1 Source language: λV

The syntax of λV is as simple as possible. Expressions can be numbers, true or false, an identifier or a list (consisting of zero or more expressions). There would be no advantage in encoding builtins such as +, $func$ or if into the syntax of the language. Those are simply pre-defined identifiers. In practice, keeping the syntax this simple has the advantage that writing a parser and AST representation is extremely trivial.

$$e ::= num \mid true \mid false \mid id \mid (l)$$
$$l ::= \varnothing \mid e\ l$$

2 Types in λV

$\tau ::= Num \mid Bool \mid Vec\{2,3,4\}\{f,b\} \mid Mat\{2,3,4\}\{f,b\} \mid Rec\,\tau \mid PureRec$

Num is a type for all numbers. We don't seperate between integers and floating point numbers, we just assume all numbers to be floating-point for simplicity and since GPUs are usually optimized for that anyways. For logical true and false, we have Bool. Then, we have various Vector and Matrix types (with dimensions 2, 3 or 4 and floating point "f" or boolean "b" elements). We simplicity, we only allow square-sized matrices at the moment, other ones are rarely needed in shaders anyways. The types $Rec\tau$ and $PureRec$ are helper types that allow us to model the restriction we have to put on recursive functions (namely: only tail-recursion is allowed). To allow minimizing the number of rules and distinct cases, we will write $Vec1b, f$ as synonym for $Bool$ or Num, respectively.

Our typing judegement has the following form:

$$\boxed{R, A \vdash e : \tau}$$

R is a *recursive context*, as explained below. A is a tuple of tuple of expressions, representing the current stack of call arguments. Both R and A are basically needed as workaround for not typing functions while allowing them in almost any context (there are some technical limitations discussed below). A typing judgement means that e is of type τ (in context R). We also use T, U as type metavariables. [I guess this is fairly common but for tuples we write $((a_1 \dots), a_2)$ for $(a_1 \dots a_2)$, i.e. appending to a tuple].

$$\text{t-num}\ \frac{}{R, \varnothing \vdash num : Num}$$

$$\text{t-true}\ \frac{}{R, \varnothing \vdash true : Bool}$$

$$\text{t-false}\ \frac{}{R, \varnothing \vdash false : Bool}$$

$$\text{t-if}\ \frac{\varnothing, \varnothing \vdash e_1 : Bool \qquad R, A \vdash e_2 : tau_1 \qquad R, A \vdash e_3 : tau_2}{R, A \vdash (if\ e_1\ e_2\ e_3) : \text{rec-match}(tau_1, tau_2)}$$

$$\text{t-app}\quad \frac{R, (A, (e_1, \ldots, e_n)) \vdash e_0 : \tau}{R, A \vdash (e_0\, e_1 \ldots e_n) : \tau}$$

$$\text{t-func}\quad \frac{R, A_r \vdash e[e_1/id_1] \ldots [e_n/id_n] : \tau}{R, ((e_1 \ldots e_n), A_r) \vdash (func\, (id_1 \ldots id_n)\, e) : \tau}$$

$$\text{t-rec-func}\quad \frac{R, \varnothing \vdash e_i : \tau_i \qquad \overrightarrow{\tau_i}, A_r \vdash e[e_1/id_1] \ldots [e_n/id_n] : Rec\, \tau}{R, ((e_1 \ldots e_n)) \vdash (\text{rec-func}\, (id_1 \ldots id_n)\, e) : \tau}$$

$$\text{t-rec}\quad \frac{n > 0 \qquad len(\tau_i) = n \qquad \varnothing, \varnothing \vdash e_i : \tau_i}{\overrightarrow{\tau_i}, ((e_1 \ldots e_n)) \vdash rec : PureRec}$$

$$\text{t-let}\quad \frac{R, A \vdash e[id_1/e_1] \ldots [id_n/e_n] : \tau}{R, A \vdash (let\, ((id_1\, e_1) \ldots (id_n\, e_n))\, e) : \tau}$$

The helper function *rec-match* combines two different types in a recursive context.

$$rec{-}match(\tau_1, \tau_2) := \begin{cases} \tau_1, & \text{for } \tau_1 = \tau_2, \\ \tau_1, & \text{for } \tau_1 = Rec\, \tau_2, \\ \tau_2, & \text{for } \tau_2 = Rec\, \tau_1, \\ \tau_2, & \text{for } \tau_1 = PureRec \wedge \tau_2 = Rec\, T, \\ \tau_1, & \text{for } \tau_2 = PureRec \wedge \tau_1 = Rec\, T, \\ Rec\, T, & \text{for } \tau_2 = PureRec \wedge \tau_1 = T \text{ (where T isn't } PureRec \text{ or } Rec\, U), \\ Rec\, T, & \text{for } \tau_1 = PureRec \wedge \tau_2 = T \text{ (where T isn't } PureRec \text{ or } Rec\, U) \end{cases}$$

Substituion is assumed to be context-sensitive (i.e. only substitute those identifiers that are really meant in that case and not those that are redefined in a deeper scope).

