1 Introduction

We present the translation from a simple high-level language with functional elements and LISP-like syntax to SPIR-V, an abstract assembly language for GPU shaders and kernels.

2 Source language: λV

Our source language is a simple shader language with functional elements. To keep proofs and formalization minimal, we will only look at a simple version of λV (basically a subset of what I implemented [1]):
We only look at fragment shaders and only allow a single observation: one output to the framebuffer. This isn't too much of a simplification for fragment shaders and modeling writes to multiple framebuffer attachments as well as inputs shouldn't be too difficult but just more writing work. Similarly, there isn't a huge difference to other shader types (such as vertex or compute shaders) except that those usually produce more or different observations and therefore one just once again needs to model the observations in a more complicated way (especially when allowing arbitrary buffer or image stores as needed to make compute shaders useful). But I don't expect the simplified model we use here to be too hard to extend to cover the given cases.

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.

$$
\begin{aligned}
e &::= num \mid true \mid false \mid identifier \mid (l) \\
l &::= \varnothing \mid e\, l
\end{aligned}
$$

3 The target language: SPIR-V

SPIR-V [2] is an SSA-form abstract typed assembly language. It has similarities to LLVM IR but is more limited: no dynamic dispatch, i.e. no function pointers, no memory allocations, no stack frame and most runtimes (i.e. where SPIR-V modules can be used as shaders or kernels) don't allow recursion, see for instance the Vulkan specification[3] for this. On the other hand, SPIR-V provides some GPU-specific primitives. It's specification does not give operational semantics - or any formal specification - at all but rather describes the layout and semantics in plain text. To formally argue about correctness we need to model at least some form of operational semantics though. We will only consider the subset of SPIR-V used

[1] See `https://github.com/nyorain/lambdaV` for my experiments with implementing a real compiler, a lot of the formal proof here is actually quite close to the compiler source code

[2] https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html

[3] `https://www.khronos.org/registry/vulkan/specs/1.2/html/chap35.html#spirvenv`

by our compiler though. For instance, our compiler never output any functions (apart from the main entrypoint) so we don't care for that.

Our judgment looks like this: $M, \vec{I}, ID, ID \to M, \vec{I}, ID, ID$. ID is a SPIR-V identifier, i.e. just a number. M is of the form $[id \mapsto V]$... the memory and maps IDs to values. Values can be

- instruction blocks of form $\vec{I}$
- values of form (V, τ), where the second value is the type,

$\vec{I}$ is an instruction vector. Furthermore, the function has inputs and outputs for the current and previous block IDs, this is needed to resolve SSA phi instructions.

We formally model execution of a (valid) SPIR-V module like this: when the module is loaded, all constants and types (declared in the header) are loaded into a memory M. All instruction blocks are loaded into the memory as well, removing the first *OpLabel* instruction that is only used to identify the blocks with an id. The header of the SPIR-V module defines the entry point function. Execution looks up the first block in the function (functions must start with a label defining the block id) $(\vec{I}_{entry}, id_{entry})$ and then behaves as specified by the operational semantics for $M, \vec{I}_{entry}, id_{entry}, 0)$.

We represent SPIR-V in the standard textual assembly format, with the new ID defined by the instruction (if any) on the left of the "=" sign.

3.1 Control flow

$$\frac{M(tid) = \vec{I}_t}{M, (\text{OpBranch}\, id_t), c, p \to M, \vec{I}_t, id_t, c}$$

$$\frac{M(id_c) = (true, Bool) \qquad M(id_t) = \vec{I}_t}{M, (\text{OpBranchConditional}\, id_c\; id_t\; id_f), c, p \to M, \vec{I}_t, id_t, c}$$

$$\frac{M(id_c) = (false, Bool) \qquad M(id_f) = \vec{I}_f}{M, (\text{OpBranchConditional}\, id_c\; id_t\; id_f), c, p \to M, \vec{I}_f, id_f, c}$$

$$\frac{p = parent_i \qquad M(var_i) = V \qquad M' = M[id_r \mapsto V]}{M, (id_r = \text{OpPhi}\, id_{type}\; var_1\; parent_1 \ldots var_n\; parent_n, \vec{I}), c, p \to M', \vec{I}, c, p}$$

