}} (\texttt{snd}\ e_1) : S_2} \qquad \text{TE-Tuple}\ \frac{\Gamma \vdash_{\text{E}} e_1 : S_1 \qquad \Gamma \vdash_{\text{E}} e_2 : S_2}{\Gamma \vdash_{\text{E}} (\texttt{tuple}\ e_1\ e_2) : S_1 \times S_2} \qquad \text{TE-If}\ \frac{\Gamma \vdash_{\text{E}} e_c : \texttt{int} \qquad \Gamma \vdash_{\text{E}} e_t : S \qquad \Gamma \vdash_{\text{E}} e_f : S}{\Gamma \vdash_{\text{E}} (\texttt{if}\ e_c\ e_t\ e_f) : S}$$

$$\text{TE-Let}\ \frac{\Gamma \vdash_{\text{E}} e_v : S_1 \qquad (x \mapsto S_1) : \Gamma \vdash_{\text{E}} e_c : S}{\Gamma \vdash_{\text{E}} (\texttt{let}\ x\ e_v\ e_c) : S}$$

Appendix A.4. Reduction rules for expressions $\hookrightarrow_{\text{E}}$

The reduction relation $\hookrightarrow_{\text{E}}$ describing the evaluation of arithmetic expressions is shown below. The evaluation of the `(index ...)` construct can only be done in the context of a `(shpr* ...)` as shown in figure 9

$$\text{E-Arith}\ \frac{e \hookrightarrow_{\text{E}} e' \qquad E \neq \cdot}{E[e] \hookrightarrow_{\text{E}} E[e']} \qquad \text{E-BinOpPlus}\ \frac{num_r = num_1 + num_2}{\texttt{(+}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} num_r} \qquad \text{E-BinOpPlus}\ \frac{num_r = num_1 + num_2}{\texttt{(+}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} num_r}$$

$$\text{E-BinOpMinus}\ \frac{num_r = num_1 - num_2}{\texttt{(-}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} num_r} \qquad \text{E-BinOpMul}\ \frac{num_r = num_1 * num_2}{\texttt{(*}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} num_r} \qquad \text{E-BinOpDiv}\ \frac{num_r = num_1/num_2 \qquad num_2 \neq 0}{\texttt{(/}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} num_r}$$

$$\text{E-BinOpDiv0}\ \frac{}{\texttt{(/}\ num_1\ 0\texttt{)} \hookrightarrow_{\text{E}} 0} \qquad \text{E-BinOpMod}\ \frac{num_r = num_1 \% num_2 \qquad num_2 \neq 0}{\texttt{(\%}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} num_r} \qquad \text{E-BinOpMod0}\ \frac{}{\texttt{(\%}\ num_1\ 0\texttt{)} \hookrightarrow_{\text{E}} 0}$$

$$\text{E-BinCompLTT}\ \frac{num_1 < num_2}{\texttt{(<}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} 1} \qquad \text{E-BinCompLTF}\ \frac{num_1 \geq num_2}{\texttt{(<}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} 0} \qquad \text{E-BinCompEqT}\ \frac{num_1 = num_2}{\texttt{(==}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} 1}$$

$$\text{E-BinCompEqF}\ \frac{num_1 \neq num_2}{\texttt{(==}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} 0} \qquad \text{E-BinCompGTT}\ \frac{num_1 > num_2}{\texttt{(>}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} 1} \qquad \text{E-BinCompFTF}\ \frac{num_1 \leq num_2}{\texttt{(>}\ num_1\ num_2\texttt{)} \hookrightarrow_{\text{E}} 0}$$

$$\text{E-Fst}\ \frac{}{\texttt{(fst (tuple}\ e_1\ e_2\texttt{))} \hookrightarrow_{\text{E}} e_1} \qquad \text{E-Snd}\ \frac{}{\texttt{(snd (tuple}\ e_1\ e_2\texttt{))} \hookrightarrow_{\text{E}} e_2} \qquad \text{E-If0}\ \frac{}{\texttt{(if 0}\ e_t\ e_f\texttt{)} \hookrightarrow_{\text{E}} e_f}$$

$$\text{E-If}\ \frac{num \neq 0}{\texttt{(if}\ num\ e_t\ e_f\texttt{)} \hookrightarrow_{\text{E}} e_t} \qquad \text{E-Let}\ \frac{}{\texttt{(let}\ x\ v\ e\texttt{)} \hookrightarrow_{\text{E}} e[x/v]}$$

Appendix A.5. Explicit Definition of Substitution on Expressions

The definition of substitution is shown below. Note for the substitution of `(letB ...)`, that the bound y only affects `(retrive ...)` constructs. Buffers are not affected by substitution as they are typed under the empty environment and can therefore not hold variables.

Definition 5 (Substitution of values)**.**

Work lists and work items

$$\begin{aligned}
(wl \fatsemi w)[x/v] &= wl[x/v] \fatsemi w[x/v] \\
\texttt{(call}\ r\ \overline{e}\texttt{)}[x/v] &= \texttt{(call}\ r\ \overline{e}[x/v]\texttt{)} \\
\texttt{(retrive}\ \overline{x}\texttt{)}[x/v] &= \texttt{(retrive}\ \overline{x}\texttt{)} \\
\texttt{(letB}\ y\ wl_1\ wl_2\texttt{)}[x/v] &= \texttt{(letB}\ y\ wl_1[x/v]\ wl_2[x/v]\texttt{)} \\
\texttt{(buf}_{S[num]}\ \overline{e}\texttt{)}[x/v] &= \texttt{(buf}_{S[num]}\ \overline{e}\texttt{)}
\end{aligned}$$

Transformations

$$
\begin{aligned}
(\texttt{shpr}\ T\ x_i\ \overline{x_a}\ e)[x/v] &= (\texttt{shpr}\ T\ x_i\ \overline{x_a}\ e[x/v]) && x \notin \overline{x_a} \\
(\texttt{shpr}\ T\ x_i\ \overline{x_a}\ e)[x/v] &= (\texttt{shpr}\ T\ x_i\ \overline{x_a}\ e) && x \in \overline{x_a} \\
(\texttt{mapr}\ T\ x_i\ x_d\ e)[x/v] &= (\texttt{mapr}\ T\ x_i\ x_d\ e[x/v]) && x \notin \{x_i, x_d\} \\
(\texttt{mapr}\ T\ x_i\ x_d\ e)[x/v] &= (\texttt{mapr}\ T\ x_i\ x_d\ e) && x \in \{x_i, x_d\} \\
(\texttt{redr}\ T\ x_d\ x_a\ e_0\ e_b)[x/v] &= (\texttt{redr}\ T\ x_d\ x_a\ e_0[x/v]\ e_b[x/v]) && x \notin \{x_d, x_a\} \\
(\texttt{redr}\ T\ x_d\ x_a\ e_0\ e_b)[x/v] &= (\texttt{redr}\ T\ x_d\ x_a\ e_0[x/v]\ e_b) && x \in \{x_d, x_a\} \\
(\texttt{shpr*}\ \overline{D}\ \mathcal{M})[x/v] &= (\texttt{shpr*}\ \overline{D}[x/v]\ \mathcal{M})
\end{aligned}
$$

Arithmetic Expressions

$$
\begin{aligned}
(k\ \overline{e})[x/v] &= (k\ \overline{e}[x/v]) && k \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{fst}, \texttt{snd}, \texttt{if}, \texttt{tuple}\} \\
(\texttt{let}\ y\ \overline{e})[x/v] &= (\texttt{let}\ y\ \overline{e}[x/v]) && x \neq y \\
(\texttt{let}\ x\ e_1\ e_2)[x/v] &= (\texttt{let}\ x\ e_1[x/v]\ e_2) \\
(\texttt{index}\ y\ e)[x/v] &= (\texttt{index}\ y\ \overline{e}[x/v]) \\
x[x/v] &= v \\
y[x/v] &= y && x \neq y
\end{aligned}
$$

Appendix A.6. Application of Unifiers
The rules for applying unifiers are shown below.

Substitution on terms

$$
\begin{aligned}
(\sigma, \langle \lambda(a \cdot n + b)/(f_1 \cdot l + f_2) \rangle)(B[e]) &= \sigma(B[\langle \lambda(a \cdot n + b)/(f_1 \cdot l + f_2) \rangle\, e]) \\
(\sigma, \langle A/A' \rangle)(B[e]) &= \sigma\left((\langle A/A' \rangle\, B)[e]\right)
\end{aligned}
$$

$$
\begin{aligned}
\langle \lambda(a \cdot n + b)/(f_1 \cdot l + f_2) \rangle\, (p \cdot n + q) &= f_1(p, q) \cdot l + f_2(p, q) \\
\langle \lambda(a \cdot n + b)/(f_1 \cdot l + f_2) \rangle\, (p \cdot m + q) &= p \cdot m + q && m \neq n
\end{aligned}
$$

Shape substitution

$$
\begin{aligned}
\langle A/A' \rangle\, A &= A' \\
\langle A/A' \rangle\, B &= B && A \neq B \\
\langle A/A' \rangle\, (S_1 \times S_2) &= \langle A/A' \rangle\, (S_1) \times \langle A/A' \rangle\, (S_2) \\
\langle A/A' \rangle\, (\texttt{int}) &= \texttt{int} \\
\langle A/A' \rangle\, (\texttt{float}) &= \texttt{float}
\end{aligned}
$$

Substitution fusing

$$
\begin{aligned}
&\sigma(u_1, \ldots, u_m) = (\sigma(u_1), \ldots, \sigma(u_m)) \\
&(\sigma, \langle B/B' \rangle)(\langle A/A' \rangle) = \sigma(\langle A/(\langle B/B' \rangle\, (A')) \rangle) \\
&(\sigma, \langle \lambda(a \cdot m + b)/(f_1'(a, b) \cdot k + f_2'(a, b)) \rangle)(\langle \lambda(a \cdot n + b)/(f_1(a, b) \cdot l + f_2(a, b)) \rangle) \\
&\quad = \begin{cases} \sigma(\left\langle \lambda(a \cdot n + b) \middle/ \left(f_1'\Big(f_1(a, b), f_2(a, b)\Big) \cdot k + f_2'\Big(f_1(a, b), f_2(a, b)\Big) \right) \right\rangle) & m = l \\ \sigma(\langle \lambda(a \cdot n + b)/(f_1(a, b) \cdot l + f_2(a, b)) \rangle) & m \neq l \end{cases}
\end{aligned}
$$

Appendix B. Omitted lemmas and proofs

Appendix B.1. Join Preserves Containment of Free Variables

Proof for lemma 1. By induction on n:

IB ($n = 0$): trivial ($B|\mathcal{C} = B^\star|\mathcal{C}^\star$)

IS ($n > 0$): Let $T_n := \overline{B_2} \twoheadrightarrow \overline{B_3}$ and $\overline{B'}|\mathcal{C}' := \mathsf{join}^\star(B|\mathcal{C}, T_1, \ldots, T_{n-1})$. From the definition of join: $\sigma = \mathsf{unify}(\overline{B'}, \overline{B_2})$ and $FV(\overline{B_3}) \subseteq FV(\overline{B_2}) \therefore FV(\sigma(\overline{B_3})) \subseteq FV(\sigma(\overline{B_2})) = FV(\sigma(\overline{B'})) \overset{IH}{\subseteq} FV(\sigma(\mathcal{C}')) \therefore FV(\overline{B^\star}) = FV(\sigma(\overline{B_3})) \subseteq FV(\sigma(\mathcal{C}')) = FV(\mathcal{C}^\star)$

□

Appendix B.2. Constructed types have contained Free Variables

Proof for lemma 2. By well-founded induction on the derivation of $\mathcal{A} \mid \Gamma \vdash_{\text{P}} wl : B|\mathcal{C}$. We do a case analysis on the last applied rule of the derivation.

TP-Retrive , in this case $T = (B_{x_1}, \ldots, B_{x_n})$ and $\mathcal{C} = \{x'_1 \mapsto B_{x'_1}, \ldots, x'_n \mapsto B_{x'_m}\}$ for fresh buffer types $B_{x'_1}$ to $B_{x'_m}$ with $\{x'_1, \ldots, x'_m\} = \bigcup_{i=1}^{m} \{x_{\min\{j \mid x_j = x_i\}}\}$. The set $\{x'_1, \ldots, x'_m\}$ contains all the variable names used in in the return. For every returned variable x_i it holds that $x_i = x_{\min\{j \mid x_j = x_i\}}$. And thus also that that $B_{x_i} = B_{x_{\min\{j \mid x_j = x_i\}}}$. Consequently, $\bigcup_{i=0}^{n}\{B_{x_i}\} = \bigcup_{i=0}^{m}\{B_{x'_i}\}$. From which it immediately follows that:

$$FV(T) = FV\left(\bigcup_{i=0}^{n}\{B_{x_i}\}\right) = FV\left(\bigcup_{i=0}^{m}\{B_{x'_i}\}\right) = FV(\mathcal{C})$$

And thus also $FV(T) \subseteq FV(\mathcal{C})$

TP-LetB , in this case $wl = (\texttt{letB}\ x\ wl_1\ wl_2)$. The premise of the rule states that $\mathcal{A} \mid \Gamma \vdash_{\text{P}} wl_1 : (B_1)|\mathcal{C}_1$ and $\mathcal{A} \mid \Gamma \vdash_{\text{P}} wl_2 : \overline{B_L}|\mathcal{C}_L$. To which we may apply the IH to get that $FV(B_1) \subseteq FV(\mathcal{C}_1)$ and $FV(\overline{B_L}) \subseteq FV(\mathcal{C}_L)$. The TP-LetB rule also gives us that the resulting type $\overline{B}|\mathcal{C}$ is $\mathsf{japply}\,(x \mapsto B_1, \mathcal{C}_1, \overline{B_L}, \mathcal{C}_L) = \sigma(\overline{B_L}|\mathcal{C}_1 \cup (\mathcal{C}_L \setminus x))$ for some σ such that $FV(\sigma(\mathcal{C}_L[x])) = FV(\sigma(B_1)) \subseteq FV(\sigma(\mathcal{C}_1))$, thus $FV(\sigma(\mathcal{C}_L)) \subseteq FV(\sigma(\mathcal{C}_1 \cup (\mathcal{C}_L \setminus x)))$. We may conclude:

$$FV(\overline{B}) = FV(\sigma(\overline{B_L})) \subseteq FV(\sigma(\mathcal{C}_L)) \subseteq FV(\sigma(\mathcal{C}_1 \cup (\mathcal{C}_L \setminus x))) = FV(\mathcal{C})$$

TP-Buf , in this case $FV(\overline{B}) = \emptyset$, B is a concrete buffer type. Thus $\emptyset = FV(\overline{B}) \subseteq FV(\mathcal{C})$.

