SASL开发者指南

# 设计概述

## 代码路径与项目结构

SASL代码路径在<salvia\_root>/sasl下。

图 1 SASL代码目录结构

SASL包含的工程包括：

* Common
  + 在SASL共享的接口
  + Token的定义
  + Diagnostic信息的定义和实现
* Parser
  + Tokenizer
  + Lexer
  + Parser
* Syntax tree
  + Syntax tree上各个节点的定义
* Semantic
  + Semantic Analyzer
  + Symbols
  + Type system and conversation
* Code\_generator
  + APIs
  + LLVM based code generator
    - Common Code generator
    - Code generator service（LLVM的封装）
    - SISD Specified Code Generator
    - SIMD Specified Code Generator
  + SASL VM（Deprecated）
* Driver
  + Static and DLL Driver
* Host
  + Shader Host（用于管线集成）

## 第三方依赖

SASL使用了boost（1.44或以上）和LLVM（3.0或以上）。如果直接使用SALVIA编译，则遵照SALVIA编译文档说明即可。

对SALVIA的其他库，SASL对eflib有实现依赖，对salviar(salvia renderer)有部分头文件依赖（主要集中在salviar的shader\_abi.h和abi\_info中）。

## 静态结构

SASL是传统的多遍编译器。

图 2 多遍编译器的工作流程

### 词法分析

SASL的词法分析分为两步，第一步将所有的代码处理成一组word，第二部将word处理成token流。这样做的原因是为了将preprocessor控制在word的处理阶段。静态结构如下：



图 3 词法分析阶段类图

code\_source和lex\_context是两个纯接口类。当调用了lexer.tokenize()后，Lexer将循环调用code\_source中的next来获得下一个word，并在lex\_context接口中获得文件、行列数等token的位置信息（严格说lex\_context是code\_source实现行列统计的一个算法），并返回一个token\_t的序列，这是语法分析时候需要的token流。

在driver中提供了一个driver\_code\_source实现了这两个接口，并且driver\_code\_source使用了Boost.Wave，提供了与C99/C++0x相同能力的preprocessor功能。

SASL的Lexer基于Boost.Spirit.Lexer开发，重新进行了封装便于第三方代码对词法进行定制。语法部分没有使用Boost.Spirit.Qi，主要是出自于编译时间的考虑。

### 语法分析

#### 基于Combinator的语法定义和语法分析器生成

SASL的语法分析采用了基于Combinator的语法分析器构造工具。VCZH有一组很好的文章<http://www.cppblog.com/vczh/archive/2008/06/06/52318.html>，《构造可配置的语法分析器》，讲述了Combinator的机制。简单的将，Combinator就是个用Compositor模式拼起来的递归下降法解析器。

Combinator的代码在generator.h和generator.cpp中。语法在grammar.h与grammar.cpp中。

SASL提供了一系列combinator的操作符，这些操作符实际上是由多个类型Parser来实现的（图 4）：‘=’为Rule Definition，类似于EBNF的‘::=’；‘|’表示分支结构，会生成一个‘A | B’ 这样的规则，会生成一个Selector Parser；‘>>’表示顺序结构；‘>’为强制顺序（如果不满足就报错）；‘\*’为重复；‘-’为不匹配规则。利用STERM(token)可指定term作为语法要素。



图 4 Parsers（部分）

因此，最终的语法可能用以下类似于EBNF的形式定义语法：

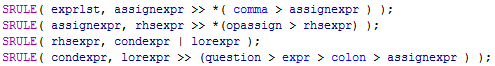


图 5 类EBNF的语法定义

#### Parser的分析结果

每一个parser都会生成一个attribute作为返回。不同类型的parser生成的attribute类型也不相同。



图 6 Attributes（部分）

每个attribute都会有一个rid（rule id），用于标示attribute是哪个rule生成的。如果attribute不是由rule生成，而是由中间parser生成，例如，如果attribute是由A >> (B | C)中的(B|C)这个Combinator表达式生成的selector\_attribute，那么这个attribute的rid恒为-1。