3.2 Computations

Most instructions simply run a computation. Defining the operational semantics for those is not too interesting and defining all the rules for the various functions wouldn't be helpful. They all look more or less like this example we give for floating point addition:

$$\frac{M(id_1) = (num_1, Num) \qquad M(id_2) = (num_2, Num) \qquad M' = M[id_r \mapsto fadd(num_1, num_2)]}{M, (id_r = \text{OpFAdd}\, id_{type}\; id_1\; id_2, \vec{I}), c, p \to M', \vec{I}, c, p}$$

where $fadd$ simply encodes the semantics of the addition of two numbers. Interestingly enough, SPIR-V does not specify how overflow or special cases (infinity or NaN arguments) are handled. Instead, this is usually specified in the runtime environment. We can for instance look once again into the vulkan specification. It specifies that “By default, the implementation may perform optimizations on half, single, or double-precision floating-point instructions that ignore sign of a zero, or assume that arguments and results are not NaNs or infinities.” and “NaNs may not be generated. Instructions that operate on a NaN may not result in a NaN.”. Basically everything that uses or results in special floating point values is undefined behavior, more or less. Newer SPIR-V versions support a flag signaling that infinities and NaNs must be preserved (since SPIR-V 1.4) but we also want to target SPIR-V 1.0 and implementations that do not provide the optional support for this flag. Given these non-guarantess, we can’t even check for infinity or NaN *after* we do an operation since then we might already have triggered undefined behavior, at least that is my interpreation of this specification. But checking whether an operation might overflow (or similar) is a pain (maybe not even possible, given that Vulkan and SPIR-V give their implementations some freedom regarding rounding of values returned by computations).

I couldn’t actually find a solution for this yet. I tried to get a clarification on this section in the Vulkan spec and found I was not the first person confused about it, see Vulkan-Docs issue 961. Possible solutions included these:

- Just make any overflow or similar undefined in the source language as well. I don’t want to do this since one of the main motivations in the first place was to get an target language program that is as deterministic as possible (at least detecting triggered undefined behavior).
- Actually evaluating whether an operation would operate on or return infinity or NaN at runtime. For each operation. That’s such a huge pain. I’m not even thinking about runtime cost here, implementing a check that safely evaluates whether addition (and multiplication, division, exp, exp2, pow, . . .) of floating point numbers would give such a result seems like a lot of work and definitely not a sane solution to me.
- Just outputting SPIR-V 1.4 (or using the previously available extension) and requiring support for this flag. That is what I went with in the end since my hardware supports it and it makes a big issue just go away, basically. Now, most computations don’t ever trigger any undefined behavior.

There are still some instructions, however, that trigger undefined behavior.

3.3 Ignored instructions

There are meta-instructions that must be inserted into a SPIR-V module for correcntess that don’t have an impact on the semantics, like the OpSelectionMerge and OpLoopMerge that provide meta-information about the control flow. Furthermore there are debug instructions, allowing to associate source-language line numbers or variables names with SPIR-V code. We simply ignore all those instructions, treating them the same way we treat OpNop:

$$\frac{}{M, (\text{OpNop}, \vec{I}), c, p \rightarrow M, \vec{I}, c, p}$$

4 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 $PureRec$ have nothing to do with recursive types but are just helper types that allow us to model the restriction we have to put on recursive functions (namely: only tail-recursion is allowed) while still deducing the types of recursive expressions. This is probably the only interesting thing about the type system. How exactly the types are used should become apparent from the typing rules below. To allow minimizing the number of rules and distinct cases, we will write $Vec(1,b,f)$ as synonym for $Bool$ or Num, respectively.

Our typing judgment 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 $((a_1 \ldots), a_2)$ for $(a_1 \ldots a_2)$, i.e. appending to a tuple].