TP-List Let the work list be $wl_1 \fatsemi w_2$, such that $\mathcal{A} \mid \Gamma \vdash_{\text{P}} wl_1 \fatsemi w_2 : B|\mathcal{C}$. From TP-List we have:

- $\mathcal{A} \mid \Gamma \vdash_{\text{P}} wl_1 : \overline{B_1}|\mathcal{C}_1$ (premise of TP-List), and by the IH: $FV(\overline{B_1}) \subseteq FV(\mathcal{C}_1)$
- $\mathcal{A} \mid \Gamma \vdash_{\text{P}} w_2 : \overline{B_2} \twoheadrightarrow \overline{B_3}|\emptyset$ (premise of TP-List combined with definition of join)
- $\overline{B}|\mathcal{C} = \mathsf{join}\,(\overline{B_1}|\mathcal{C}_1, \overline{B_2} \twoheadrightarrow \overline{B_3})$

The required result follow immediately from the IH and lemma 1 (Join preserves containment of free variables).

TP-Call can not be the last rule as they produce an arrow (not a B).

□

Appendix B.3. The output of validate⋆ is concrete

Lemma 8. *The output of* $\mathsf{validate}^\star(\mathcal{M}, \overline{B}|\mathcal{C}_1\ , T_1, \ldots, T_n)$ *is concrete if it exists*

Proof. The unifier σ in the premise of TP-VALIDATE ensures that $\sigma(\mathcal{C}) = \mathcal{M}$ and because store is concrete and $FV(B) \subseteq FV(\mathcal{C})$, it must hold that $\sigma(B)$ is concrete. □

Corollary 9 (Store consistent substitution preserves types)**.** *If*

- $\vdash$ *(main (prog* $\mathcal{A}$ $w_1 \fatsemi wl$*)* $\mathcal{M}$*)* : B
- *(main (prog* $\mathcal{A}$ $w_1 \fatsemi wl$*)* $\mathcal{M}$*)* $\rightsquigarrow_{\mathrm{P}}$ *(main (prog* $\mathcal{A}$ $w_2 \fatsemi wl$*)* $\mathcal{M}$*)*
- $w_2 = \sigma(w_1)$
- σ *is consistent with* $\mathcal{M}$

then $\vdash$ *(main (prog* $\mathcal{A}$ $w_2 \fatsemi wl$*)* $\mathcal{M}$*)* : B

Lemma 10. *If for any* $\mathcal{M}$, B_0, B_1 $\mathcal{C}_1$, $\mathcal{C}_{rest}$ *and* $T_1, \ldots, T_n$

- $\mathsf{validate}^\star(\mathcal{M}, \sigma\,(\,B_0|\mathcal{C}_1 \cup \mathcal{C}_{rest}\,)\,, T_1, \ldots, T_n)$ *has a value, and*
- $FV(B_1) \subseteq FV(\mathcal{C}_1)$

then $\mathsf{validate}^\star(\mathcal{M},\ B_1|\mathcal{C}_1\,)$ *has a value.*

Proof. Because $\mathsf{validate}^\star(\mathcal{M}, \sigma\,(\,B_0|\mathcal{C}_1 \cup \mathcal{C}_{rest}\,)\,, T_1, \ldots, T_n)$ has a value, $\sigma(\mathcal{C}_1 \cup \mathcal{C}_{rest})$ and $\mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1 \cup \mathcal{C}_{rest})}$ must be unifiable and $\mathsf{dom}(\mathcal{C}_1 \cup \mathcal{C}_{rest})) \subseteq \mathsf{dom}(\mathcal{M})$ by TP-VALIDATE. And so, $\sigma(\mathcal{C}_1)$ and $\mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)}$ must also be unifiable. Which can only hold if, $\mathcal{C}_1$ and $\mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)}$ are also unifiable. Additionally we know that $\mathsf{dom}(\mathcal{C}_1) \subseteq \mathsf{dom}(\mathcal{C}_1 \cup \mathcal{C}_{rest})) \subseteq \mathsf{dom}(\mathcal{M})$. Finally, it is given that $FV(B_1) \subseteq FV(\mathcal{C}_1)$.

We conclude that all premises of TP-VALIDATE are fulfilled to give $\mathsf{validate}^\star(\mathcal{M},\ B_1|\mathcal{C}_1\,)$ a value. □

Appendix B.4. Proofs and Corollaries of the Validation Lemma

for lemma 3. Take an arbitrary σ consistent with $\mathcal{M}$ and $\mathcal{C}_1$, the proof continues by induction on n.

IB $n = 0$, so we may unfold validate (remove the $\star$) and reformulate our goal as follows. We need to show that:

$$B = \mathsf{validate}(\mathcal{M},\ B_1|\mathcal{C}_1\,) \overset{?}{\iff} B = \mathsf{validate}(\mathcal{M}, \sigma\,(\,B_1|\mathcal{C}_1\,))$$

⇒ By TP-VALIDATE we know that $B = \mathsf{validate}(\mathcal{M},\ B_1|C_1\,) = \sigma_I(B_1)$ for an $\sigma_I = \mathsf{unify}\Big(\mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)}, \mathcal{C}_1\Big)$. Our goal is to prove that $\mathsf{validate}(\mathcal{M}, \sigma\,(\,B_1|C_1\,)) = \sigma_I(B_1) = B$.

Because σ is consistent with $\mathcal{M}$ and C_1, we know ,by definition, that $\sigma \subseteq \mathsf{unify}(\mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)}, \mathcal{C}_1) = \sigma_I$. As $\mathcal{M}$ is concrete and $\mathsf{dom}(\mathcal{C}_1) \subseteq \mathsf{dom}(\mathcal{M})$, all free variables in $\mathcal{C}_1$ must be bound to a concrete value by σ_I. σ is a subset of σ_I and hence also only transforms variables to concrete terms. Consequently, all free variables in $\mathcal{C}_1$ that are also in $\mathsf{dom}(\sigma)$ will be replaced by concrete terms in $\sigma(\mathcal{C}_1)$, the variables that remain in $\sigma(\mathcal{C}_1)$ are substituted with concrete terms by $\sigma_I \setminus \sigma$ and it will be the case that $(\sigma_I \setminus \sigma)(\sigma(\mathcal{C}_1)) \subseteq \mathcal{M}$ (see results from Robinson et al.[14]), thus:

$$\sigma_I \setminus \sigma = \mathsf{unify}\left(\mathcal{M}\big|_{\mathsf{dom}(\sigma(\mathcal{C}))}, \sigma(\mathcal{C})\right)$$

We also know that $FV(\sigma(B_1)) \subseteq FV(\sigma(\mathcal{C}_1))$ because we had that $FV(B_1) \subseteq FV(\mathcal{C}_1)$. Additionally, we know that $\mathsf{dom}(\sigma(\mathcal{C}_1)) = \mathsf{dom}(\mathcal{C}_1) \subseteq \mathsf{dom}(\mathcal{M})$.

Now, we may conclude that:

$$\mathsf{validate}(\mathcal{M}, \sigma\,(\,B_1|\mathcal{C}\,)) = (\sigma_I \setminus \sigma)(\sigma(B_1)) = \sigma_I(B_1) = \mathsf{validate}(\mathcal{M},\, B_1|C_1\,) = B$$

Also note that $FV(\sigma(B_1)) \subseteq FV(\sigma(\mathcal{C}_1))$ because $FV(B_1) \subseteq FV(\mathcal{C}_1)$. And that $\mathsf{dom}(\mathcal{C}_1) = \mathsf{dom}(\sigma(\mathcal{C}_1))$.

$\boxed{\Leftarrow}$ We know that $FV(B_1) \subseteq FV(\mathcal{C}_1)$ because this is given and, we know that $\mathsf{dom}(\mathcal{C}_1) = \mathsf{dom}(\sigma(\mathcal{C}_1)) \subseteq \mathsf{dom}(\mathcal{M})$. So two of the premises of TP-VALIDATE for the left side validate are satisfied. The remaining premise requires the existence of $\mathsf{unify}(\mathcal{M}, \mathcal{C}_1)$. Because the validate in the given works, we know that $\mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_1))$ exists and even that: $B = \mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_1))\,(\sigma(B_1))$. From this we know that $\sigma(\mathcal{C}_1)$ is unifiable with $\mathcal{M}$, so $\mathcal{C}_1$ must also be unifiable, therefore, $\mathsf{unify}(\mathcal{M}, \mathcal{C}_1)$ exists.

What remains to be proven is that:

$$\mathsf{validate}(\mathcal{M}, \sigma\,(\,B_1|\mathcal{C}_1\,)) = \mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_1))(\sigma(B_1)) \stackrel{?}{=} \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)\,(B_1) = \mathsf{validate}(\mathcal{M},\, B_1|\mathcal{C}_1\,)$$

It suffices to prove that $\forall X \in FV(B_1).\mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_1))\,(\sigma(X)) = \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)\,(X)$ because it immediately implies our goal. Take an arbitrary $X \in FV(B_1)$, this X must also be in $FV(\mathcal{C}_1)$ and thus also in $\mathsf{dom}(\mathsf{unify}(\mathcal{M}, \mathcal{C}_1))$. There are two possibilities:

- $X \in \mathsf{dom}(\sigma)$. In this case, $\sigma(X)$ is a concrete value because σ is store compatible. So, we may apply any unifier to $\sigma(X)$ without any effect . Because $\sigma \subseteq \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)$, we know that $\mathsf{unify}(\mathcal{M}, \mathcal{C}_1)\,(X) = \sigma(X)$. So:

$$\mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_1))\,(\sigma(X)) = \sigma(X) = \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)\,(X)$$

- $X \notin \mathsf{dom}(\sigma)$. Because $X \in FV(\mathcal{C}_1)$ it must occur in one of its keys: $\exists k.X \in FV(\mathcal{C}_1[k])$. We know that $\mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_1))\,(\sigma(\mathcal{C}_1)) = \mathcal{M}\big|_{\mathsf{dom}(\sigma(\mathcal{C}_1))} = \mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)}$, the same holds without σ. So we may write:

$$\begin{array}{rrcccc} & \mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_1))\,(\sigma(\mathcal{C}_1)) & = & \mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)} & = & \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)\,(\mathcal{C}_1) \\ \stackrel{k \in \mathsf{dom}(\mathcal{C}_1)}{\Rightarrow} & \mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_1))\,(\sigma(\mathcal{C}_1[k])) & = & \mathcal{M}[k] & = & \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)\,(\mathcal{C}_1[k]) \end{array}$$

The above is only possible if $\mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_1))\,(\sigma(X)) = \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)\,(X)$ because $X \in FV(\mathcal{C}_1[k])$.

$\boxed{\texttt{IH}}$ $\forall 0 \leq n < n_1.B = \mathsf{validate}^\star(\mathcal{M},\, B_1|C_1\,, T_1, \ldots, T_n) \iff B = \mathsf{validate}^\star(\mathcal{M}, \sigma\,(\,B_1|C_1\,), T_1, \ldots, T_n)$

$\boxed{\texttt{IS}}$ We must show that:

$$B = \mathsf{validate}^\star(\mathcal{M},\, B_1|C_1\,, T_1, \ldots, T_{n_1}) \iff B = \mathsf{validate}^\star(\mathcal{M}, \sigma\,(\,B_1|C_1\,), T_1, \ldots, T_{n_1})$$

Let $T_{n_1} = B_X \twoheadrightarrow B_Y$ our goal is now equivalent to showing that $B = \mathsf{validate}(\mathcal{M}, \underbrace{B_1|C_1\,, T_1, \ldots, T_{n_1-1}}_{B^*_{n_1-1}|\mathcal{C}^*_{n_1-1}}, B_X \twoheadrightarrow B_Y)$ if and only if $\iff B = \mathsf{validate}(\mathcal{M}, \sigma\,(\,B_1|C_1\,), T_1, \ldots, T_{n_1-1}, B_X \twoheadrightarrow B_Y)$.

The proof is the same in both directions. We will work out the $\Rightarrow$ direction, $\Leftarrow$ is analogous.

Since $\mathsf{validate}^\star(\mathcal{M},\, B_1|\mathcal{C}_1\,, T_1, \ldots, T_{n_1-1}, B_X \twoheadrightarrow B_Y)$ has a value we may conclude that:

- $B^*_{n_1}|\mathcal{C}^*_{n_1} := \mathsf{join}^\star(\,B_1|\mathcal{C}_1\,, T_1, \ldots, T_{n_1-1}, B_X \twoheadrightarrow B_Y)$ has a value.
  From this existence we may further derive the existence of $B^*_{n_1-1}|\mathcal{C}^*_{n_1-1} := \mathsf{join}^\star(\,B_1|\mathcal{C}_1\,, T_1, \ldots, T_{n_1-1})$.
  From TM-JOIN we get that $B^*_{n_1}|\mathcal{C}^*_{n_1} = \sigma_j\,(\,B_Y|\mathcal{C}^*_{n_1-1})$ with $\sigma_j = \mathsf{unify}(B^*_{n_1-1}, B_X)$.

- $\sigma_{n_1} := \mathsf{unify}(\mathcal{M}, \sigma_j(\mathcal{C}^*_{n_1-1}))$ has a value that ensures that $\sigma_{n_1}(\sigma_j(B_Y)) = B$.
  Because $\sigma_{n_1}(\sigma_j(\mathcal{C}^*_{n_1-1})) \subseteq \mathcal{M}$ we may also derive that $\sigma_{n_1-1} := \mathsf{unify}(\mathcal{M}, \mathcal{C}^*_{n_1-1})$ exists.
- $FV(B_1) \subseteq FV(\mathcal{C}_1)$ and with lemma 1 we get: $FV(B^*_{n_1-1}) \subseteq FV(\mathcal{C}^*_{n_1-1})$.
- $\mathsf{dom}\left(\mathcal{C}^*_{n_1-1}\right) = \mathsf{dom}(\sigma_j(\mathcal{C}_{n_1})) = \mathsf{dom}(\mathcal{C}_{n_1}) \subseteq \mathsf{dom}(\mathcal{M})$.