所有的attribute会构成一个树状结构，叶节点为terminal/token，根节点由根rule（SASL中为program）生成。这棵树我们称之为parse tree（分析树），它是语法分析的直接结果。

#### AST及其生成

AST(Abstract Syntax Tree)是Parse Tree更简明的形式[[1]](#footnote-1)。下图为SASL所建立的AST结构（部分）。代码在sasl\_syntaxtree工程下的多个文件中。



图 7 SASL的AST结构（部分）

在UML中没有反映出来的是，node上还挂载了symbol和semantic\_info的信息，用于处理语义。

从parse tree到AST的转换由syntax\_tree\_builder完成。这一转换没有什么技巧，完全靠parse tree的attribute上挂在的rule id，使用大量的if-else实现AST建立的逻辑。

### 语义分析

SASL的语义分析阶段主要有两个任务：符号表的构建和类型推导。也会生成一些代码生成阶段所需要的语义信息。实现部分的代码在semantic\_analyser中。

生成的符号（symbol）和语义信息将挂载在node上（node中的相关方法见图 8）。

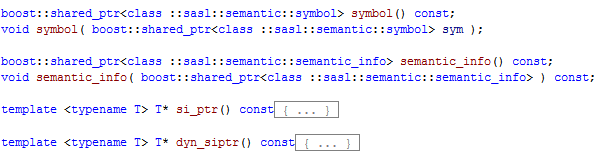


图 8 node中semantic\_info和symbol的存取函数

其中，semantic\_info是node所持有的信息，而symbol以弱引用的形式挂接在node上，并没有symbol的所有权。

另外需要注意的一点是，semantic\_analyser在分析前会先获得语法树节点的一个拷贝，然后将信息挂载在拷贝出的节点上。因此实际上semantic\_analyser分析出的结果将是源语法树的完整拷贝。这样做的原因是为了让语法树本身可以被多个semantic\_analyser进行多次分析。

#### 符号表

SASL和C++一样是支持重载的。为了在链接期，每个函数都有一个唯一名称，所以SASL也引入了name mangling机制[[2]](#footnote-2)。所以每个符号都保存了mangled name和unmangled name。查询时，一般以unmangled name为主。

Symbol本身是树状结构，因此Symbol成员变量也保存children和parent的属性。

Symbol有三个重要函数，find，find\_overloads和find\_assign\_overloads。



图 9 Symbol

find是使用mangled name在symbol tree中由下而上查找符号。find\_overloads是用的name为unmangled name。在实际的程序中有两个函数，仅有名字的那个会查找到同名函数的所有重载形式；而携带了实参、type converter和pety（SASL中的类型系统有关的类，下一节将会讲到）的函数，会根据这些信息选择合适的函数并返回。如果没有返回任何函数，那说明查找的函数不存在，或者类型不匹配；如果返回多个，那说明是有歧义。

#### 类型系统

类型系统有两个工作，类型管理和类型转换的判定。SASL中的pety实现了类型管理。它可以根据类型声明/定义/引用节点获得一个Type ID，或者从Type ID获得一个类型。

Semantic中的caster实现了类型转换功能。本质上caster是一个（源类型，目标类型，转换结果）的三元表。转换结果包括等价、隐式转换、显式转换和无转换等几种。借助于Caster的API，语义分析可以查询源类型到目标类型能否转换，以及是不是隐式转换等信息。

在进行分析前，内建的类型需要提前注册到pety中，并且caster中也会初始化内建类型的互相转换信息。

#### 语义信息

不同节点对应的语义信息保存在不同的类型中，这些类型均从semantic\_info继承，并挂载在node上。它们将在以后供代码生成使用。



图 10 semantic\_info继承结构

上图展示了semantic\_info的继承结构和使用范围最广的storage\_si的字段。

现在回顾看SASL中的semantic\_info是失败的。在最初的设计时，考虑了不同类型的节点间的语义数据的不同处；但是实际在实现的时候，又为了方便和减少类型转换，又将很多不同的信息写到了storage\_si中。以后在重构的时候可能会去掉继承层次，使用一个大的semantic\_info来表示所有语义信息。

