# 表达式值的存储

## LLVM中基本数据类型及存储类型

值是编译器所需要处理的基本数据。它出现在各个角落，条件分支、表达式、返回语句。甚至是函数地址也可以被视作是值类型。

对于编译器而言，最基本的值是整数和浮点。其他的值都可以用这两者来表达，例如布尔和指针。如果你是从一个最基本的指令集开始写起，那么整数和浮点的数值运算、转换、基于整数寄存器的跳转和地址取值是一个寄存器机的最基本操作。所有更加高级的操作，例如数组、结构体、指针、函数、对象等，都可以建立在这一基础上。

如果一个指令系统在整数和浮点数之外，额外提供了布尔、分支、函数调用和结构体的支持，那么它与高级语言将会贴近更多，生成代码的方式也更加简单。

在高级语义的数据结构上，LLVM提供了相当良好的支持。它支持的原生类型（First class）包括: 各种精度的整型和浮点数，指针、向量，结构体和数组。这些类型的数据存取和运算都是有指令直接支撑，而不需要自行计算并生成更加原始的指令。

在存储类型上，LLVM提供了Value, Argument, Alloca, GlobalVariable, Pointer五种存储类型。Value是右值，它不可取引用，不可更改。Argument表示了函数实参，它是Value的一个派生类。所以对参数的任何更改行为实际上都是不被允许的。Alloca保存了栈地址，GlobalVariable保存了全局变量的地址，Pointer则是一般意义上的指针。

除了存储指令，LLVM所有的指令都是针对Value的操作，并返回一个Value。所以

Var a = Alloca int

Var b = Alloca int

Var c = Alloca int

c = ADD a, b

这样的操作，在LLVM中实际上是将a和b的地址相加，并把C从变量替换成一个左值（注意，是替换，变量的值没有任何变化）。

在LLVM中，正确的做法应当类似于下面这样：

a = Alloca int

b = Alloca int

c = Alloca int

a\_v = load a

b\_v = load b

c\_v = ADD a, b

store c, c\_v

要先将值从变量中读出，进行操作，再保存到另外一个变量中。

## 表达式值的数据结构

一个的表达式参数或结果可能是左值或右值。例如++x输入一个左值返回一个左值，而x++就返回一个右值。A+B则是需要两个右值并返回一个右值。

一个左值可以很方便的转化为右值，但是右值转化成左值通常是很困难的。地址信息被丢弃了，或者它根本就是一个字面常量，都会导致一个右值将永远是右值。将右值构造成左值的唯一办法，就是构造临时对象并将右值赋予左值。当这个左值被读取时，如果临时对象除了初始化之外从未被写过，并且它关联的右值依然有效，那么这个操作会被优化成直接返回那个原始的右值，从而避免临时左值的读写操作。

在Clang（一个C++编译器的前端）中对左值和右值进行了严格的区分。这是由于C++需要额外的处理临时对象。临时对象意味着尽管它有右值的语义，但是实际上是左值的存储。这是需要将真正的左值和临时的左值区分开，并提供特定语境下的转化。

SASL没有处理复杂的临时对象问题，因此它使用了一个相对简单的办法来解决左右值的判定和存储。

我们设计了一个数据结构，用于保存任何可能的值。

struct Data{

bool isRef;

Value\* rval;

Alloca\* local;

GlobalVariable\* global;

struct Aggregated{

Data\* parent;

int index;

} agg;

};

rval用于处理Argument和右值时的情况。Local意味着它是一个局部变量，global说明它是一个全局变量，agg则用于处理structure member。Parent指向包含当前变量的聚合变量，index则指明了当前变量在聚合变量中的位次。

SASL提供了load, load\_ptr 和 store 来数据的存取，而不要关心它的具体存储类型。

## 左值/右值语义

在Data这个结构中，rval, local, global和agg四个值是互斥的。当然这里的我们也可以选择union+enum的方式来表达。

首先来看，这个结构如何表达左值/右值语义。

来看isRef，这是一个标记位。它表示了data存储的值究竟是值本身还是地址。如果是isref为真，那么data便可以被认为是一个左值。Isref为假，那么当它是rval的时候，它就是一个真正的右值了。如果是Alloca或者GlobalVariable，因为它们本身就代表了地址，那么它仍然是一个右值。如果是agg，那么要取决于它的聚合量是左值还是右值。

如果参数需要左值，那么可以直接从data拷贝，或者使用load\_ptr + isRef创建一个新的右值Data。如果参数需要右值，那么可以通过load的方式获取一个右值。

## 数据存取的实现

具体的实现参见可以参见以下代码：

# ****图形管线与Shader的交互****

## ****入口函数与非入口函数****

入口函数是Shader的主函数。来看这样一段程序

Code:

1. float4x4 wvpMat;
3. struct VS\_INPUT{
4. float4 pos: SV\_Position;
5. float4 tex: SV\_Texcoord0;
6. };
8. struct VS\_OUTPUT{
9. float4 pos: SV\_Position;
10. float4 tex: SV\_Texcoord0;
11. };
13. float4 world\_pos( float4 p ){
14. return mul(p, wvpMat);
15. }
17. VS\_OUTPUT vs\_main(VS\_INPUT in){
18. VS\_OUTPUT o;
19. o.pos = world\_pos(in.pos);
20. o.tex = in.tex;
21. return o;
22. }