The last three points give us all the premises of TP-VALIDATE to derive that some B' exists for the following result.

$$B' = \mathsf{validate}^\star\left(\mathcal{M},\, B^*_{n_1-1}\middle|\mathcal{C}^*_{n_1-1}\right) = \mathsf{validate}^\star(\mathcal{M},\, B_1|\mathcal{C}_1\,, T_1, \ldots, T_{n_1-1}) = \sigma_{n_1-1}(B^*_{n_1-1})$$

To which we may apply the IH and get the following: (the same result, removing the σ, is obtained for the $\Leftarrow$ direction):

$$B' = \mathsf{validate}^\star(\mathcal{M}, \sigma(\,B_1|\mathcal{C}_1\,), T_1, \ldots, T_{n_1-1})$$

As before, we may define $B^{*'}_{n_1-1}\middle|\mathcal{C}^{*'}_{n_1-1} := \mathsf{join}^\star(\sigma(\,B_1|\mathcal{C}_1\,), T_1, \ldots, T_{n_1-1})$ and know that $\sigma'_{n_1-1} := \mathsf{unify}(\mathcal{M}, \mathcal{C}^{*'}_{n_1-1})$ exists.

Now, we observe that it must hold that $B' = \sigma_{n_1-1}(B^*_{n_1-1}) = \sigma_{n_1}(\sigma_j(B^*_{n_1-1}))$. Because $\sigma_{n_1-1}(X) = \sigma_{n_1}(\sigma_j(X))$ for any free variable X of $B^*_{n_1-1}$. Formally we use lemma 1 to get that $FV(B^*_{n_1-1}) \subseteq FV(\mathcal{C}_{n_1-1})$, and then we use the definition of unify combined with the fact that $\mathsf{range}(\mathcal{M})$ is concrete:

$$\forall X \in FV(B^*_{n_1-1}).\exists k.X \in FV(C^*_{n_1-1}[k]).\sigma_{n_1-1}(C^*_{n_1-1}[k]) = \mathcal{M}[k] = \sigma_{n_1}(\sigma_j(C^*_{n_1-1}[k]))$$

We may write: $\sigma'_{n_1-1}(B^{'*}_{n_1-1}) = B' = \sigma_{n_1}(\sigma_j(B^*_{n_1-1})) = \sigma_{n_1}(\sigma_j(B_X))$ and conclude that $B^{'*}_{n_1-1}$ and B_X are unifiable and, even stronger, that they both can be unified to the concrete value B'. Because $\sigma'_j := \mathsf{unify}(B^{'*}_{n_1-1}, B_X)$ is a most-general unifier, it holds that $\sigma'_j(B^*_{n_1-1})$ (and $\sigma'_j(B_X)$) can be unified to B'.

- Define T as the unique unifier such that $T(\sigma'_j(B^{*'}_{n_1-1})) = B'$ with $\mathsf{dom}(T) = FV(\sigma'_j(B^{*'}_{n_1-1}))$ and $\mathsf{range}(T)$ concrete values. This T certainly exist because of the argument above.
- Let $S := \sigma'_{n_1-1} = \mathsf{unify}(\mathcal{M}, \mathcal{C}^{*'}_{n_1-1})$.
- Let $W := S\big|_{FV(C^{*'}_{n_1-1})\setminus\mathsf{dom}(T)} \cup T$
- We want to show that $W = \mathsf{unify}(\mathcal{M}, \sigma'_j(\mathcal{C}^{*'}_{n_1-1}))$. To this end, we need that $\mathsf{dom}(W) = FV(\mathcal{C}^{*'}_{n_1-1})$ (holds by definition) and that $W(\mathcal{C}^{*'}_{n_1-1}) \subseteq \mathcal{M}$. We prove the later condition by proving that $W(\sigma'_j(C^{*'}_{n_1-1}[v])) = \mathcal{M}[v]$ for arbitrary $v \in \mathsf{dom}(\mathcal{C}^{*'}_{n_1-1})$. This holds if $\forall X \in FV(C^{*'}_{n_1-1}[v]).W(\sigma'_j(X)) = S(X)$. Arbitrarily take such an X (if $FV(C^{*'}_{n_1-1}[v]) = \emptyset, W(\sigma'_j(C^{*'}_{n_1-1}[v])) = C^{*'}_{n_1-1}[v] = S(C^{*'}_{n_1-1}[v]) = \mathcal{M}[v]$). There are three possibilities:
  - if $X \notin \mathsf{dom}(\sigma'_j) \wedge X \in \mathsf{dom}(T) \therefore X \in FV(B^{*'}_{n_1-1})$: $W(\sigma'_j(X)) = W(X) = T(X) = S(X)$ since $\sigma'_j(X) = X$ and $\forall X \in FV(B^{*'}_{n_1-1}).S(X) = T(\sigma'_j(X))$ because $S(B^*_{n_1-1}) = T(\sigma'_j(B^*_{n_1-1}))$
  - if $X \notin \mathsf{dom}(\sigma'_j) \wedge X \notin \mathsf{dom}(T)$: $W(\sigma'_j(X)) = W(X) = S(X)$
  - if $X \in \mathsf{dom}(\sigma'_j) \therefore X \in FV(B^{*'}_{n_1-1})$: $W(\sigma'_j(X)) = T(\sigma'_j(X)) = S(X)$
- We now have: $W = \mathsf{unify}\left(\mathcal{M}, \sigma'_j(\mathcal{C}^{*'}_{n_1-1})\right)$ and $W(\sigma'_j(B^{*'}_{n_1-1})) = W(\sigma'_j(B_X)) = B'$
- We had: $\sigma_{n_1}(\sigma_j(B^*_{n_1-1})) = \sigma_{n_1}(\sigma_j(B_X)) = B'$ and $\sigma_{n_1}(\sigma_j(B_Y)) = B$

- So: $W(\sigma'_j(B^{*'}_{n_1-1})) = \mathsf{unify}\Big(\mathcal{M}, \sigma'_j(\mathcal{C}^{*'}_{n_1-1})\Big)\,(\sigma'_j(B^{*'}_{n_1-1})) = \mathsf{unify}\Big(\mathcal{M}, \sigma'_j(\mathcal{C}^{*'}_{n_1-1})\Big)\,(\sigma'_j(B_X)) = W(\sigma'_j(B_X)) = B'$
  and thus $W(\sigma'_j(B_Y)) = \mathsf{unify}\Big(\mathcal{M}, \sigma'_j(\mathcal{C}^{*'}_{n_1-1})\Big)\,(\sigma'_j(B_Y)) = B$ because $FV(B_Y) \subseteq FV(B_X)$.
- And hence $B = \mathsf{validate}^{\star}\Big(\mathcal{M}, \sigma'_j(\,B_Y \big| \mathcal{C}^{*'}_{n_1-1}\,)\Big) = \mathsf{validate}^{\star}\Big(\mathcal{M},\, B^{*'}_{n_1} \big| \mathcal{C}^{*'}_{n_1}\Big) = \mathsf{validate}^{\star}(\mathcal{M}, \sigma\,(\,B_1|C_1\,)\,, T_1, \ldots, T_{n_1})$

□

We can derive two corollary of lemma 3.

Corollary 11. *For any σ consistent with $\mathcal{M}$ and $\mathcal{C}_1$ such that $FV(\sigma(B_1)) = \emptyset$,*
$B = \mathit{validate}^{\star}(\mathcal{M},\, B_1|\mathcal{C}_1\,, T_1, \ldots, T_n) \iff B = \mathit{validate}^{\star}(\mathcal{M}, \sigma\,(\,B_1|\emptyset\,)\,, T_1, \ldots, T_n)$

Proof of corollary 11. $\boxed{\Rightarrow}$ We know that $B = \mathsf{validate}^{\star}(\mathcal{M}, \sigma\,(\,B_1|\sigma(C_1)\,)\,, T_1, \ldots, T_n)$ because of lemma 3. Looking at TM-Join, we notice that the final constraint $\mathcal{C}$ in TP-Validate is: $\sigma_n(\sigma_{n-1}(\ldots\sigma_1(\sigma(C_1))\ldots))$. The corresponding constraint on the right-hand side of " $\iff$ " is $\sigma_n(\sigma_{n-1}(\ldots\sigma_1(\sigma(\emptyset))\ldots)) = \emptyset$. Since TM-Join ensures that no new free variables are introduced, all free variables in the result of join must be those of $\sigma(B_1)$, in this case none. Hence, $FV(\sigma(B)) \subseteq FV(\emptyset)$. The other premises of validate are also trivially satisfied ($\emptyset = \mathsf{unify}(\mathcal{M}, \emptyset)$, $\mathsf{dom}(\mathcal{C}) = \emptyset \subseteq \mathsf{dom}(\mathcal{M})$).

$\boxed{\Leftarrow}$ We know that $B = \mathsf{validate}^{\star}(\mathcal{M},\, \sigma(B_1)|\emptyset\,, T_1, \ldots, T_n)$ for a $\sigma \subseteq \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)$. For any $\mathcal{M}' = \mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)} \subseteq \mathcal{M}$ we know that $\mathcal{M}'$ only has concrete values, any application of a unifier to $\mathcal{M}'$ will result in $\mathcal{M}'$. The the final $\mathcal{C}$ derived for $\mathsf{validate}^{\star}(\mathcal{M},\, \sigma(B_1)|\mathcal{M}'\,, T_1, \ldots, T_n)$ will be $\mathcal{M}'$. So:

$$\mathsf{unify}(\mathcal{M}, \mathcal{C}) = \mathsf{unify}(\mathcal{M}, \mathcal{M}') = \emptyset = \mathsf{unify}(\mathcal{M}, \emptyset)$$

and hence

$$B = \mathsf{validate}^{\star}(\mathcal{M},\, \sigma(B_1)|\mathcal{M}'\,, T_1, \ldots, T_n) = \mathsf{validate}^{\star}(\mathcal{M}, \sigma(\,B_1|\mathcal{M}'\,), T_1, \ldots, T_n)$$

and by lemma 3 we have:

$$B = \mathsf{validate}^{\star}(\mathcal{M},\, B_1|\mathcal{C}'_1\,, T_1, \ldots, T_n)$$

□

The second corollary follows immediately form the previous one.

Corollary 12. *For any σ consistent with $\mathcal{M}$ and $\mathcal{C}_1$ such that $FV(\sigma(B_1)) = \emptyset$,*
$B = \mathit{validate}^{\star}(\mathcal{M},\, B_1|\mathcal{C}_1\,, T_1, \ldots, T_n) \iff B = \mathit{validate}^{\star}(\mathcal{M}, \sigma\,(\,B_1|\mathcal{M}\,)\,, T_1, \ldots, T_n) = \mathit{validate}^{\star}(\mathcal{M},\, \sigma\,(B_1)|\mathcal{M}\,, T_1, \ldots, T_n)$

Appendix B.5. Substitution Lemmas

Lemma 13. *For any arithmetic expression e, any valid environment Γ with $x \in \Gamma$ and any value v such that $\vdash_{\mathrm{E}} v : \Gamma[x]$, it holds that if $\Gamma \vdash_{\mathrm{E}} e : S$, then $\Gamma \vdash_{\mathrm{E}} e[x/v] : S$*

Lemma 14. *For any transformation t, and any valid environment Γ with $x \in \Gamma$ and any value v such that $\vdash_{\mathrm{E}} v : \Gamma[x]$, it holds that if $\Gamma \mid \emptyset \vdash_{\mathrm{P}} t : B_1 \twoheadrightarrow B_2|\emptyset$, then $\Gamma \mid \emptyset \vdash_{\mathrm{P}} t[x/v] : B_1 \twoheadrightarrow B_2|\emptyset$.*

Appendix B.6. Preservation

Proof. Proof for lemma 6

All programs p have the (canonical) form $\texttt{(prog}\ \mathcal{A}\ wl\texttt{)}$. So we know that $\vdash \texttt{(main (prog}\ \mathcal{A}\ wl\texttt{)}\ \mathcal{M}\texttt{)} : B$ for some $\mathcal{A}$ and wl. By TP-Main and TP-Prog, we have that all definitions in $\mathcal{A}$ are well-typed by TT-Abst and that $\emptyset \mid \mathcal{A} \vdash_{\text{P}} wl : B_{wl}|\mathcal{C}_{wl}$ with $B = \mathsf{validate}(\mathcal{M}, B_{wl}|\mathcal{C}_{wl})$.

We will prove preservation by structural induction on the evaluation relation $\rightsquigarrow_P$. We inspect all possible last rules of the derivation tree.

A general approach we will take with most rules is to show that $\mathsf{validate}(\mathcal{M}, B_{wl}|\mathcal{C}_{wl}) = B = \mathsf{validate}(\mathcal{M}, B'_{wl}|\mathcal{C}'_{wl})$ with $B'_{wl}|\mathcal{C}'_{wl}$ the type of wl', the work list after taking the step. We may do this because our evaluation rules only make changes to the work list, they may access elements outside the work list, but they do not change them.