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

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

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

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

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

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

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

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

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

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), as usually.

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

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

PureRec is the type of a *rec* call but when we have an *if* expression where one branch just results in a *rec* call, i.e. having type *PureRec* while the the other branch contains a value of type τ (or $Rec\,\tau$), we can deduce that the function must return a value of type τ in general. But instead of giving this *if* expression then the type τ, we give it the type $Rec\,\tau$ since the returned value can't be used for any further computations (except control flow, at the moment this only means *if*). Furthermore, this type system encodes the requirement for *rec-func* constructs to have at least one *rec* call in its body (since typing requires the body of *rec-func* to be of type $Rec\ \tau$).

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 (since they are not typed). 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 recursive 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 (simple) example showing how simple $func$ constructs can be (ab-)used 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-typed in our source language. There is no (finite) 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 (independent from whether or not the expression actually terminates). 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):

- $Vec(I, f) \to Vec(I, f)$, e.g. the unary minus, fract, exp
- $Vec(I, f) \to Num$, e.g. length
- $Vec(b, f) \to Bool$, e.g. any-of
- $Vec(I, f) \times Vec(I, f) \to Vec(I, f)$, e.g. plus
- $Vec(I, f) \times Vec(I, f) \to Num$, e.g. distance
- $Vec(I, T) \times Vec(I, T) \to Vec(I, b)$, e.g. less-than 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:

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

5 λV operational semantics

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

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

$$\overline{(if\ true\ e_2\ e_3) \rightarrow e_2}$$

$$\overline{(if\ false\ e_2\ e_3) \rightarrow e_3}$$

$$\overline{((func\ (id_1 \ldots id_n)\ e)\ e_1 \ldots e_n) \rightarrow e[e_1/id_1] \ldots [e_n/id_n]}$$

$$\overline{((\text{rec-func}\ (id_1 \ldots id_n)\ e)\ e_1 \ldots e_n) \rightarrow e[e_1/id_1] \ldots [e_n/id_n][(\text{rec-func}\ (id_1 \ldots id_n)\ e)/rec]}$$

The reduction rule for builtins are intuitively defined, just copying the underlying SPIR-V (and target environment, e.g. Vulkan) semantics, which in turn usually just refer to the IEEE floating point standard.

The design decision we made for handling the (by default ill-defined) issues like overflow, infinities and NaN's in Section 3.2 is important here since it significantly modifies the semantics of those builtins in SPIR-V. Otherwise we would transitively introduce a lot of undefined behavior in our source language.

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

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

$$\frac{\vdash e_0 \rightarrow^{+} e_0'}{\vdash (e_0 \ldots) \rightarrow^{+} (e_0' \ldots)}$$

Reduction semantics:

$$\overline{\vdash e \rightarrow^{*} e}$$

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

As usually, we define a evaluation function, $eval_\lambda(e) = o$, that is defined as the observation o that e can be reduced to. All possible observations in our simplified version of the source language is a single $Vec(I, f)$, the value written to the framebuffer by the fragment shader.

6 Translation

We model translation as a recursive function *translate*:

$$
\begin{aligned}
BackEdge &::= (ID, ID*) \\
R_o &::= BackEdge* \\
CE &::= e \mid genexpr(ID, \tau) \\
ID &::= \text{SPIR-V numeral ID} \\
D &::= identifier \to (CE, D) \\
A &::= (D, ce*)* \\
C &::= [ID \mapsto Num]* \\
\\
Input &::= \{expr : CE, defs : D, args : A, idc : ID, idl : ID, rec : ID\} \\
Output &::= \{code : \vec{I}, consts : C, ido : ID, idc : ID, idl : ID, type : \tau, rec : R_o\}
\end{aligned}
$$