When typing expressions inside a *rec-func* construct, the recursive context R holds the types the *rec-func* was called with. This means that one cannot call recursive functions with function objects. In practice this is a technical limitation we cannot overcome since SPIR-V does not support dynamic dispatch and allowing recursion on arbitrary (possibly different for each recursive call) functions yields cases where we can't inline function calls anymore, i.e. can't unroll recursive functions to simple loops. In practice, we could put a more relaxed restriction on our type system: It is allowed to call recursive functions with function values as long as all recursive calls use the same function value. Or even more general: As long as there is only a finite number of functions used in the recursive calls (meaning basically that you recurse with newly instantianted functions in each recursion, it should be somewhat intuitive that a case like that can't be inlined/unrolled anymore). But that is a much more complicated restriction, yields a more complex type system and code generation. And in practice one can simply use workarounds. For instance:

```
(let ((nat-fold (func (n accum f) (
  let ((body (rec-func (n accum) (
       if (eq n 0)
         accum
         (rec (- n 1) (f accum n))
    ))))
    (body n accum)
)))) ... )
```

One can use function value parameters in recursive functions by simply defining a non-recursive wrapper function.

The author did not know about the Curry paradox and functional combinators (maybe he shouldn't have picked a functional source language) and one can write recursive expressions like that in λV as well. Below is a (simpler) example showing how simple *func* constructs can be abused to get recursion.

```
(let
  ((sumup (func (self n)
    (if (eq n 0) 0 (+ n (self self (- n 1)))))))
  (sumup sumup 10))
```

But those expressions are not well-types. There is no derivation tree for the well-typedness of the example expression since we define our type rules by substituion, meaning that for recursive function constructs like this one would need an infinitely large derivation tree. In short: we expect programmers to play nice and use the *rec-func* construct we provide since we can't support arbitrary recursion. Sadly.

There are furtheremore a lot of more uninteresting typing rules for the builtin primitives such as arithmetic or trigonometric functions. We annotate those builtins with types (just some examples in the list below):

- $Vecif \to Vecif$, e.g. the unary minus, fract, exp
- $Vecif \to Num$, e.g. length
- $Vecbf \to Bool$, e.g. anyOf
- $Vecif \times Vecif \to Vecif$, e.g. plus
- $Vecif \times Vecif \to Num$, e.g. distance
- $Vecif \times Vecif \to Vecbf$, e.g. lessThan or equal

Note that the generic i must be the same for all parameters/return types. This allows us to just give one generic rule for all of those builtins:

$$\frac{\textit{builtin} \text{ of type } (\tau_1 \dots \tau_n) \to \tau \qquad \varnothing, \varnothing \vdash e_i : \tau_i}{R, \varnothing \vdash (\textit{builtin}\, e_1 \dots e_n) : \tau}$$

3 λV operational semantics

To reason about correctness properties of our translation, we will define small-step operational semantics for λV.

$$\frac{}{\vdash (let\ ((id_1\ e_1) \ldots (id_n\ e_n))\ e) \to e[e_1/id_1] \ldots [e_n/id_n]}$$

$$\frac{}{\vdash (if\ true\ e_2\ e_3) \to e_2}$$

$$\frac{}{\vdash (if\ false\ e_2\ e_3) \to e_3}$$

$$\frac{}{\vdash ((func\ (id_1 \ldots id_n)\ e)\ e_1 \ldots e_n) \to e[e_1/id_1] \ldots [e_n/id_n]}$$

$$\frac{}{\vdash ((\text{rec-func}\ (id_1 \ldots id_n)\ e)\ e_1 \ldots e_n) \to e[e_1/id_1] \ldots [e_n/id_n][(\text{rec-func}\ (id_1 \ldots id_n)\ e)/rec]}$$

The reduction rule for builtins are intuitively defined. Overflow is handled by returning $\pm\infty$, operations on $\pm\infty$ or NaN lead to NaN. Operations like division by zero or normalizing a zero-length vector give undefined results. Basically, the semantics of all builtins are just copied from SPIR-V.

The conversion semantics are fairly simple due to our general list syntax:

$$\frac{\vdash e \to e'}{\vdash e \to^+ e'}$$

$$\frac{\vdash e_i \to^+ e_i'}{\vdash (\ldots e_i \ldots) \to^+ (\ldots e_i' \ldots)}$$

Reduction semantics:

$$\frac{}{\vdash e \to^* e}$$

$$\frac{\vdash e \to^+ e'' \qquad \vdash e'' \to^* e'}{\vdash e \to^* e'}$$

4 Translation

We model translation as a recursive function *translate*:

$$\begin{aligned}(ce \times D \times R_i \times A \times ID) &\to (C \times ID \times ID \times I \times R_o) \\ (expr \times defs \times r_i \times args \times id_i) &\mapsto (consts \times id_o \times id_t \times instructions \times r_o)\end{aligned}$$

ce is simply the expression to be translated. We extend expressions by an intermediate type generated during translation, an already translation expression holding its ID, its type ID and its type. $ce ::= e \mid genexpr(ID, ID, \tau)$

D is a mapping from identifiers to expressions and their environment. This is basically how we realize context-relative substitution in our translation.

R_i and R_o are information needed to generate recursive functions. R_i is a tuple of header and continue block IDs for the currrent recursive frame, as well as the type IDs of its parameters. R_o is a set of blocks and their respective parameter IDs for recursvie calls (i.e. edges to the continue block).

A models the current argument call stack, pretty much the same way we modeled it for our typing rules.

ID represents a SPIR-V ID. There are input and output to the translation function for the next usable id (e.g. if a translation of an expression gets 42 as input, uses IDs 43, 44 and 45, it returns 46) as well as an output for the type ID of the generated expression.

C is a set of defined constant instructions. In SPIR-V constants cannot be defined inline but have to be defined in a special section before the start of the program, that's why keep them as a separate vector.

I is the generated SPIR-V instruction vector.

5 Correctness

Basically, when $eval_\lambda(e) = o$ in λV and $translate(e, \varnothing, \varnothing, \varnothing, 20) = (c, \_, \_, I, \varnothing)$, then $eval_S(c, I) = o$