For rules of the form $E_w\,[wl_1] \rightsquigarrow_{\text{P}} E_w\,[wl'_1]$, we may formulate the following sufficient condition to ease proving preservation.

$$(\emptyset \mid \mathcal{A} \vdash_{\text{P}} wl_1 : B_1|\mathcal{C}_1 \wedge \emptyset \mid \mathcal{A} \vdash_{\text{P}} wl'_1 : B_1|\mathcal{C}_1) \implies E_w\,[wl_1] \rightsquigarrow_{\text{P}} E_w\,[wl'_1] \text{ preserves types}$$

In these cases, will only show that the type of wl_1 equals that of wl'_1. This suffices for preservation because $wl = E_w\,[wl_1]$ for some E_w. To be precise, wl must have the from $wl_1 \fatsemi w_2 \fatsemi \ldots \fatsemi w_n$ for some n. For $n = 1$ the implication is trivial because $wl_1 = wl$ and $wl'_1 = wl'$ and therefore $B_{wl}|\mathcal{C}_{wl} = B_1|\mathcal{C}_1 = B'_1|\mathcal{C}'_1 = B'_{wl}|\mathcal{C}'_{wl}$. For $n > 1$, we derive $\emptyset \mid \mathcal{A} \vdash_{\text{P}} wl : B_{wl}|\mathcal{C}_{wl}$ by repeated application of TP-List. Hence, $B_{wl}|\mathcal{C}_{wl} = \mathsf{join}^\star(B_1|\mathcal{C}_1, T_2, \ldots, T_n)$ with $\emptyset \mid \mathcal{A} \vdash_{\text{P}} wl_1 : B_1|\mathcal{C}_1$ and $\forall i \in \{2, \ldots, n\}.\emptyset \mid \mathcal{A} \vdash_{\text{P}} w_i : T_i|\emptyset$. In this case we have that $B = \mathsf{validate}^\star(\mathcal{M}, B_1|\mathcal{C}_1, T_2, \ldots, T_n)$. We must show that $\mathsf{validate}^\star(\mathcal{M}, B'_1|\mathcal{C}'_1, T_2, \ldots, T_n) = B$ with $\emptyset \mid \mathcal{A} \vdash_{\text{P}} wl'_1 : B'_1|\mathcal{C}'_1$, which trivially holds if $B_1|\mathcal{C}_1 = B'_1|\mathcal{C}'_1$.

- E-Call is a rule of the form $E_w\,[wl_1] \rightsquigarrow_{\text{P}} E_w\,[wl'_1]$ where $wl_1 = \overline{D^v} \fatsemi \texttt{(call}\ r\ \overline{v}\texttt{)}$ and $wl'_1 = t[\overline{x}/\overline{v}]$ with t the body of $\texttt{(abst}\ r\ \ldots\texttt{)}$. We know that $\emptyset \mid \emptyset \vdash_{\text{P}} \overline{D^v} : B_v|\emptyset$ (by TP-Buf). Because the $E_w\,[wl_1]$ is well-typed, wl_1 is well-typed (by TP-List), the part corresponding to the $\texttt{(call}\ \ldots\texttt{)}$ above must have been typed as: $\emptyset \mid \emptyset \vdash_{\text{P}} \texttt{(call}\ r\ v_1\ \ldots\ v_n\texttt{)} : T|\emptyset$ by TP-Call. Combined by TP-List we get that $\emptyset \mid \emptyset \vdash_{\text{P}} wl_1 : B_1|\mathcal{C}_1$ with $B_1|\mathcal{C}_1 = \mathsf{join}\,(B_v|\emptyset, T)$.

  TP-Call also gives us that $\mathcal{A}$ must have contained a $\texttt{(abst}\ r\ (S_1, \ldots, S_n) \to T\ x_1\ \ldots\ x_n\ t\texttt{)}$ and that $\forall i.\Gamma \vdash_{\text{E}} v_i : S_i$. From TP-main we know that the chosen abstraction in $\mathcal{A}$ must have been well-typed by TT-Abst and thus that $(x_1 \mapsto S_1) : \ldots : (x_m \mapsto S_m) \mid \emptyset \vdash_{\text{P}} t : T|\emptyset$. By lemma 14 we get $\emptyset \mid \emptyset \vdash_{\text{P}} t[\overline{x}/\overline{v}] : T|\emptyset$ and thus also $\Gamma \mid \mathcal{A} \vdash_{\text{P}} t[\overline{x}/\overline{v}] : T|\emptyset$. We conclude that the the E-Call rule replaces the $\texttt{(call}\ \ldots\texttt{)}$ construct by a $t[\overline{x}/\overline{v}]$ of identical type. Hence $B'_1|\mathcal{C}'_1 = \mathsf{join}\,(B_v|\emptyset, T) = B_1|\mathcal{C}_1$.

- E-CallRedr is a rule of the form $E_w\,[wl_1] \rightsquigarrow_{\text{P}} E_w\,[wl'_1]$ where $wl_1 = (D_i^v) \fatsemi \texttt{(redr}\ r\ T\ x_d\ x_a\ v_0\ e_b\texttt{)}$ and $wl'_1 = (D_o^v) \fatsemi \texttt{(redr}\ r\ T\ x_d\ x_a\ e_b[x_d, x_a/v_{i,0}, e_0]\ e_b\texttt{)}$ in which the value buffer D_o^v is D_i^v without its first element $v_{i,0}$.

  Because wl_1 is well typed, we know that:

  - (D_i^v) must have been concretely typed by TP-Buf as $\emptyset \mid \emptyset \vdash_{\text{P}} (D_i^v) : (S_d^c[m])|\emptyset$
  - The $\texttt{(redr}\ \ldots\texttt{)}$ must have been typed by TT-Redr as $\emptyset \mid \emptyset \vdash_{\text{P}} \texttt{(redr}\ r\ T\ x_d\ x_a\ v_0\ e_b\texttt{)} : (S_d[n]) \twoheadrightarrow (S_a[1])|\emptyset$. From the premises of TT-Redr we also know that $\vdash_{\text{E}} v_0 : S_a$ and $(x_a \mapsto S_a) : (x_d \mapsto S_d) \vdash_{\text{E}} e_b : S_a$.
  - From TP-List we get $\emptyset \mid \emptyset \vdash_{\text{P}} wl_1 : B_1|\mathcal{C}_1$ with $B_1|\mathcal{C}_1 = \mathsf{join}\,((S_d^c[m])|\emptyset, (S_d[n]) \twoheadrightarrow (S_a[1]))$. Since v_0 is a concrete value and $\vdash_{\text{E}} v_0 : S_a$, the type S_a must also be concrete, so $\sigma(S_a) = S_a$ for any unifier including $\sigma := \mathsf{unify}(S_d^c[m], S_d[n])$. So we know that $B_1|\mathcal{C}_1 = S_a|\emptyset$

  Because $(x_a \mapsto S_a) : (x_d \mapsto S_d) \vdash_{\text{E}} e_b : S_a$ it must also hold that $(x_a \mapsto \sigma(S_a)) : (x_d \mapsto \sigma(S_d)) \vdash_{\text{E}} e_b : \sigma(S_a)$. And as $\sigma(S_a) = S_a$, we may state that: $(x_a \mapsto S_a) : (x_d \mapsto \sigma(S_d)) \vdash_{\text{E}} e_b : S_a$. Combined with lemma 13

we get $\vdash_{\text{E}} e_b[x_d, x_a/v_{i,0}, v_0] : S_a$ because $\vdash_{\text{E}} v_0 : S_a$ and $\vdash_{\text{E}} v_{i,0} : \sigma(S_d) = S_d^c$. As a consequence the type of the `(redr ...)` does not change.

The shape of the elements in the first buffer does not change, but the size decreases with one. As $\emptyset \mid \emptyset \vdash_{\text{P}} (D_i^v) : (S_d^c[m])|\emptyset$, we may write that $\emptyset \mid \emptyset \vdash_{\text{P}} (D_0^v) : (S_d^c[m'])|\emptyset$ with $m' = m - 1$. Note that E-CallRedr only applies if m is strictly positive.

The type of wl_1' is derived by TP-List, the resulting type is $\mathsf{join}\,((S_d^c[m'])|\emptyset\,, (S_d[n]) \twoheadrightarrow (S_a[1])) = S_a|\emptyset = B_1|\mathcal{C}_1$. The join must succeed because n was fresh in TT-Redr and thus unifies effortlessly with m and m'.

We conclude that $B_1|\mathcal{C}_1 = B_1'|\mathcal{C}_1'$ and have hence proven preservation for this case.

- E-CallRedr0: is a rule of the form $E_w\,[wl_1] \rightsquigarrow_{\text{P}} E_w\,[wl_1']$ where $wl_1 = (\texttt{(buf}_{S_d^c[k]}\ \texttt{)})\fatsemi\texttt{(redr}\ r\ T\ x_d\ x_a\ v_0\ e_b\texttt{)}$ and $wl_1' = \texttt{(buf}_{S_a[1]}\ v_0\texttt{)}$ with $k \leq 0$.

  Analogous to the E-CallRedr case we get: $B_1|\mathcal{C}_1 = \mathsf{join}\,((S_d^c[k])|\emptyset\,, (S_d[n]) \twoheadrightarrow (S_a[1])) = S_a[1]|\emptyset$ and $\vdash_{\text{E}} v_0 : S_a$.

  From TP-Buf we get that $wl_1' = \texttt{(buf}_{S_a[1]}\ v_0\texttt{)}$ is typed $S_a[1]|\emptyset = B_1'|\mathcal{C}_1' = B_1|\mathcal{C}_1$.

  We conclude that $B_1|\mathcal{C}_1 = B_1'|\mathcal{C}_1'$ and have hence proven preservation for this case.

- E-CallShpr is a rule of the form $E_w\,[wl_1] \rightsquigarrow_{\text{P}} E_w\,[wl_1']$ where $wl_1 = (D_1^v, \ldots, D_m^v)\fatsemi\texttt{(shpr}\ r\ T\ x_i\ (x_{v,1}\ \ldots\ x_{v,m})\ e\texttt{)}$ and $wl_1' = \texttt{(shpr*}\ (\texttt{(buf}_{S_o[k]}\ \ldots\texttt{))}\ \mathcal{M}_s\texttt{)}$ with k the length of the output buffer $S_o^c[k] = \mathsf{join}\,(\overline{B_i^v}|\emptyset\,, T)$ and $\mathcal{M}_s = (x_{v,1} : D_{i,1}^v) : \ldots : (x_{v,m} : D_{i,m}^v)$. We will show that the types of wl_1 and wl_1' are identical.

  Let B_i^c be the type of input buffer D_i^v ($\emptyset \mid \emptyset \vdash_{\text{P}} (D_1^v, \ldots, D_m^v) : (B_1^c, \ldots, B_m^c)|\emptyset$). The `(shpr ...)` in wl_1 must be well typed by TT-Shpr and we know that:

  - $T = (S_1[e_1], \ldots, S_m[e_m]) \twoheadrightarrow (S_o[e_o])$
  - $(x_i \mapsto \texttt{int}), (x_{v,1} \mapsto S_1[e_1]), \ldots, (x_{v,m} \mapsto S_m[e_m]), \Gamma \vdash_{\text{E}} e : S_o$
  - $\forall i. onlyFree(S_i)$

  The full work list wl_1 will have been typed $\mathsf{join}\,((B_1^c, \ldots, B_m^c), (S_1[e_1], \ldots, S_m[e_m]) \twoheadrightarrow (S_o[e_o])) = B_1|\mathcal{C}_1$ by TP-List. Let σ be the unifier $\mathsf{unify}((B_1^c, \ldots, B_m^c), (S_1[e_1], \ldots, S_m[e_m]))$ used in join. From the premise of E-CallShpr we know that: $B_1|\mathcal{C}_1 = S_o^c[k]|\emptyset$.

  We must show that $wl_1' = \texttt{(shpr*}\ (\texttt{(buf}_{S_o[k]}\ e[x_i/0], \ldots, e[x_i/(k-1)]\texttt{)})\ \mathcal{M}_s\texttt{)}$ is typed $S_o^c[k]|\emptyset$. Looking at TT-BufferShpr we can refine our goal to $\Gamma_s \vdash_{\text{B}} (\texttt{(buf}_{S_o[k]}\ e[x_i/0], \ldots, e[x_i/(k-1)]\texttt{)}) : S_o^c[k]$ with $\Gamma_s = (x_{v,1} \mapsto B_1^c, \ldots, x_{v,m} \mapsto B_m^c)$. If $k > 0$, TP-Buf indeed validates that there are k values in the buffer $(e[x_i/0], \ldots, e[x_i/(k-1)])$. Otherwise, TP-Buf requires that the buffer is written $\texttt{(buf}_{S_o[k]}\texttt{)}$, which is the case because $e[x_i/0], \ldots, e[x_i/(k-1)]$ is empty list in this case. What remains it to be shown is that $\Gamma_s \vdash_{\text{E}} e[x_i/num] : S_o$ for any number num.

  From TT-Shpr before the application, we know that $(x_i \mapsto \texttt{int}), (x_{v,1} \mapsto S_1[e_1]), \ldots, (x_{v,m} \mapsto S_m[e_m]) \vdash_{\text{E}} e : S_o$. To which we may apply the unifier σ to get:

$$
\begin{array}{lccccc}
 & (x_i \mapsto \texttt{int}), (x_{v,1} \mapsto & S_1[e_1] &), \ldots, (x_{v,m} \mapsto & S_m[e_m] &) \vdash_{\text{E}} e : S_o \\
\iff & (x_i \mapsto \texttt{int}), (x_{v,1} \mapsto & \sigma(S_1[e_1]) &), \ldots, (x_{v,m} \mapsto & \sigma(S_m[e_m]) &) \vdash_{\text{E}} e : \sigma(S_o) \\
\iff & (x_i \mapsto \texttt{int}), (x_{v,1} \mapsto & B_1^c &), \ldots, (x_{v,m} \mapsto & B_m^c &) \vdash_{\text{E}} e : S_o^c \\
\iff & (x_i \mapsto \texttt{int}), & & \Gamma_s & & \vdash_{\text{E}} e : S_o^c
\end{array}
$$

  Now we use lemma 13 to get our remaining goal: $\Gamma_s \vdash_{\text{E}} e[x_i/num] : S_o^c$.

- E-EndShpr is a rule of the form $E_w\,[wl_1] \rightsquigarrow_{\text{P}} E_w\,[wl_1']$ where $wl_1 = \texttt{(shpr*}\ (D^v)\ \mathcal{M}_s\texttt{)}$ and $wl_1' = (D^v)$. By TT-BuffferShpr we know that $\emptyset \mid \emptyset \vdash_{\text{P}} \texttt{(shpr*}\ (D^v)\ \mathcal{M}_s\texttt{)} : (B_s)|\emptyset$ iff $\Gamma_s \vdash D^v : B_s$. But because B_s is the type of a concrete buffer, it does not contain any variables. We may use lemma 13 to get $\emptyset \vdash D^v : B_s$. TP-Buf now gives us $\emptyset \mid \emptyset \vdash_{\text{P}} (D^v) : B_s|\emptyset$. The type is preserved.