$$translate : Input \to Output$$

Input.expr is simply the expression to be translated. We extend expressions by an intermediate type generated during translation, an already translation expression holding its ID and its type.
Input.defs is a mapping from identifiers to expressions and their environment. This is basically how we realize context-sensitive substitution in our translation.
Input.rec and **Output.rec** are information needed to generate recursive functions. Inside a *rec-func* construct, **Input.rec** is the block ID of the continue block, to which recursive calls should jump.
Output.rec is a set of blocks and their respective parameter IDs for recursvie calls (i.e. edges to the continue block).
Input.args models the current argument call stack, pretty much the same way we modeled it for our typing rules. This is once again needed because we can't translate function expressions on their own.
ID represents a SPIR-V ID.
Input.idc is the next usable ID.
Input.idl is the ID of the current code generation block (i.e. label).
Output.ido is the ID holding the result of the translated expression.
Output.idc is the next usable ID after the translation. E.g. if a translation of an expression gets *Input.idc* $= 42$ as input, uses IDs 42, 43, 44 and 45, it returns 46 as *Output.idc*.
Output.idl is the label the code generation finishes in. This is different iff code generation inserts a new *OpLabel* instruction, i.e. starting a new basic block.

Output.consts 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.

Output.type is the type of the generated expression.

Finally, *Output.code* is the generated SPIR-V instruction vector.

6.1 Translation: utility

We define the function as a set of conditions in which $translate(I) = O$ is defined. The next sections will present a set of conditions for each expression, you can basically imagine each section being one (giant) derivation rule for the defining judgment $translate(I) = O$ with all the conditions as premises.

We first need a utility function *ioa*, modeling the *insert or assign* semantic of a mapping (in our compiler we simply use a hash map):

$$ioa(D, id \mapsto v)(id_c) := \begin{cases} v & \text{for } id = id_c \\ D(id_c) & \text{otherwise} \end{cases}$$

We write $ioa(D, (id \mapsto v)^*))$ as a shortcut for subsequent insertion/replacement,

$$ioa(ioa(\dots ioa(D, id_1 \mapsto v_1) \dots), id_n \mapsto v_n)$$

Furthermore, when generating the SPIR-V module from a full λV program, we define all types in the header (giving them IDs) and can therefore define a function $typeid : \tau \to ID$ that returns the SPIR-V type ID associated with a given type in our formalization. We define $typeid(Rec\tau) = typeid(\tau)$. The expression $typeid(PureRec)$ is intentionally undefined, it is never needed for well-defined programs. Similar to the way we declare types once and can then use them via their IDs during the translation, we also define the constants *true* and $false$ once and are able to use them during translation. Their translation rules are not shown below since they are therefore trivial (basically a no-op, just returning the ID of the respective constant).

6.2 Translation: func

This is probably the most interesting (and yet one the most simple) translation rules: to translate a function call we simply translate its body (effectively always inlining the function) and replace all occurrences of function parameter with translations of the expressions bound to them (see the translation of identifiers in the next subsection). Of course, this might lead to code bloat, when huge functions or complex expression passes as paremters are inlined. This compiler doesn't care too much and separate optimization passes can still perform common subexpression elimination or similar (they could technically even refactor code that is generated multiple times out into its own function, if possible). The reason we translate functions (or rather: function calls; we can't translate function in itself, remember how they are not even valid expressions since not typed at all) like this even though SPIR-V offers functions is that there is no dynamic dispatch in SPIR-V. Therefore force-inlining basically everything is the only way to get higher-order functions (well, with the restrictions outlined in the beginning).

$$\begin{aligned}
\text{I.expr} &= (\text{func } (id_1 \dots id_n) \text{ e}) \\
\text{I.args} &= (A_r, (\text{ndefs}, e_1 \dots e_n)) \\
\text{ndefs} &:= \text{ioa}(\text{I.defs}, (id_i \mapsto (e_i, \text{I.defs}))\dots) \\
\text{O} &= \text{translate}(\{\text{expr: e, defs: ndefs, args: } A_r\text{, idc: I.idc, idl: I.idl, rec: I.rec}\})
\end{aligned}$$