很显然，vs\_main是一个合法的VS程序的主函数，那么我们称vs\_main为入口函数，称world\_pos为非入口函数。Shading language的入口函数，其实和C语言的主在概念上没有什么区别。但是在SASL中，我们要求一个入口函数它所有的输入和输出都要正确的关联到语义 上。SM4中这一条件被放宽了，入口函数也可以提供无语义的uniform参数。

## ****语义分类****

对于Shading Language而言，最重要的两个操作是从图形管线中获取数据并将数据写回到管线中。流水线中的数据是附带了语义信息的，用于表达这个数据的用途。例如 SV\_Position就指明了这样一个数据是表示位置的。用户输入的数据、SL输出的数据，都是依靠语义信息来确保读取和写入的正确性。例如 SV\_Position只能从某个顶点流的特定偏移量获取，SV\_Color的数据才能被写到color buffer中。

SASL支持的语义集合是HLSL Shader Model 4.0的子集。目前参考的HLSL版本为4.0。

在Shader Model 4.0的所有输入语义中，一些语义的值直接来自于外部存储，例如SV\_Position的数据来自顶点流，一些语义的值则是来自于管线执行中间计算的结果。输出语义也是如此。

Shader从设计之初便需要应对每秒百万到数亿的调用，因此一些平常不可见的开销问题在这里也变得尤为显著，例如函数参数压栈的开销。所 以将所有输入数据均按值或者按地址传递到入口函数中是不妥的。为了尽可能的减少内存读写的次数，从外部存储读入（例如Vertex Buffer）或者写入的外部存储（例如Stream Output或者Frame Buffer）的数据，我们一律以指针+偏移的形式将数据传递到Shader中，称之为Stream类型，而临时的语义变量，如 SV\_IsFrontFace，我们则暂存到一个临时的buffer中，称之为buffer类型。

在SASL中我们将shader的全部语义分为四类，Stream\_in，stream\_out，buffer\_in，buffer\_out。

Shader还有一种特有的存储类型，uniform。这一类型在编译期的时候是一个变量，在代码生成期/优化期是一个常量。如果将这一类 型的量按照编译期常量来处理，那么便能获得更高的运行时性能，比方说一些条件展开可以通过优化而被消除。但是，这也意味着一旦uniform量发生变化 后，shader便最少需要重新执行代码生成乃至于重新编译。这将会带来巨大的性能开销。由于SASL主要执行在CPU上，CPU对于动态代码的执行优化 要远远优于GPU，例如间接地址读取指令和分支预测。因此我们将uniform作为一个普通的变量经由buffer\_in来执行输入，以平衡代码调用和编 译之间的开销。

## ****数据结构与入口签名****

SASL最终将生成如下的签名：

Code:

1. struct stream\_in{
2. float4\* pos;
3. float4\* tex;
4. };
6. struct buffer\_in{ float4x4 wvpMat; };
7. struct stream\_out{}; // empty.
8. struct buffer\_out{
9. float4 pos;
10. float4 tex;
11. };
13. float4 world\_pos( float4 pos, buffer\_in\* bi );
14. void vs\_main( stream\_in\* si, buffer\_in\* bi, stream\_out\* so, buffer\_out\* bo );

通过对语义和常量进行重整，SASL减少了不必要的拷贝开销。

## ****结构体的语义布局与常规布局****

我们注意到，VS\_OUTPUT对于返回值和堆栈变量的类型时的意义是不同的。在返回值时，它匹配了语义输出，而在堆栈变量时，它只是一个普通结构体的内存布局。这就要求，VS\_OUTPUT在分析时必须同时产生并保存两套内存布局信息。

但是实际上由于布局差异仅仅在入口函数才存在，并且只有当结构体作为入口函数参数或返回值的时候才会使用语义布局，其他函数内无论是参数还 是变量都是使用普通布局，因此我们运用一个临时对象，将语义布局的值拷贝成一个普通布局的对象。也就是说，入口函数内的代码中所有对这个参数值的读取实际 上都是对临时对象的读取。其代码类似于下段：

Code:

1. void vs\_main( stream\_in\* si, buffer\_in\* bi, stream\_out\* so, buffer\_out\* bo ){
2. // initialization
3. VS\_INPUT \_\_tmp\_in = {\*si->pos, \*si->tex};
4. VS\_OUTPUT \_\_tmp\_out;
5. // end initialization
7. VS\_OUTPUT o;
8. o.pos = world\_pos( \_\_tmp\_in.pos, bi );
9. o.tex = \_\_tmp\_in.tex;
11. \_\_tmp\_out = o;
13. // return
14. bo->pos = \_\_tmp\_out.pos;
15. bo->tex = \_\_tmp\_out.tex;
16. return;
17. // end return
18. }

那么通过临时对象的构造，便可以将其余部分的代码通过常规布局生成，避免了在普通布局和语义布局之间复杂的判断和逻辑。尽管临时变量的使用导致了代码在外观上看起来很低效，但是实际上这种极为简单的冗余代码，是非常适合LLVM这种基于SSA的优化方案的。