- E-INDEX is a rule of the form $E_w\,[wl_1] \rightsquigarrow_{\mathrm{P}} E_w\,[wl_1']$ where $wl_1 = \texttt{(shpr*}\ E_B[E[\texttt{(index}\ x_1\ num\texttt{)}]]\ \mathcal{M}_s\texttt{)}$ and $wl_1' = \texttt{(shpr*}\ E_B[E[v_{num}]]\ \mathcal{M}_s\texttt{)}$, for some v_{num} an element of the buffer $\mathcal{M}_s[x_1]$.

  TT-BUFFERSHPR specifies that the type of a `(shpr* ...)` is the type of its body ($E_B[E[\texttt{(index}\ x_1\ num\texttt{)}]]$) typed under the environment Γ_s derived from $\mathcal{M}_s$. The TE-INDEX rule specifies that $\Gamma_s \vdash_{\mathrm{E}} \texttt{(index}\ x_1\ num\texttt{)} : S$ if Γ_s contains an entry for x with a type of the form $S[e_l]$ with e_l an affine function in one free variable. Such a type can only be added to the environment by two rules: TT-SHPR and TM-BUFFERSHPR (the other rules only add non-buffer types). Since a `(shpr ...)` and a `(shpr* ...)` are not nestable (definition of w), the shape of wl learns us that the type for x in Γ_s must have originated from the TM-BUFFERSHPR rule. Hence, Γ_s must contain a mapping $x : S[e_l]$. This can only be the case if $\mathcal{M}_s$ contain a buffer for x of the form $\texttt{(buf}_{S[num]}\ \overline{v}\texttt{)}$ where each v is of type S (TM-STORETYPE). The v_{num} selected by E-INDEX is therefore of type S: $\Gamma_s \vdash_{\mathrm{E}} v_{num} : S$.

  To summarize, the `(index` x n`)` of type S is replaced by a value of type S. The rest of the type derivation remains identical, and we can conclude that the type and constrains of wl_1 and wl_1' are identical.

- E-OUTOFBOUNDS: Same argument as for E-INDEX but now we replace the value with a zero of shape S. Note that the buffer is tagged with the expected type, this ensures that the rule also works if the selected buffer in $\mathcal{M}_s$ is empty because it has a size ≤ 0.

- E-RETRIVE: We cannot take the same approach as we did for the previous steps. As opposed the the previous rules, the type of $wl_1 = \texttt{(retun ...)}$ and $wl_1' = \overline{\texttt{(buf ...)}}$ are not the same. Instead, we will show that $\mathsf{validate}(\mathcal{M}, B_{wl}|\mathcal{C}_{wl}) = B = \mathsf{validate}(\mathcal{M}, B'_{wl}|\mathcal{C}'_{wl})$ with $B'_{wl}|\mathcal{C}'_{wl}$ the type of wl', the work list after taking the step. Before the step, the type of the `(retrive` $x_1 \ldots x_n$`)` is $(B_{x_1}, \ldots, B_{x_n})\big|\{x'_1 \mapsto B_{x'_1}, \ldots, x'_n \mapsto B_{x'_m}\}$, with all $B_{x'_i}$ fresh and $\{x'_1, \ldots, x'_m\} = \bigcup_{i=1}^{m} \{x_{\min\{j \mid x_j = x_i\}}\}$. After E-RETRIVE, the type is concrete: $(B^c_1, \ldots, B^c_n)|\emptyset$.

  There are only m unique values for x_i namely $\{x'_1, \ldots, x'_m\}$. As such, there must exist an injection g such that $\forall i \in \{1, \ldots, n\}.x_i \equiv x'_{g(i)}$. This injection also works for the buffers: $\forall i \in \{1, \ldots, n\}.B_{x_i} \equiv B_{x'_{g(i)}}$. Let us look at a unifier σ that transforms $(B_{x_1}, \ldots, B_{x_n})$ into $(B^c_1, \ldots, B^c_n)$. This unifier also transforms $(B_{x'_{g(1)}}, \ldots, B_{x'_{g(n)}})$ into $(B^c_1, \ldots, B^c_n)$.

  One such unifier is: $\sigma := \mathsf{unify}\Big(\mathcal{M}\big|_{x'_1,\ldots,x'_m}, \{x'_1 \mapsto B_{x'_1}, \ldots, x'_m \mapsto B_{x'_m}\}\Big)$ as E-RETRIVE selects these buffers. σ is compatible with $\{x'_1 \mapsto B_{x'_1}, \ldots, x'_m \mapsto B_{x'_m}\}$ and $\mathcal{M}$ by definition. We may hence use corollary 11 as we do below:

$$
\begin{array}{rcl}
B & = & \mathsf{validate}^{\star}(\mathcal{M}, B_{wl}|\mathcal{C}_{wl}) \\
 & = & \mathsf{validate}^{\star}(\mathcal{M}, \overbrace{(B_{x_1}, \ldots, B_{x_n})\big|\{x'_1 \mapsto B_{x'_1}, \ldots x'_m \mapsto B'_{x_m}\}}^{\text{type of } \texttt{(retrive } x_1, \ldots x_n\texttt{)}}, \overbrace{T_2, \ldots, T_n}^{\text{type of } wl_2, \ldots, wl_n}) \\
 & \Updownarrow & \text{corollary 11} \\
 & = & \mathsf{validate}^{\star}(\mathcal{M}, \sigma((B_{x_1}, \ldots, B_{x_n}))|\emptyset, T_2, \ldots, T_n) \\
 & = & \mathsf{validate}^{\star}(\mathcal{M}, (B^c_1, \ldots, B^c_n)|\emptyset, T_2, \ldots, T_n) \\
 & = & \mathsf{validate}^{\star}(\mathcal{M}, B'_{wl}|\mathcal{C}'_{wl}) = B
\end{array}
$$

  And thus we have proven this case.

- E-LETBEND is a rule of the form $E_w\,[wl_1] \rightsquigarrow_{\mathrm{P}} E_w\,[wl_1']$ where $wl_1 = \texttt{(letB}\ x\ \texttt{(buf}_{S[num]}\ \overline{v}\texttt{)}\ \overline{D_1}\texttt{)}$ and $wl_1' = \overline{D_1}$. We immediately see that $B_1|\mathcal{C}_1 = B_2|\emptyset = B'_1|\mathcal{C}'_1$ in the type derivation tree below.

$$\dfrac{\boxed{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} \overline{D_1^v} : \overline{B_2}|\emptyset} \qquad \dfrac{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} (\texttt{buf}_{S[num]}\ \overline{v}) : B_1|\emptyset \qquad \emptyset = \sigma = \mathsf{unify}\,(\emptyset \cup \{x \mapsto B_1\}, \emptyset) \qquad \overline{\overline{B_2}|\emptyset = \sigma(\overline{B_2}|\emptyset) = \mathsf{japply}\,(x \mapsto B_1, \emptyset, \overline{B_2}, \emptyset)}}{\boxed{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} (\texttt{letB}\ x\ (\texttt{buf}_{S[num]}\ \overline{v})\ \overline{D_1^v}) : \overline{B_2}|\emptyset}}\ \text{TT-JAPPLY}}{}\ \text{TP-LETB}$$

- E-LETB1: Looking at the rule, we know that wl must have been of the form $(\texttt{letB}\ x\ wl_1\ wl_L)\ ;\!;\ wl_2\ ;\!;\ldots;\!;\ wl_n$. Because the program is well typed, we know that the $(\texttt{let}\ \ldots)$ must have been typed $\mathsf{japply}\,(x \mapsto B_1, \mathcal{C}_1, \overline{B_L}, C_L)$ by TP-LETB with $\emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl_1 : (B_1)|\mathcal{C}_1$ and $\emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl_L : \overline{B_L}|C_L$. The full program will be typed B by:

$$\begin{array}{rcll} B & = & \mathsf{validate}^\star(\mathcal{M}, \overbrace{\mathsf{japply}\,(x \mapsto B_1, \mathcal{C}_1, \overline{B_L}, C_L)}^{\text{type of } (\texttt{letB}\ \ldots)}, & \overbrace{T_2, \ldots, T_n}^{\text{type of } wl_2;\!;\ldots;\!;wl_n}\ ) \\ & = & \mathsf{validate}^\star(\mathcal{M}, \sigma_j(\overline{B_L}|\mathcal{C}_1 \cup (C_L \setminus x)), & T_2, \ldots, T_n\ ) \end{array}$$

Because the above $\mathsf{validate}^\star$ succeeds and $FV(B_1) \subseteq FV(\mathcal{C}_1)$ (lemma 2), we may derive by Lemma 10 that $\mathsf{validate}(\mathcal{M},\, B_1|\mathcal{C}_1\,)$ has a value. And we may use TP-MAIN to get that $(\texttt{main}\ (\texttt{prog}\ \mathcal{A}\ wl_1)\ \mathcal{M})$ is well typed. So we may use the IH on the premise of E-LETB1:
$(\texttt{main}\ (\texttt{prog}\ \mathcal{A}\ wl_1)\ \mathcal{M}) \rightsquigarrow_{\mathrm{P}} (\texttt{main}\ (\texttt{prog}\ \mathcal{A}\ wl_1')\ \mathcal{M})$. We obtain, that there exists a concrete B_1^c such that $\mathsf{validate}(\mathcal{M},\, B_1|\mathcal{C}_1\,) = \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)(B_1) = B_1^c = \mathsf{unify}(\mathcal{M}, \mathcal{C}_1')(B_1') = \mathsf{validate}(\mathcal{M},\, B_1'|\mathcal{C}_1'\,)$, with $\emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl_1' : (B_1')|\mathcal{C}_1'$.

Let $\sigma_j := \mathsf{unify}\,(\mathcal{C}_1 \cup \{x \mapsto B_1\}, \mathcal{C}_L)$ be the unifier used in japply with $x \mapsto B_1$. Let $\sigma_j' := \mathsf{unify}\,(\mathcal{C}_1 \cup \{x \mapsto B_1'\}, \mathcal{C}_L)$ be the unifier used in japply with $x \mapsto B_1'$. We may derive:

$$\begin{array}{rcll} B & = & \mathsf{validate}^\star(\mathcal{M}, \overbrace{\mathsf{japply}\,(x \mapsto B_1, \mathcal{C}_1, \overline{B_L}, C_L)}^{\text{type of } (\texttt{letB}\ wl_1\ \ldots)}, & \overbrace{T_2, \ldots, T_n}^{\text{type of } wl_2,\ldots,wl_n}\ ) \\ & = & \mathsf{validate}^\star(\mathcal{M}, \sigma_j(\overline{B_L}|\mathcal{C}_1 \cup (C_L \setminus x)), T_2, \ldots, T_n) & \\ & \Updownarrow & \sigma_{\mathcal{M}} = \mathsf{unify}(\mathcal{M}, \sigma_j(\mathcal{C}_1 \cup (C_L \setminus x))) \text{ and corollary 12} & (FV(\sigma_{\mathcal{M}}(\sigma_j(B_L))) = \emptyset \text{ by lemma 2}) \\ B & = & \mathsf{validate}^\star(\mathcal{M},\, \sigma_{\mathcal{M}}\,(\sigma_j(\overline{B_L}))|\mathcal{M}\,, T_2, \ldots, T_n) & \\ & \Updownarrow & (\star) & \\ B & = & \mathsf{validate}^\star\big(\mathcal{M},\, \sigma'_{\mathcal{M}}\,\big(\sigma'_j(\overline{B_L})\big)\big|\mathcal{M}\,, T_2, \ldots, T_n\big) & \\ & \Updownarrow & \sigma'_{\mathcal{M}} = \mathsf{unify}(\mathcal{M}, \sigma'_j(\mathcal{C}_1' \cup (C_L \setminus x))) \text{ and corollary 12} & (FV(\sigma'_{\mathcal{M}}(\sigma_j(B_L'))) = \emptyset \text{ by lemma 2}) \\ B & = & \mathsf{validate}^\star(\mathcal{M}, \sigma'_j(\overline{B_L}|\mathcal{C}_1' \cup (C_L \setminus x)), T_2, \ldots, T_n) & \\ & = & \mathsf{validate}^\star(\mathcal{M}, \underbrace{\mathsf{japply}\,(x \mapsto B_1', \mathcal{C}_1', \overline{B_L}, C_L)}_{\text{type of } (\texttt{letB}\ wl_1'\ \ldots)}, & \underbrace{T_2, \ldots, T_n}_{\text{type of } wl_2,\ldots,wl_n}\ ) \end{array}$$

And hence the type remains B due to TP-MAIN.

$(\star)$ .We prove that $\sigma_{\mathcal{M}}(\sigma_j(\overline{B_L})) = \sigma_{\mathcal{M}}(\sigma'_j(\overline{B_L}))$. Take an arbitrary free variable $X \in FV(\overline{B_L})$, it is sufficient to show that: $\sigma_{\mathcal{M}}(\sigma_j(X)) = \sigma_{\mathcal{M}}(\sigma'_j(X))$. Because of lemma 2 we know that $X \in FV(\mathcal{C}_L)$. There are two possibilities:

- $X \in FV(\mathcal{C}_L[x])$: In this case we know that $\sigma_j(\mathcal{C}_L[x]) = B_1$ and $\sigma'_j(\mathcal{C}_L[x]) = B_1'$. We also already know that $\mathsf{unify}(\mathcal{M}, \mathcal{C}_1)(B_1) = B_1^c = \mathsf{unify}(\mathcal{M}, \mathcal{C}_1')(B_1')$

By construction of $\sigma_{\mathcal{M}}$, we get $\sigma_{\mathcal{M}}(\sigma_j(\mathcal{C}_1)) = \mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)} = \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)\,(\mathcal{C}_1)$. So the effect of $\sigma_{\mathcal{M}} \circ \sigma_j$ on the free variables of $\mathcal{C}_1$ is identical to the effect of $\mathsf{unify}(\mathcal{M}, \mathcal{C}_1)$. And because $FV(B_1) \subseteq FV(\mathcal{C}_1)$ (lemma 2), the effect is also identical for B_1. The same result holds with primes: $\sigma'_{\mathcal{M}}(\sigma'_j(B'_1)) = \mathsf{unify}(\mathcal{M}, \mathcal{C}'_1)\,(B'_1)$. We may thus write:

$$\begin{aligned}
\sigma_{\mathcal{M}}(\sigma_j(\mathcal{C}_L[x])) &= \sigma_{\mathcal{M}}(\sigma_j(B_1)) \\
&= \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)\,(B_1) \\
&= B_1^c \\
&= \mathsf{unify}(\mathcal{M}, \mathcal{C}'_1)\,(B'_1) \\
&= \sigma'_{\mathcal{M}}(\sigma'_j(B'_1)) \qquad = \sigma'_{\mathcal{M}}(\sigma'_j(\mathcal{C}_L[x]))
\end{aligned}$$

If $\sigma_{\mathcal{M}}(\sigma_j(X)) \neq \sigma'_{\mathcal{M}}(\sigma'_j(X))$ then $\sigma_{\mathcal{M}}(\sigma_j(\mathcal{C}_L[x])) \neq \sigma'_{\mathcal{M}}(\sigma'_j(\mathcal{C}_L[x]))$. Which contradicts the above, therefore it must be the case that $\sigma_{\mathcal{M}}(\sigma_j(X)) = \sigma'_{\mathcal{M}}(\sigma'_j(X))$, our goal is true.