6.3 Translation: identifier

$$
\begin{aligned}
\text{I.expr} &= \text{identifier} \\
\text{I.defs(identifier)} &= (\text{e, I.defs}) \\
\text{O} &= \text{translate}(\{\text{expr: e, defs: ndefs, args: I.args, idc: I.idc, idl: I.idl, rec: I.rec}\})
\end{aligned}
$$

6.4 Translation: numbers

All constants must be declared in the SPIR-V header, we therefore don't generate code for a constant number but simply return our constant definition (in O.consts) and return the id of the constant.

$$
\begin{aligned}
\text{I.expr} &= num \\
\text{I.args} &= \varnothing \\
\text{O.code} &= \varnothing \\
\text{O.consts} &= [\text{I.idc} \mapsto num] \\
\text{O.idc} &= \text{I.idc} + 1 \\
\text{O.type} &= \text{Num} \\
\text{O.idl} &= \text{I.idl} \\
\text{O.rec} &= \varnothing
\end{aligned}
$$

6.5 Translation: let

$$
\begin{aligned}
\text{I.expr} &= (\text{let } ((id_1 e_1) \ldots (id_n e_n)) \text{ e}) \\
\text{ndefs} &:= \text{ioa}(\text{I.defs}, (id_i \mapsto e_i, \text{I.defs})) \ldots) \\
\text{O} &= \text{translate}(\{\text{expr: e, defs: ndefs, args: I.args, idc: I.idc, idl: I.idl, rec: I.rec}\})
\end{aligned}
$$

6.6 Translation: list

This translation rule is only useful (and well-defined) when e_0 isn't an atomic expression such as $(func \ldots)$ or a builtin.

$$
\begin{aligned}
\text{I.expr} &= (e_0\ e_1\ \ldots e_n) \\
\text{nargs} &:= (\text{I.args}, (\text{I.defs}, e_1 \ldots e_n)) \\
\text{O} &= \text{translate}(\{\text{expr: } e_0\text{, defs: I.defs, args: nargs, idc: I.idc, idl: I.idl, rec: I.rec}\})
\end{aligned}
$$

6.7 Translation: if

This is the first translation actually generating SPIR-V code. The main complexity in this translation is handling cases where one (or both) of the given branches is just a recursive call (i.e. of type $PureRec$).

$$
\begin{aligned}
\text{I.expr} &= (\text{if } e_c\ e_t\ e_f) \\
O_c &= \text{translate}(\{\text{expr: } e_c\text{, defs: defs, args: I.args, idc: I.idc + 4, idl: I.idl, rec: } \varnothing\}) \\
O_t &= \text{translate}(\{\text{expr: } e_t\text{, defs: defs, args: I.args, idc: } O_c\text{.idc, idl: I.idc + 0, rec: I.rec}\}) \\
O_f &= \text{translate}(\{\text{expr: } e_f\text{, defs: defs, args: I.args, idc: } O_f\text{.idc, idl: I.idc + 1, rec: I.rec}\}) \\
\text{O.consts} &= \text{consts: } O_c\text{.consts } O_t\text{.consts } O_f\text{.consts} \\
\text{O.ido} &= \text{I.idc} + 4 \\
\text{O.idl} &= \text{I.idc} + 2 \\
\text{O.type} &= \begin{cases} O_t.type & \text{for } O_f.type = PureRec \\ O_f.type & \text{otherwise} \end{cases} \\
\text{O.idc} &= O_f\text{.idc} \\
\text{O.rec} &= O_t\text{.rec, } O_f\text{.rec}
\end{aligned}
$$

O.code is defined as:

	O_c.code	
	OpSelectionMerge (I.idc + 4) None	
	OpBranchConditional O_c.ido (I.idc + 1) (I.idc + 2)	
(I.idc + 0) =	OpLabel	
	O_t.code	
	OpBranch (I.idc + 3)	**[if O_t.type $\neq$ PureRec]**
(I.idc + 1) =	OpLabel	
	O_f.code	
	OpBranch (I.idc + 3)	**[if O_f.type $\neq$ PureRec]**
(I.idc + 2) =	OpLabel	**[if O.type $\neq$ PureRec]**
(I.idc + 3) =	OpPhi typeid(O.type)	**[if O.type $\neq$ PureRec]**
	O_t.ido (I.idc + 1)	**[if O_t.type $\neq$ PureRec]**
	O_f.ido (I.idc + 2)	**[if O_f.type $\neq$ PureRec]**