### 代码生成

开门见山的说，针对现在的需求，SASL代码生成是个过度设计了。原先我们希望能支持多种虚拟机/后端，而实际上在目前来看，除了LLVM都不太可能有其他的后端了，包括原先在计划中的SAVM[[3]](#footnote-3)短时间内也不会有什么进展。

#### 代码生成的类及继承体系



图 11 Code Generator的继承体系

图 11为Code Generator的类及其继承结构。整个Generator类似于Bridge的手法，左侧负责将语法树在语义的指导下拆分成函数和指令，而右侧的service部分，则负责将这些拆分落实成平台相关的内容（例如调用LLVM的API生成LLVM IR）。这里使用继承主要目的是为了代码复用，因此选用了并不为大众所推荐的实现继承。

#### 不同类型代码生成类的职能

SISD和SIMD的问题，在之前的撰文中屡有说明，简单来说，SISD是类似于我们用的C的代码生成方式，而SIMD是为Pixel Shader服务的，因为ddx/ddy的需要，我们需要确保对每条指令都同时在一个4x4的block上执行，再执行下一条指令。由于指令执行方式的不同，我们也调整了数据的组织方式。在SASL中， 这样的组织方式我们称之为ABI（Application Binary Interface），因为它确实是与ABI相关的。在未来也许这个名字或许会更改为Data Layout Interface之类更加准确的描述。关于SIMD和SISD在数据组织上的不同，可以参见文档“Design of Shader ABI(CHN)”[[4]](#footnote-4)。

关于cgllvm\_vs和cgllvm\_ps的问题，在PPT“IntroToSALVIA”中，我曾经指出SASL需要处理管线的semantic信息，因而对vs和ps都设计了特殊的entry function供管线调用。这两个文件主要也是处理这entry function的问题。

#### 代码生成与cgllvm\_sctxt

在代码生成时需要使用到之前生成的一些信息。例如，在生成函数调用的时候，需要能查询到调用函数的Function\*，在生成表达式A+B的时候，分别需要A和B的value信息。这些信息保存在cgllvm\_sctxt中，并通过一个map与node关联起来。这样，借助于符号表或者节点的智能指针，便能获得对应节点所挂接的代码生成信息。当然，这也意味着代码生成的代码需要将生成的结果以cgllvm\_sctxt和对应的节点关联起来。

# 操作符与内建函数开发指南

截止SALVIA 0.3版，SASL还缺乏很多的操作符和内建函数。本章节旨在帮助希望为SASL开发操作符和内建函数的朋友。

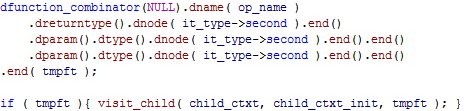
SASL操作符和内建函数的扩展开发较简洁。大致上可以归纳为三步：撰写操作符/内建函数的原型；在cgllvm\_impl中处理操作符和内建函数；在cg\_service中撰写LLVM IR的生成代码。

此处，将以绝对值函数abs为例，介绍操作符和内建函数的开发。

## 注册函数原型

原理上来说，内建函数和操作符和普通的函数声明并没有区别。只不过为了让内建函数和操作符真正是“内建”的，编译器得做一点额外的注册工作和判断工作。

在semantic\_analyser中，我们为内建函数和操作符原型的注册统一在了register\_builtin\_functions()函数下。现有的代码使用了两种风格的函数/操作符注册：