– $X \notin FV(\mathcal{C}_L[x]) \Rightarrow X \in FV(\mathcal{C}_L) \setminus FV(\mathcal{C}_L[x])$. Therefore, it must hold that $\exists k \neq x.X \in FV(\mathcal{C}_L[k])$. Take this k and we may use the definition of $\sigma_{\mathcal{M}}$ and $\sigma'_{\mathcal{M}}$ to write the next equality of concrete types.

$$\sigma_{\mathcal{M}}\left(\sigma_j(\mathcal{C}_L[k])\right) = \mathcal{M}[k] = \sigma'_{\mathcal{M}}\left(\sigma'_j(\mathcal{C}_L[k])\right)$$

This can only hold if the following equality holds for any free variable of $\mathcal{C}_L[k]$, in particular X.

$$\sigma_{\mathcal{M}}\left(\sigma_j(X)\right) = \sigma'_{\mathcal{M}}\left(\sigma'_j(X)\right)$$

- E-LetBBuf: Looking at the rule, we know that wl must have been of the form $\texttt{(letB}\ x\ (D^v)\ wl_L\texttt{)} \fatsemi wl_2 \fatsemi \ldots \fatsemi wl_n$. Because the program is well typed, we know that the $\texttt{(let ...)}$ must have been typed $\mathsf{japply}\,(x \mapsto B_1^c, \emptyset, \overline{B_L}, C_L)$ by TP-LetB with $\emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} D^v : (B_1^c)|\emptyset$ and $\emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl_L : \overline{B_L}|C_L$. The full program will be typed B by:

$$\begin{aligned}
B &= \mathsf{validate}^\star(\mathcal{M}, \overbrace{\mathsf{japply}\,(x \mapsto B_1^c, \emptyset, \overline{B_L}, C_L)}^{\text{type of } \texttt{(letB ...}\, wl_L\texttt{)}}, \overbrace{T_2, \ldots, T_n}^{\text{type of } wl_2,\ldots,wl_n}) \\
&= \mathsf{validate}^\star(\mathcal{M}, \sigma(\overline{B_L}|C_L \setminus x), T_2, \ldots, T_n)
\end{aligned}$$

For $\sigma = \mathsf{unify}(\emptyset \cup \{x \mapsto B_1^c\}, \mathcal{C}_L) = \mathsf{unify}(B_1^c, \mathcal{C}_L[x])$ which ensures that all free variables in the type corresponding to x in $\mathcal{C}_L$ are replaced by the concrete values of B_1^c (which is concrete because it corresponds to a concrete buffer).

Let us look at $\texttt{(main (prog}\ \mathcal{A}\ wl_L\texttt{)}\ (x \mapsto D^v) : \mathcal{M}\texttt{)}$. Lemma 10 gives us that $\mathsf{validate}(\mathcal{M}, \sigma\,(\overline{B_L}|\mathcal{C}_L \setminus x))$ has a value from the fact that $\mathsf{validate}(\sigma(\overline{B_L}|C_L \setminus x), T_2, \ldots, T_n)$ has a value. We may also derive that a B' exists such that

$$\begin{array}{rl}
B' = & \mathsf{validate}((x \mapsto D^v) : \mathcal{M}, \sigma\,(\overline{B_L}|\mathcal{C}_L \setminus x) \cup \{x \mapsto B_1^c\}) \\
\Updownarrow & \sigma(\mathcal{C}_L[x]) = B_1^c \\
B' = & \mathsf{validate}((x \mapsto D^v) : \mathcal{M}, \sigma\,(\overline{B_L}|\mathcal{C}_L))
\end{array}$$

To wich we may apply lemma 10 again to get that $\mathsf{validate}((x \mapsto D^v) : \mathcal{M}, \overline{B_L}|\mathcal{C}_L)$ has a value. Combining this with $FV(B_L) \subseteq FV(\mathcal{C}_L)$ (lemma 2) and $\mathsf{dom}(\mathcal{C}_L) \subseteq \mathsf{dom}((x \mapsto D^v) : \mathcal{M})$ (because of the existence of the aforementioned validate), we may apply the IH to the premise of E-LetBBuf:

$$\texttt{(main (prog}\ \mathcal{A}\ wl_L\texttt{)}\ (x \mapsto D^v) : \mathcal{M}\texttt{)} \rightsquigarrow_{\mathrm{P}} \texttt{(main (prog}\ \mathcal{A}\ wl'_L\texttt{)}\ (x \mapsto D^v) : \mathcal{M}\texttt{)}$$

. Let $\emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl'_L : \overline{B'_L}|C_L$, we get that there exist a concrete B' such that:

$$\mathsf{validate}((x \mapsto B_1^c) : \mathcal{M}, B_L|\mathcal{C}_L) = \mathsf{validate}((x \mapsto B_1^c) : \mathcal{M}, B'_L|\mathcal{C}'_L) = B'$$

Formulated differently, we have:

$$\mathsf{unify}((x \mapsto B_1^c) : \mathcal{M}, \mathcal{C}_L)(B_L) = \mathsf{unify}((x \mapsto B_1^c) : \mathcal{M}, \mathcal{C}'_L)(B'_L) = B'$$

Now, let $\sigma' = \mathsf{unify}((x \mapsto B_1^c) : \mathcal{M}, \mathcal{C}'_L)\big|_{FV(\mathcal{C}'_L[x])} = \mathsf{unify}(B_1^c, \mathcal{C}'_L[x])$ and note that $\sigma'(\mathcal{C}'_L[x]) = B_1^c$ just as $\sigma(\mathcal{C}_L[x]) = B_1^c$. We may split up $\mathsf{unify}(\mathcal{M}, \mathcal{C}_L)$ into σ (or σ') and the remaining unifier as follows:

$$\begin{aligned}
\mathsf{unify}((x \mapsto B_1^c) : \mathcal{M}, \mathcal{C}_L))(B_L) &= \mathsf{unify}((x \mapsto B_1^c) : \mathcal{M}, \mathcal{C}'_L)(B'_L) &&= B' \\
&\Downarrow \text{split up both sides} \\
\mathsf{unify}((x \mapsto B_1^c) : \mathcal{M}, \sigma(\mathcal{C}_L))(\sigma(B_L)) &= \mathsf{unify}((x \mapsto B_1^c) : \mathcal{M}, \sigma'(\mathcal{C}'_L))(\sigma'(B'_L)) &&= B' \\
&\Downarrow FV(\sigma(C_L[x])) = FV(\sigma'(C'_L[x])) = \emptyset \\
\mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_L \setminus x))(\sigma(B_L)) &= \mathsf{unify}(\mathcal{M}, \sigma'(\mathcal{C}'_L \setminus x))(\sigma'(B'_L)) &&= B'
\end{aligned}$$

With this equality we can now prove that the type is preserved. The final step uses corollary 12 to introduce the unifier $\mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_L \setminus x))$ in the first line and with the unifier $\mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}'_L \setminus x))$ on the fouth line:

$$\begin{aligned}
B &\overset{}{=} \mathsf{validate}^\star(\mathcal{M}, \sigma(\overline{B_L}|\mathcal{C}_L \setminus x), T_2, \ldots, T_n) \\
&\overset{\text{corollary 12}}{=} \mathsf{validate}^\star(\mathsf{unify}(\mathcal{M}, \sigma(\mathcal{C}_L \setminus x))\,(\sigma(B_L))|\mathcal{M}\,, T_2, \ldots, T_n) \\
&\overset{\text{prev result}}{=} \mathsf{validate}^\star(B'|\mathcal{M}\,, T_2, \ldots, T_n) \\
&\overset{\text{prev result}}{=} \mathsf{validate}^\star(\mathsf{unify}(\mathcal{M}, \sigma'(\mathcal{C}'_L \setminus x))\,(\sigma(B'_L))|\mathcal{M}\,, T_2, \ldots, T_n) \\
&\overset{\text{corollary 12}}{=} \mathsf{validate}^\star(\sigma'(B'_L|\mathcal{C}'_L \setminus x), T_2, \ldots, T_n) \\
&= \mathsf{validate}^\star(\mathcal{M}, \underbrace{\mathsf{japply}\,(x \mapsto B_1^c, \emptyset, \overline{B'_L}, C'_L)}_{\text{type of } (\texttt{letB} \ldots wl'_L)}, \underbrace{T_2, \ldots, T_n}_{\text{type of } wl_2,\ldots,wl_n})
\end{aligned}$$

And hence this case is proven

- E-Other: Follows immediately from the preservation of scalar closed arithmetic expressions (see below). In case of the `(call ...)` for example, we had that E_R matched $(\overline{(\texttt{buf}_{S[num]}\ \overline{v})}\,;(\texttt{call}\ r\ (\overline{v}, \cdot, \overline{e}))$. By taking a step on the expression at the position of $\cdot$, the type remains identical. Note that the steps taken will not use any buffer type stored in Γ, these cases are handled by E-Index and E-IndexOutOfBounds. There are no reduction rules $\hookrightarrow_{\text{E}}$ that alter `(index ...)` constructs. The changed expressions will therefore be scalar closed.

□

Appendix B.7. Progress en Preservation of scalar-closed arithmetic expression

Definition 6 (Scalar-closed arithmetic expression)**.** *An expression e is said to be scalar closed if all its free variables come only from usages of `(index ...)` constructs.*

Lemma 15 (Progress of scalar-closed arithmetic expressions)**.** *Any scalar-closed expression e such that $\Gamma \vdash_{\text{E}} e : S$ for some Γ with $\mathsf{range}(\Gamma)$ buffer types. Is either*

- *a value*
- *there exists an e' such that $e \hookrightarrow_{\text{E}} e'$*
- *there exists an E such that $e = E[\texttt{(index}\ x\ v\texttt{)}]$*

Proof. By induction on the typing derivation and case analysis on the last applied rule to type e. (The typing rules can be found in Appendix A.3 on page 42)

- TE-Int and TE-Float: e is a value.

- TE-Var: Not possible Γ only contains buffer types, nu shaped S
- TE-BinOp and TE-BinComp: The IH gives us that either
  - e_1 is a value, if e_2 is a value as well, one of the binary operation reduction rules applies. If not, by IH, e_2 a step can be taken for e_2 and thus also for e by E-Arith. If no such step can be taken, e_2 must have an E_2 such that it is E[`(index x v)`], in that case we can extend E_2 to ensure that $e = E$[`(index x v)`].
  - there exists an e_1' such that $e_1 \hookrightarrow_{\text{E}} e_1'$, in this case E-Arith applies, and a step can be taken.
  - there exists an E_1 such that $e_1 = E_1$[`(index x v)`], in this case, we may extend E_1 to build an E such that $e = E$[`(index x v)`].
- TE-Fst and TE-Snd: the IH gives us that either
  - e_1 is a value, which must have been of the form `(tuple` v_1 v_2`)` (by TE-Tuple), in which case $E - Fst$ or $E - Snd$ applies.
  - there exists an e_1' such that $e_1 \hookrightarrow_{\text{E}} e_1'$, in this case E-Arith applies, and a step can be taken.
  - there exists an E_1 such that $e_1 = E_1$[`(index x v)`], in this case, we may extend E_1 to build an E such that $e = E$[`(index x v)`].
- TE-Tuple: The IH gives us that either
  - e_1 is a value, if e_2 is a value as well, the tuple is a value. If not, by IH, e_2 a step can be taken for e_2 and thus also for e by E-Arith. If no such step can be taken, e_2 must have an E_2 such that it is E[`(index x v)`], in that case we can extend E_2 to ensure that $e = E$[`(index x v)`].
  - there exists an e_1' such that $e_1 \hookrightarrow_{\text{E}} e_1'$, in this case E-Arith applies, and a step can be taken.
  - there exists an E_1 such that $e_1 = E_1$[`(index x v)`], in this case, we may extend E_1 to build an E such that $e = E$[`(index x v)`].
- TE-If: The IH gives us that either
  - e_c is a value, which must be numeric by TE-Int. If this number is 0, E-If0 applies, otherwise E-If applies.
  - there exists an e_c' such that $e_c \hookrightarrow_{\text{E}} e_c'$, in this case E-Arith applies, and a step can be taken.
  - there exists an E_c such that $e_c = E_c$[`(index x v)`], in this case, we may extend E_c to build an E such that $e = E$[`(index x v)`].
- TE-Let: The IH gives us that either
  - e_v is a value and E-Let applies
  - there exists an e_1' such that $e_v \hookrightarrow_{\text{E}} e_v'$, in this case E-Arith applies, and a step can be taken.
  - there exists an E_v such that $e_v = E_v$[`(index x v)`], in this case, we may extend E_v to build an E such that $e = E$[`(index x v)`].