6.8 Translation: Computations

The various builtins are intuitively translated to a single instruction. For instance, a translation of the "vec4" builtin, using 4 numbers to construct a *Vec(4,f)* could look like this:

$$
\begin{aligned}
\text{I.expr} &= \text{vec4} \\
\text{I.args} &= ((\text{ndefs, } e_1\ e_2\ e_3\ e_4)) \\
O_0 &= \{\text{idc: I.idc} + 1\} \\
O_i &= \text{translate}(\{\text{expr: } e_i\text{, defs: ndefs, args: } \varnothing\text{, idc: } O_{i-1}\text{.idc, idl: I.idl, rec: } \varnothing\}) \ldots \\
\text{O.idc} &= O_4\text{.idc} \\
\text{O.ido} &= \text{I.idc} + 1 \\
\text{O.type} &= \text{Vec(4, f)} \\
\text{O.rec} &= \varnothing \\
\text{O.consts} &= O_i.consts \ldots \\
\text{O.code} &= (\text{I.idc} + 1) = \text{OpCompositeConstruct typeid(Vec(4, f)) } O_1\text{.ido } O_2\text{.ido } O_3\text{.ido } O_4\text{.ido}
\end{aligned}
$$

6.9 Translation: rec-func

Translating *rec-func* is by far the most complicated translation. Since most SPIR-V runtimes don't allow recursion (GPUs traditionally don't have a stack) we have to unroll it into a loop. That is the reason we only support this limited form of recursion in our source language. We provide the frame, including all basic blocks and SSA phi functions for recursive calls from within the function body, i.e. (after translation) jumps back to the begin of the loop body (via a separate continue block SPIR-V needs) from within the loop body.

$$
\begin{aligned}
\text{I.expr} &= (\text{rec-func } \ (id_1 \ldots id_n) \text{ e}) \\
\text{I.args} &= ((\text{ndefs}, e_1 \ldots e_n)) \\
\\
\text{hb} &:= \text{I.idc} + 0 \\
\text{lb} &:= \text{I.idc} + 1 \\
\text{cb} &:= \text{I.idc} + 2 \\
\text{mb} &:= \text{I.idc} + 3 \\
O_0 &:= \{\text{idc: I.idc} + 4\} \\
O_i \ldots &:= \text{translate}(\{\text{expr: } e_i, \text{ defs: ndefs, args: } \varnothing, \text{ idc: } O_{i-1}.\text{idc, idl: I.idl, rec: } \varnothing\}) \ldots \\
D_b &:= \text{ioa}(\text{I.defs}, (id_i \mapsto (\text{genexpr}(O_n.idc + \text{i}, O_i.\text{type})), \text{I.defs}) \ldots) \\
O_b &:= \text{translate}(\{\text{expr: e, defs: } D_b, \text{ idc: } O_n.idc + 2n, \text{ idl: lb, rec: cb}\}) \\
\\
O_b.\text{type} &= \text{Rec } \tau \\
O_b.\text{rec} &= ((block_1, bid_1^1 \ldots bid_n^1) \ldots (block_m, bid_1^m \ldots bid_n^m)) \\
\\
\text{O.rec} &= \varnothing \\
\text{O.consts} &= O_b.\text{consts } O_i.\text{consts} \ldots \\
\text{O.ido} &= O_b.\text{ido} \\
\text{O.idl} &= \text{mb} \\
\text{O.type} &= \tau
\end{aligned}
$$

Furthermore, $O.code$ is defined as:

	OpBranch hb	
	O_i.code	**[for i = 1..n]**
hb =	OpLabel	
$(O_n$.idc + i) =	OpPhi typeid(O_i.type) O_i.ido I.idl (O_n.idc + n + i) cb	**[for i = 1..n]**
	OpLoopMerge mb cb None	
	OpBranch lb	
lb =	OpLabel	
	O_b.code	
	OpBranch mb	
cb =	OpLabel	
$(O_n$.idc + n + i) =	OpPhi typeid(O_n)	**[for i = 1..n]**
	$block_k$ bid_i^k	**[for k = 1..m]**
	OpBranch hb	
mb =	OpLabel	

6.10 Translation: rec

The definition of this *rec* translation only makes sense together with the translation of *rec-func*. Interesting here is that this translation does not return a value or block ID (dummy value 0) in O.ido, O.idl, since neither is defined. For well-typed expressions this will never again be needed during the translation since the result of this expression can't be used anyways, that is the idea of tail recursion. With our current type system, every *rec* expression must be wrapped immediately into an *if* expression.

$$
\begin{aligned}
\text{I.expr} &= \text{rec} \\
\text{I.args} &= ((\text{ndefs}, e_1 \ldots e_n)) \\
O_0 &= \{\text{idc: I.idc}\} \\
O_i &= \text{translate}(\{\text{expr: } e_i, \text{ defs: ndefs, args: } \varnothing, \text{ idc: } O_{i-1}.\text{idc, idl: I.idl, rec: } \varnothing\}) \ldots \\
\\
\text{O.idc} &= O_n.\text{idc} \\
\text{O.ido} &= 0 \\
\text{O.idl} &= 0 \\
\text{O.type} &= \text{PureRec} \\
\text{O.consts} &= O_i.consts \ \ldots \\
\text{O.rec} &= (\text{I.idl}, (O_i.\text{ido} \ldots)) \\
\text{O.code} &= O_i.code \ldots \quad \text{OpBranch I.rec}
\end{aligned}
$$

6.11 Translation: genexpr

We needed this additional expression alternative $genexpr(ID, \tau)$ to realize *rec-func* parameters. Their translation is straightforward, we simply return the already generated ID.

$$
\begin{aligned}
\text{I.expr} &= \text{genexpr}(id_e, \tau_e) \\
\text{I.args} &= \varnothing \\
\text{O} &= \{\text{code: } \varnothing, \text{ consts: } \varnothing, \text{ ido: id, idc: I.idc, idl: I.idl, type: } \tau_e, \text{ rec: } \varnothing\}
\end{aligned}
$$

6.12 Translation to program

TODO: describe how a translated expression is wrapped to generate a valid spirv module. Introduce *startid*, *startblock*.

7 Correctness

We want to prove whole program correctness. Valid programs are expressions that are well-typed with no recursive context and no arguments and have a type τ_o that qualifies as observation, i.e. $Vec(i, f)$, since only those values can be returned (written into a framebuffer) by a fragment shader. We furthermore define the utility function $init_S : (C \times \vec{I}) \to M$ that returns a SPIR-V memory object initialized with the constants from C and the block mappings from the full-program vector given in the $\vec{I}$ argument. Another utility function $eval_S : (M \times ID) \to Vec(1, f)$ models the full evaluation of the program loaded into the present SPIR-V memory and returns the observation mapped to the given ID after program execution according to the SPIR-V operational semantics we outlined. It basically starts execution at the block *startblock* and applies the rules from the SPIR-V operational semantics until no instructions are left and then returns the value present in the memory for the given ID.

The correctness theorem looks like this:

When we have a well-typed program e with $\varnothing, \varnothing \vdash e : \tau_o$ and $eval_\lambda(e) = o$ in λV and $translate(\{expr : e, defs : \varnothing, args : \varnothing, idc : startid, idl : startblock, rec : \varnothing\}) = O$, then $eval_S(init_S(O.consts, O.code), O.ido) = o$.

We proof this by induction over the well-typedness, i.e. the type derivation tree of expression e. We can start with the simple cases, even without additional Lemmas.