这是Combinator风格直接构造语法树节点的注册方式。为了书写方便，SASL后来又提供了新的注册方式：



比之前更加直观易用。我们提倡新加的 intrinsic/operator都采用后一种形式。这个函数的使用方法是

register\_intrinsic( child\_ctxt\_init, “<function name>”) % <param0 type> % <param1 type> % … >> <result type>;

child\_ctxt\_init 是一个固定量，用于传递语义分析所需要的上下文信息。%指定函数或操作符的参数类型，>>用于指定返回值的类型。这里参数和返回值的类型均是shared\_ptr<type\_specifier>。对于内置类型，可以用BUILTIN\_TYPE( \_sint32 ) 这个宏来获得对应的shared\_ptr<type\_specifier>。因此我们可以为float abs(float)函数撰写如下的注册代码：

register\_intrinsic(child\_ctxt\_init,“abs”) % BUILTIN\_TYPE(\_float) >> BUILTIN\_TYPE(\_float);

这样注册上函数后，semantic就能正确的解析和匹配abs函数了。如果是操作符的话，需要使用register\_function(...)来进行注册，操作符函数的名称使用operator\_name( operators op ); 获得。

## 在cgllvm\_impl中处理内建函数

接下来，我们需要将实现abs的函数体。内建函数之所以“内建”，实际上是将函数和操作符指令的生成固化在了编译器内。一般而言，我们建议将实际和LLVM有关的代码放置在cg\_service中，而在cgllvm\_impl中处理完函数的参数后调用cg\_service的相应代码即可。

先来讨论内建函数的处理。内建函数的处理代码在cgllvm\_impl::process\_intrinsics中。它的逻辑并不复杂，以下伪代码便可以描述了：

void cgllvm\_impl**::**process\_intrinsics**()**

**{**

// 1. 提取intrinsic的信息，如名字之类的

// 2. 参数匹配，并且要移调没有被调用过的指令，减少代码生成的数量

// 3. 没啥好说的，用名字来匹配intrinsic并作相应处理吧

**if** **(** intrin**->**name**()** **==** "ddx" **){**

// blah, blah, blah

**}** **else** **if** **(** /\*...\*/ **)** **{**

// blah, blah, blah

**}** **else** **if** **(** intrin**->**name**()** **==** "abs" **)** **{**

// code for abs

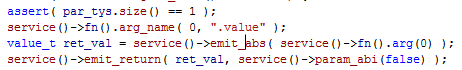
**}** **else** **{**

EFLIB\_ASSERT\_UNIMPLEMENTED**();**

**}**

**}**

我们需要把abs的函数体生成代码放在code for abs中。这部分的代码如下：



其中，service()->fn().arg\_name( 0, “.value” ); 是为了让生成的LLVM函数的参数有个可以辨识的名字；service()->fn().arg(0) 是获得第一个参数的值。emit\_abs是执行实际的abs指令生成，在接下来的过程中我们要去实现它。最后emit\_return是让函数体返回emit\_abs所生成的返回值。

接下来我们便将介绍emit\_abs的实现。

## 在cgllvm\_impl中处理操作符

操作符的处理与内建函数主要有两点区别，一是我们要把操作符函数体的生成代码放在expression的处理过程中；一个是操作符通常都是inline的，不需要return。‘add’可以作为编写操作符的良好范例。

## 编写LLVM IR的生成代码

emit\_abs有一个value\_t的参数和一个value\_t的返回值。在emit\_abs中，我们需要写一些LLVM的代码来产生表达式。

在这里我们假设大家都了解LLVM的指令[[5]](#footnote-5)，以及知道如何使用IRBuilder来生成LLVM指令[[6]](#footnote-6)。翻遍文档，LLVM并没有提供abs指令。根据IEEE，我们可以用最高位的掩码来处理它。伪代码就是：

i = bitcast f to int32

masked\_i &= 0x7FFFFFFF

abs\_f = bitcast masked\_i to float

我们将它写成LLVM的代码



实际代码要比伪代码复杂一些。一个是因为emit\_abs的参数为arg\_value，这是一个value\_t类型。我们需要load成LLVM的Value\*；另外，尽管签名中参数类型是float，但是实际对应的llvm type并不一定是float，它会会根据ABI的变化而变化。在abi\_c和abi\_llvm下，SASL的float对应的是LLVM的float，但是在abi\_package下，SASL的float对应的实际上是LLVM的<16 x float> 的类型，这是一个vector。