□

Corollary 16. *Any closed expression e such that $\emptyset \vdash_{\text{E}} e : S$ is either a value or there exists an e' such that $e \hookrightarrow_{\text{E}} e'$*

Proof. Such expression cannot contain a `(index x e)` construct because x must be in the environment (Γ) for such an expression to be well-typed. □

Lemma 17 (Preservation of arithmetic expressions)**.** *If* $\Gamma \vdash_{\mathrm{E}} e : S$ *and* $e \hookrightarrow_{\mathrm{E}} e'$*, then* $\Gamma \vdash_{\mathrm{E}} e' : S$ *for any* Γ*.*

Proof. By structural induction on e and case analysis of the reduction rules. Only two rules deserve special attention here, the other rules are straightforward. The preservation of E-Arith follows from the preservation of the reduction in the premise. For E-Let, we notice that the body of the let was initially typed under an extended environment that had a binding for x to S_1. TE-Let also gives us that the value of the binding is of type S_1. All occurrences of x replaced in e were previously typed as S_1 by TE-Var. The substitution changes this part of the typing derivation into an expression of the same type. The resulting type hence remains unchanged. □

Appendix B.8. Progress of Work Lists

Proof of lemma 4. To show that $\texttt{(main (prog } \mathcal{A}\ E_{wl}[wl]\texttt{)}\ \mathcal{M}\texttt{)} \rightsquigarrow_{\mathrm{P}} \texttt{(main (prog } \mathcal{A}\ E_{wl}[wl']\texttt{)}\ \mathcal{M}\texttt{)}$ for any E_{wl}, it suffices (and is sufficient) to show that this reduction stems from a rule of the from $E_w\,[wl] \rightsquigarrow_{\mathrm{P}} E_w\,[wl']$ with $E_w\,[\cdot] = \texttt{(main (prog } \mathcal{A}\ E_{wl}[\cdot]\texttt{)}\ \mathcal{M}\texttt{)}$. Luckily, all our reduction rules are of this form.

The proof will proceed by well founded induction on the derivation of $\emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl : B_T|\mathcal{C}_T$. We do case analysis on the last applied rule in the derivation of $\emptyset \mid \mathcal{A} \vdash_{\mathrm{P}} wl : B_T|\mathcal{C}_T$.

TP-Retrive $(\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} \texttt{(retrive } x_1\ \ldots\ x_n\texttt{)} : (B_{x_1}, \ldots, B_{x_n})\big|\{x'_1 \mapsto B_{x'_1}, \ldots, x'_m \mapsto B_{x'_m}\})$ with $\{x'_1, \ldots, x'_m\} = \bigcup_{i=1}^{m} \left\{x_{\min\{j \mid x_j = x_i\}}\right\}$
Because we know that $\exists \overline{B}.\overline{B} = \mathsf{validate}\big(\mathcal{M}^B, (B_{x_1}, \ldots, B_{x_n})\big|\{x_1 \mapsto B_{x'_1}, \ldots, x_m \mapsto B_{x'_m}\}\big)$, $\mathcal{M}^B$ must have the keys $x'_1, \ldots, x'_m$ in its domain (by TP-Validate). These keys are identical to the names $x_1, \ldots, x_n$ with duplicates removed. By definition $\mathcal{M}^E$ has the same domain, therefore E-Retrive can be applied.

TP-LetB $(\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} \texttt{(letB } x\ wl_1\ wl_L\texttt{)} : \overline{B_T}|\mathcal{C}_T$ with $\overline{B_T}|\mathcal{C}_T = \mathsf{japply}\,(x \mapsto B_1, \mathcal{C}_1, \overline{B_L}, \mathcal{C}_L)$ if $\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl_1 : (B_1)|\mathcal{C}_1$ and $\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl_L : \overline{B_L}|\mathcal{C}_L)$
There are three possibilities:

- wl_1 is not a value: From TT-JApply we know that $\mathcal{C}_T = \sigma(\mathcal{C}_1 \cup (\mathcal{C}_L \setminus x))$ for some σ. Combing TT-JApply with our premise gives us $\exists B.B = \mathsf{validate}(\mathcal{M},\, B_T|\mathcal{C}_T) = \mathsf{validate}\,(\mathcal{M}, \sigma\,(\overline{B_L}|\mathcal{C}_1 \cup (\mathcal{C}_L \setminus x)))$. This means that
  - there is a substitution that can be applied to $\sigma(\mathcal{C}_1)$ to make it equal to $\mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)}$. Therefore, there also exists a $\sigma' = \mathsf{unify}(\mathcal{M}, \mathcal{C}_1)$, and
  - $\mathsf{dom}(\mathcal{C}_1 \cup (\mathcal{C}_L \setminus x)) \subseteq \mathsf{dom}(\mathcal{M})$ and thus also $\mathsf{dom}\,(\mathcal{C}_1) \subseteq \mathsf{dom}\,(\mathcal{M})$.
  - By lemma 2 we know that $FV(B_1) \subseteq FV(\mathcal{C}_1)$

  The above three points are the premises of TP-Validate for the type of wl_1: we may derive that there exists a $B' = \mathsf{validate}(\mathcal{M}, B_1|\mathcal{C}_1)$. We can now use the IH to derive that wl'_1 exists such that $E_w\,[wl_1] \rightsquigarrow_{\mathrm{P}} E_w\,[wl'_1]$. This is the premise of E-LetB1, and hence the step E-LetB1 can be applied to our `(let ...)`.
- wl_1 is a value, wl_L is not a value: The canonical form of a value of type (B_1) is $(\texttt{(buf}_{B_1}\ \overline{v}\texttt{)})$ and we know form TP-Buf that its type is $B_1|\emptyset$. Combing this with TT-JApply, we get that $\mathcal{C}_T = \sigma(\emptyset \cup (\mathcal{C}_L \setminus x)) = \sigma(\mathcal{C}_L \setminus x)$ and $\sigma = \mathsf{unify}(\emptyset \cup \{x \mapsto B_1\}, \mathcal{C}_L)$. Therefore we have that $\sigma(\mathcal{C}_L[x]) = B_1$ (if $x \in \mathsf{dom}\,(\mathcal{C}_L)$).
  - Because $\exists B.B = \mathsf{validate}(\mathcal{M},\, B_T|\mathcal{C}_T) = \mathsf{validate}\,(\mathcal{M}, \sigma\,(\overline{B_L}|\mathcal{C}_L \setminus x))$ there is a substitution that can be applied to $\sigma(\mathcal{C}_L \setminus x)$ to make it $\mathcal{M}\big|_{\mathcal{C}_L \setminus x}$. Because $\sigma(\mathcal{C}_L[x])$ is the concrete type B_1 we may extend $\mathcal{M}$ with the value $(\texttt{(buf}_{B_1}\ \overline{v}\texttt{)})$ of type B_1 for x and obtain that there also exists a $\sigma' = \mathsf{unify}(x \mapsto (\texttt{(buf}_{B_1}\ \overline{v}\texttt{)}) : \mathcal{M}, \mathcal{C}_L)$. If $x \notin \mathcal{C}_L$, this still holds because TM-UnifyM only considers keys in the intersection of the domains.

– $\mathsf{dom}(\mathcal{C}_L \setminus x) \subseteq \mathsf{dom}(\mathcal{M})$ so $\mathsf{dom}(\mathcal{C}_L) \subseteq \mathsf{dom}(x \mapsto (\texttt{(buf}_{B_1}\ \overline{v}\texttt{)}) : \mathcal{M})$

– By lemma 2 we know that $FV(\overline{B_L}) \subseteq FV(\mathcal{C}_L)$

The above three points are the premises of TP-VALIDATE for the type of wl_L with the store $(x \mapsto (\texttt{(buf}_{B_1^v}\ \overline{v}\texttt{)}) : \mathcal{M})$: we may derive that there exists a $B' = \mathsf{validate}\big((x \mapsto (\texttt{(buf}_{B_1^v}\ \overline{v}\texttt{)}) : \mathcal{M}),\ B_L|\mathcal{C}_L\big)$. We can now use the IH to derive that wl'_L exists such that $E_w\,[wl_L] \rightsquigarrow_{\text{P}} E_w\,[wl'_L]$ with $E_w\,[\cdot] = \texttt{(main (prog}\ \mathcal{A}\ E_{wl}[\cdot]\texttt{)}\ (x \mapsto (\texttt{(buf}_{B_1^v}\ \overline{v}\texttt{)}) : \mathcal{M})\texttt{)}$. Therefore, E-LETBBUF applies.

- wl_1 and wl_L are values: E-LETBEND applies. Note that TP-LETB gives us that the type of wl_1 must be a single buffer.

TP-Buf the work list must be of the form $(\overline{\texttt{(buf}\ \overline{e}\texttt{)}})$. If all the elements of the all the buffers are values, wl is a value. If not, take the first such non-value and call it e. We have that $wl = E_{wl}[E_R[e]]$. From TP-BUF and TM-BUFFER we get that $\vdash_{\text{E}} e : S$ for some shape S. From corollary 16 we now get that there must be a e' such that $e \hookrightarrow_{\text{E}} e'$. This is the premise of E-OTHER so we may conclude that this rule ($E_w\,[E_R[e]] \rightsquigarrow_{\text{P}} E_w\,[E_R[e']]$) can be applied to wl.

TT-BufferShpr Either the D in $\texttt{(shpr*}\ D\ \mathcal{M}_s\texttt{)}$ is a value (E-ENDSHPR applies) or it is not. In the that case, E-OTHER applies because the D only consists of scalar-closed expression (TT-BUFFERSHPR and the case above)

TP-List $(\Gamma \mid \mathcal{A} \vdash_{\text{P}} wl_1 \fatsemi w_2 : \overline{B_T}|\mathcal{C}_T$ with $\overline{B_T}|\mathcal{C}_T = \mathsf{join}\,(B_1|\mathcal{C}_1\,, T_2)$ if $\Gamma \mid \mathcal{A} \vdash_{\text{P}} wl_1 : \overline{B_1}|\mathcal{C}_1$ and $\Gamma \mid \mathcal{A} \vdash_{\text{P}} w_2 : T_2|\emptyset\,)$
There are two possibilities:

- wl_1 is a value (typed by TP-BUF) and therefore has the form $\overline{\texttt{(buf}\ \overline{v}\texttt{)}}$, also written as $(D_1^v, \ldots, D_m^v)$.

  TP-Call , we have the work list $\overline{\texttt{(buf}_S\ \overline{v}\texttt{)}} \fatsemi \texttt{(call}\ r\ \overline{v_a}\texttt{)}$ with $\overline{v_a}$ values (if not values E-OTHER applies due to TP-CALL and corollary 16). We know the number of arguments passed to the call $|x_a| = |v_a|$ and that a definition is in $\mathcal{A}$ due to TP-CALL, therefore E-CALL applies.

  TT-Mapr we have the work list $(\texttt{(buf}_{S[num]}\ \overline{v}\texttt{)}) \fatsemi \texttt{(mapr}\ (S_1[n]) \twoheadrightarrow (S_2[n])\ x_i\ x_o\ e\texttt{)}$. We are certain that the first item of the list is a single buffer because join succeeds in TP-LIST and ensures the type of the first element of the list unifies with a list of length one (from TT-MAPR). Hence E-CALLMAPR applies.

  TT-Redr we have the work list $(\texttt{(buf}_{S[num]}\ \overline{v}\texttt{)}) \fatsemi \texttt{(redr}\ (S_d[n]) \twoheadrightarrow (S_a[1])\ x_d\ x_a\ e_0\ e_b\texttt{)}$. We know that there is just one buffer by the argument above for TT-MAPR. If e_0 is not a value, we know it a step can be taken with E-OTHER (TT-REDR and corollary 16). If it is a value, there are two options: if $|\overline{v}| = 0$, E-CALLREDR0 applies else ($|\overline{v}| > 0$) E-CALLREDR applies.

  TT-Shpr , we have the work list $(D_1^v, \ldots, D_m^v) \fatsemi \texttt{(shpr}\ (S_1[e_1], \ldots, S_m[e_m]) \twoheadrightarrow (S_o[e_o])\ x_i\ (x_{v1}\ \ldots\ x_{vm})\ e\texttt{)}$. From the successful application of TT-SHPR we derive that the $\texttt{(shpr}\ \ldots\texttt{)}$ construct is typed $(S_1[e_1], \ldots S_m[e_m]) \twoheadrightarrow S_o[e_o$ The successful application of join in TP-LIST tells us that there is a σ such that $\sigma = \mathsf{unify}((B_1, \ldots, B_m), (S_1[e_1], \ldots S_m[e_m]))$ with $\forall i.\ \vdash D_i^v; B_i$. From this, we know that the correct number of buffers are supplied and that $\mathcal{M}_s$ is well-defined. Since all buffers D_i^v have a concrete type, the unifier σ will replace all free variables in $S_o[e_0]$ by concrete values, as a concequence $k = \sigma(e_o)$ will be a concrete integral value and $k - 1$ exists.

- wl_1 is not a value. We know that $\exists \overline{B}.\overline{B} = \mathsf{validate}(\mathcal{M}, \overline{B_T}|\mathcal{C}_T) = \mathsf{validate}(\mathcal{M}, \sigma(\overline{B_3})|\sigma(\mathcal{C}_1))$ for some $\overline{B_3}$ and some σ created in TM-JOIN. From TP-VALIDATE, we now know that $\mathsf{unify}\Big(\mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)}, \sigma(\mathcal{C}_1)\Big)$ has a value. Therefore, there must also exist a $\sigma' = \mathsf{unify}\Big(\mathcal{M}\big|_{\mathsf{dom}(\mathcal{C}_1)}, \mathcal{C}_1\Big)$. We also know that $\mathsf{dom}(\mathcal{C}_T) = \mathsf{dom}(\sigma(\mathcal{C}_1)) = \mathsf{dom}(\mathcal{C}_1) \subseteq \mathsf{dom}(\mathcal{M})$ and by lemma 2 that $FV(B_1) \subseteq FV(\mathcal{C}_1)$. The previous three facts allow us to state that $\exists B'.B' = \mathsf{validate}(\mathcal{M}, B_1|\mathcal{C}_1)$. So, we may use the IH

and obtain that there exist a wl_1' such that
$\texttt{(main (prog } \mathcal{A}\ E_{wl}[wl_1]\texttt{) } \mathcal{M}\texttt{)} \rightsquigarrow_{\mathrm{P}} \texttt{(main (prog } \mathcal{A}\ E_{wl}[wl_1']\texttt{) } \mathcal{M}\texttt{)}$ for any E_{wl}. By substituting $E_{wl}[\cdot]$ for $E_{wl}[\cdot \fatsemi w_2]$ we get
$\texttt{(main (prog } \mathcal{A}\ E_{wl}[wl_1 \fatsemi w_2]\texttt{) } \mathcal{M}\texttt{)} \rightsquigarrow_{\mathrm{P}} \texttt{(main (prog } \mathcal{A}\ E_{wl}[wl_1' \fatsemi w_2]\texttt{) } \mathcal{M}\texttt{)}$ for any E_{wl}, our goal.