7.1 Correctness: numbers

When e is a number *num* it obviously evaluates to itself as observation. Looking up its translation rule, we see that it translates to no code at all. But since $eval_S(init_S(O.consts, O.code), O.ido)$ returns the value of $O.ido$ — in case of the translation of a constant that's simply the ID we gave the constant — in the memory after execution — which in our case is the same memory as before the execution — we simply get *num* since that's the constant we added in our translation, that gets loaded into initial memory by $init_S$. We could trivially formalize this proof given formalizations of $eval_S$ and $init_S$ but the proof would just trivially look like described here.

7.2 Correctness: computations

Since we defined all builtins to just have the semantics of their SPIR-V counterparts (see the example for the operational semantics of OpFAdd, all we do is basically saying "this function behaves like specified in SPIR-V"), the correctness proof for those is trivial. We use the induction hypothesis for the arguments passed to the builtins.

To formally prove this we would have to strengthen our induction argument, stating that $O.type$ is the same as the type of e and that the $O.ido$ is a value of this type as well. Otherwise we can't guarantee that the types actually match and that the instruction is valid. But this can be verified separately or via a Lemma by just looking at each translation rule.

7.3 Correctness: list

When e is a list of form $(e_0 e_1 \dots e_n)$ and e_0 can be reduced to e_0', the operational semantics specify that e can be reduced to $(e_0' e_1 \dots e_n)$. Furthermore, given our typing rule "t-app", we can inductively assume the correctness theorem for e_0 with arguments $(e_1 \dots e_n)$. Here, we once again have to strengthen our induction argument to argue about arguments: When an expression e is only well-typed under arguments $((e_1^1 \dots e_1^{n_1}) \dots (e_m^1 \dots e_m^{n_m}))$, then the evaluations of a translation with any arguments $((d_1, (g_1^1 \dots g_1^{n_1})) \dots (d_m, (g_m^1 \dots g_m^{n_m})))$ so that (for all i = $1..m$; for all k in $1..n_i$) g_i^k with all identifier mappings from d_i substituted is the same as e_i^k must evaluate to the same as $((\dots(e\, e_m^1 \dots e_m^{n_m}) \dots) e_1^1 \dots e_1^{n_1})$.

Given the translation rule for list and our induction hypothesis given by the type derivation, we know that the correctness theorem holds for e_0 with arguments $e_1 \dots e_n$, meaning its translation evaluates to the same as e_0 with those arguments.

7.4 Correctness: let

Now e is of form $(let(\dots(id_i e_i)\dots)e_b)$. Per induction hypothesis and type rule we can assume our correctness argument for e with substituted identifiers. To show that this substitution is the same as adding the definitions to our $I.defs$ mapping that we translate e_b with (i.e. what our translation does), we define a Lemma:

Lemma subst-def: The translation of $(e[r(e_1)/id_1]\dots[r(e_n)/id_n])$ (we write $r(e_i)$ to denote the expression e_i with all identifiers known to the current context — the $defs$ of the translation — replaced by their definitions) with definitions $defs$ evaluates to the same value as the translation of e with definitions $ioa(defs, (id_i \mapsto (e_i, defs))\dots)$. This Lemma is only needed because we chose to model translation closely to how it can be implemented instead of relying on "magical" context sensitive substitution. It should be possible to prove using induction over e, the only interesting rule being the identifiers for which the equality should be obvious, given that we just substitute/replace at different times.

So per induction hypothesis we know that $eval_\lambda(e[e_i/id_i]\dots)$ is the same as the evaluation of the translation in SPIR-V (technically we have to apply the current arguments from our strengthened induction argument to both). But our Lemma gives us that the translation of $e[e_i/id_i]\dots$ evaluates to the same value as e translated with $id_i \mapsto e_i \dots$ definitions, which is exactly how we defined the *let* translation. Furthermore, our operational semantics give us that $(let\,(\dots(id_i\;e_i)\dots)\;e_b)$ can be reduced to $e_b[e_i/id_i]\dots$, i.e. they both evaluate to the same value.

7.5 Correctness: func

7.6 Correctness: rec-func and rec