□

Appendix C. Omitted listings

```
1         let deg2rad = 0.01745329252
2         let from = positions._1  in
3         let to = positions._2  in
4         let dlon = (to._1 - from._1)*deg2rad  in
5         let flat = from._2*deg2rad in
6         let tlat = to._2*deg2rad
7         let dz = sin(to._2) - sin(from._2) in
8         let dx = cos(dlon) * cos(to._2) - cos(to._2) in
9         let dy = sin(dlon) * cos(to._2) in
10        asin(sqrt(dx*dx+dy*dy+dz*dz)/2)*2*6371
```

Listing 11: Haversine computation

Appendix D. Extensions

We added some extensions to our language to ease programming further. They are syntactical sugar for constructs in the core language as defined in the main body of this paper.

Appendix D.1. Iterate

To repeatedly apply an identical transformation we provide the $\texttt{(iterate } x\ e\ wl\texttt{)}$ construct also written as "`for x in 0..e wl`" to ease presentation. It allows to repeat a program wl for v times, with an iteration variable x which will run from 0 to $v-1$ with v the concrete value e reduces to.

The output of the previous iteration will be the input of the next iteration. We require that the output of an `(iterate ...)` construct has the same buffer type as its input. In other words: wl must have the type $(B_1[n]) \multimap B_1[n]$

This construct can be added as a form of syntactic sugar, which simply repeats the content of the construct the appropriate number of times. This rule used to be in the core semantics and is included in our implementation. The typing rule used for this construct can be found below.

First we add some constructs to the semantic entities.

$$\begin{aligned} t &::= \ \ldots \mid \texttt{(iterate } x\ e\ wl\texttt{)} \\ E_R &::= \ \ldots \mid \overline{D^v} \fatsemi \texttt{(iterate } x\ E\ wl\texttt{)} \end{aligned}$$

We must then also describe how substitution works for out new construct.

$$\texttt{(iterate } x\ e\ wl\texttt{)}[x/v] = \texttt{(iterate } x\ e[x/v]\ wl[x/v]\texttt{)}$$

Now we can write the typing and evaluation rules.

E-ITERATE

$$\frac{\forall 0 \leq i < \mathsf{max}(0, v).wl_i = wl_a[x/i]}{E_w\left[(D^v) \fatsemi \texttt{(iterate } x \ v \ wl_a\texttt{)}\right] \rightsquigarrow_{\mathrm{P}} E_w\left[(D^v) \fatsemi wl_0 \fatsemi \ldots \fatsemi wl_{\mathsf{max}(0,v)-1}\right]}$$

TP-ITERATE

$$\frac{\Gamma \vdash_{\mathrm{E}} e \texttt{ : int} \qquad x \mapsto \texttt{int} : \Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl \texttt{ : } B_1 \twoheadrightarrow B_1|\emptyset}{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} \texttt{(iterate } x \ e \ wl\texttt{) : } B_1 \twoheadrightarrow B_1|\emptyset}$$

Adding these rules does not violate the safety. The required additions to the proofs are shown below.

- **Preservation**: E-ITERATE is a rule of the form $E_w\left[wl_1\right] \rightsquigarrow_{\mathrm{P}} E_w\left[wl_1'\right]$ where $wl_1 = (D^v) \fatsemi \texttt{(iterate } x \ v \ wl_a\texttt{)}$ and $wl_1' = (D^v) \fatsemi wl_a[x/0] \fatsemi \ldots \fatsemi wl_a[x/(v-1)]$ in which $wl_a[\ldots]$ occurs $\mathsf{max}(0, v)$ times. IT suffices to prove that $B_1|\mathcal{C}_1 = B_1|\mathcal{C}_1$ with $B_1|\mathcal{C}_1$ the type of wl_1 and $B_1'|\mathcal{C}_1'$ the type of wl_1'.

  The `(iterate ...)` construct must have been typed by TP-ITERATE as: $\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} \texttt{(iterate } x \ v \ wl_a\texttt{)} : B_i \twoheadrightarrow B_i|\emptyset$. The premises of this rule gives us that $\Gamma \vdash_{\mathrm{E}} v : \texttt{int}$ and thus that v and $x \mapsto \texttt{int} : \Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl_a \texttt{ : } B_i \twoheadrightarrow B_i|\emptyset$, so with the lemma 18 we get that $\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl_a[x/num] \texttt{ : } B_i \twoheadrightarrow B_i|\emptyset$ for any number num.

  Let $B_d^c|\emptyset$ be the (concrete) type of D^v and let $\sigma = \mathsf{unify}(B_d^c, B_1)$, it holds that $B_d^c = \sigma(B_d^c) = \sigma(B_1)$.

  Now have that the type of wl_1 is $B_1|\mathcal{C}_1 = \mathsf{join}\left(B_d^c|\emptyset\ , B_i \twoheadrightarrow B_i\right) = B_d^c|\emptyset$ (last equality by TM-JOIN).

  The type $B_1'|\mathcal{C}_1'$ of wl_1', the work list after applying the rule, depends on v, the number of iterations. We prove that $B_1'|\mathcal{C}_1' = B_1|\mathcal{C}_1 = B_d^c|\emptyset$ for any v by induction.

  IB If $v = 0$, wl_1' simply becomes (D^v) and the type remains $B_d^c|\emptyset = B_1|\mathcal{C}_1$

  IH For any $v < v_1 . \mathsf{join}^\star(B_d^c|\emptyset\ , \overbrace{B_1 \twoheadrightarrow B_1, \ldots, B_1 \twoheadrightarrow B_1}^{v \text{ times}}) = B_d^c|\emptyset = B_1|\mathcal{C}_1$

  IS For $v_1 - 1$ iterations we had the following:

$$\mathsf{join}^\star(B_d^c|\emptyset\ , \overbrace{B_1 \twoheadrightarrow B_1, \ldots, B_1 \twoheadrightarrow B_1}^{v_1 - 1 \text{ times}}) = B_d^c|\emptyset$$

  We may replace the first $B_d^c|\emptyset$ by $\mathsf{join}\left(B_d^c|\emptyset\ , B_i \twoheadrightarrow B_i\right)$ because $\mathsf{join}\left(B_d^c|\emptyset\ , B_i \twoheadrightarrow B_i\right) = B_d^c|\emptyset$. And we get the following.

$$\mathsf{join}^\star(\mathsf{join}\left(B_d^c|\emptyset\ , B_i \twoheadrightarrow B_i\right), \overbrace{B_1 \twoheadrightarrow B_1, \ldots, B_1 \twoheadrightarrow B_1}^{v_1 - 1 \text{ times}}) = B_d^c|\emptyset$$

  By definition of $\mathsf{join}^\star()$ we now get:

$$\mathsf{join}^\star(B_d^c|\emptyset\ , \overbrace{B_1 \twoheadrightarrow B_1, \ldots, B_1 \twoheadrightarrow B_1}^{v_1 \text{ times}}) = B_d^c|\emptyset$$

  We conclude that $B_1|\mathcal{C}_1 = B_1'|\mathcal{C}_1'$ for any number v, and have hence proven preservation for this case.

- **Progress** For TP-ITERATE, we have the work list $\overline{\texttt{(buf } \overline{v}\texttt{)}} \fatsemi \texttt{(iterate } x \ v_c \ wl_a\texttt{)}$ typed by TM-ITERATE with v_c a value, to which E-ITERATE applies. Because the `(iterate ...)` construct is typed $T_2 = (B_2) \twoheadrightarrow (B_2)$, we know that $\overline{\texttt{(buf } \overline{v}\texttt{)}}$ only contains one buffer. If it were to contain multiple buffers $\mathsf{join}\left(B_1|\mathcal{C}_1\ , (B_2) \twoheadrightarrow (B_2)\right)$ would not succeed.

  If v_c is not a value, E-OTHER applies (corollary 16).

We also need a substitution lemma for work items in this case:

Lemma 18. *For any transformation t, well-typed under Γ, and for any value v such that $\Gamma \vdash_{\mathrm{E}} t[x/v] : \Gamma[x]$ it holds that if $\Gamma \mid \emptyset \vdash_{\mathrm{P}} t \texttt{ : } T|\emptyset$ then$\Gamma \mid \emptyset \vdash_{\mathrm{P}} t[x/v] \texttt{ : } T|\emptyset$ (if the same fresh variable names are chosen)*

And the

```
abstraction veclen_part1(baseX:int,baseY:int): (A[n],A[n]) -> A[n]{
    shaper(i:int, x:A[n], y:A[n]): (A × A)[n]{
        tuple(x[i],y[i])
    }
}
abstraction veclen_part2(baseX:int,baseY:int): (float,float)[n] -> float[n]{
    mapper(i: int, d: (float × float)): float{
        sqrt((d._1-baseX)^2 + (d._2-baseY)^2)
    }
}

(retrive x newy) ⨟ (call veclen_part1 0 0) ⨟ (call veclen_part2 0 0)
```

Listing 12: Desugarred version of lising listing 6

Appendix D.2. Abstractions with multiple transformations

The typing rules, as they are presented in section 3 do not allow abstractions to have more than one transformation. This is because the TP-List rule does not allow transformations to be composed. The defined abstractions are syntactic sugar for defining multiple abstractions.

To use abstractions with multiple transformations in our original semantics, they need to be desugared into multiple abstractions. A call to such an abstraction desugars to a sequence of calls to the split up abstractions. The program in listing listing 6 (page 6) for example is desugared to listing 12.

During the development Gaiwan we nevertheless formalized the typing rules needed to allow abstractions with multiple transformations. The required rules are shown below. These rules are used in our implementation of the type system. Note how TP-ListTrans and TM-JoinTrans do not need to deal with constraints as transformation types do not have constraints.

$$\text{TP-ListTrans}\quad \frac{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl : T_1|\emptyset \qquad \Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} w : T_2|\emptyset \qquad T|\emptyset = \mathsf{join}\,(T_1, T_2)}{\Gamma \mid \mathcal{A} \vdash_{\mathrm{P}} wl \fatsemi w : T|\emptyset}$$

$$\text{TM-JoinTrans}\quad \frac{\sigma = \mathsf{unify}((B_2),(B_3)) \qquad FV(B_2) \subseteq FV(\overline{B_1}) \qquad FV(B_4) \subseteq FV(B_3)}{\sigma(T)|\emptyset = \mathsf{join}\,((\overline{B_1} \twoheadrightarrow (B_2)),((B_3) \twoheadrightarrow (B_4)))}$$

We chose not to include these rules in our core semantics because they further complicate the proofs and do not simplify presentation.

Appendix E. Benchmark results

The full benchmark results are shown below. We included an additional column showing the execution time of a sequential implementation of Bitonic Sort executed on an Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz. Missing values rendered as 0.00.

$\log_2 n$	OpenCl (ms)	Gaiwan (ms)	$\frac{\text{Gaiwan}}{\text{OpenCL}}$	Gaiwan Full (ms)	$\frac{\text{Gaiwan Full}}{\text{OpenCL}}$	CPU (ms)
13	0.76	34,000	44,533.0	87,000	$1.14 \cdot 10^5$	1.49
14	0.79	38,000	47,823.0	94,000	$1.18 \cdot 10^5$	3.26
15	0.84	48,000	56,960.0	$1.1 \cdot 10^5$	$1.31 \cdot 10^5$	7.15
16	0.91	59,000	64,493.0	$1.29 \cdot 10^5$	$1.41 \cdot 10^5$	15.34
17	1.01	78,000	77,124.6	$1.56 \cdot 10^5$	$1.54 \cdot 10^5$	33.84
18	1.21	97,000	79,953.5	$1.76 \cdot 10^5$	$1.45 \cdot 10^5$	73.06
19	1.48	$1.13 \cdot 10^5$	76,424.0	$1.99 \cdot 10^5$	$1.35 \cdot 10^5$	159.95
20	2.08	$1.33 \cdot 10^5$	64,063.0	$2.24 \cdot 10^5$	$1.08 \cdot 10^5$	347.68
21	3.29	$1.53 \cdot 10^5$	46,558.0	$2.53 \cdot 10^5$	76,988.00	770.01
22	5.80	$1.82 \cdot 10^5$	31,372.0	$2.9 \cdot 10^5$	49,988.00	1,685.03
23	15.18	$2.1 \cdot 10^5$	13,836.4	$3.27 \cdot 10^5$	21,545.10	3,755.78
24	32.16	$2.61 \cdot 10^5$	8,116.5	$3.82 \cdot 10^5$	11,879.30	8,107.69
25	65.86	$2.67 \cdot 10^5$	4,054.1	$4 \cdot 10^5$	6,073.60	17,482.61
26	137.10	$3.39 \cdot 10^5$	2,472.7	$4.83 \cdot 10^5$	3,523.01	0.00
27	289.15	$3.87 \cdot 10^5$	1,338.4	$5.23 \cdot 10^5$	1,808.76	0.00
28	609.41	$4.31 \cdot 10^5$	707.2	$5.84 \cdot 10^5$	958.30	0.00
29	1,287.92	$4.78 \cdot 10^5$	371.1	$6.44 \cdot 10^5$	500.03	0.00
30	2,721.56	$5.9 \cdot 10^5$	216.8	$7.89 \cdot 10^5$	289.91	0.00

Appendix F. Idris Code Sample

A sketch of a part of an Idris implementation of size-polymorphic types

```
joinSizes : Nat Nat Nat Nat Nat Nat Nat -> (Nat, Nat)
joinSizes fa fb a1 b1 a2 b2 ta tb = ?impl

join :  (Buffer fa fb -> Buffer a1 b1)
-> (Buffer a2 b2 -> Buffer ta tb)
-> {auto p : joinSizes  fa fb a1 b1 a2 b2 ta tb = (fan, fbn, tan, tbn) }
-> (Buffer fan fbn -> Buffer tan tbn)
join = ?impl
```