这也要求在写LLVM实现时，要处理多种ABI和类型的情况，并且返回值的ABI应当与输入参数相同或相容。

最后将结果封装成value\_t返回，注意参数中的vkind和abi。



## 编写测试

在代码编写完成后，我们还需要为之编写JIT测试。

实际上我们更鼓励大家先写用例，后写程序实现。SASL的测试是灰盒测试。一方面我们要求用例以黑盒测试的形式呈现，即输入一段文本，编译，执行，输出值并进行比较；另一方面我们希望在设计黑盒用例时，能尽量照顾到代码的执行路径，完成白盒测试的职责。

Test基本的编写步骤是：

* 找到<salvia>/sasl/test/repo/auto\_test/repo/question/v1a1下，新建或者修改一个\*.ss/svs/sps，编写需要测试的内容。其中ss对应着C风格普通的程序，svs对应vertex shader，sps对应pixel shader。如果测试文件是新加的，请在<salvia>/sasl/test/repo/auto\_test/repo/tests.cmake中添加你的文件，这样工程就可以跟踪测试文件的修改并及时的更新依赖。
* abs的测试内容如下。test\_abs就是我们所需要测试的函数。注意这里没有直接去测试abs的原因是SASL生成intrinsic时使用了默认的调用协议，该调用协议在windows平台上的兼容性有问题。而test\_abs作为一个普通函数，SASL对ABI进行了调整，确保了在windows下的兼容性。

float test\_abs**(**float v**)**

**{**

**return** abs**(**v**);**

**}**

* 执行一下CMake，更新下工程。
* 如果是新建的测试文件，请转到Visual Studio中，Rebuild 工程sasl\_test\_repo，以便正确的更新你刚刚写入的文件。这里其实是SASL CMake脚本的一处失误，以后会进行修正。
* 进入sasl\_test\_jit工程，打开general.cpp，在这里撰写cpp的测试代码。

在这里，我们假定你已经会基本了解Boost.Test，明确Test Case和Test Suite，熟悉BOOST\_CHECK/REQUIRE\_\*等一系列宏。

对于JIT test，每一个你编写的测试文件（ss/svs/sps）对应一个Boost Test Case。每一个test case都会遵照以下结构：

BOOST\_FIXTURE\_TEST\_CASE**(** intrinsics**,** jit\_fixture **){**

// 编译测试文件。对general模式编译的代码应该用init\_g

// 如果要编译Vertex Shader和Pixel Shader，分别用init\_vs和init\_ps。

init\_g**(**"./repo/question/v1a1/intrinsics.ss"**);**

// 定义一个本地的Function

jit\_function**<**float **(**float**)** **>** test\_abs**;**

// 获取测试文件编译出来"test\_abs"，并绑定到test\_abs上。

// 注意jit\_function的签名需要和ss文件中的签名一致。

function**(** test\_abs**,** "test\_abs" **);**

BOOST\_REQUIRE**(** test\_abs **);**

// 执行实际的测试工作

**{**

float p **=** 123.456f**;**

BOOST\_CHECK\_CLOSE**(** fabsf**(**p**),** test\_abs**(**p**),** 0.000001f **);**

float n **=** **-** p**;**

BOOST\_CHECK\_CLOSE**(** fabsf**(**n**),** test\_abs**(**n**),** 0.000001f **);**

float z **=** 0.0f**;**

BOOST\_CHECK\_CLOSE**(** fabsf**(**z**),** test\_abs**(**z**),** 0.000001f **);**

**}**

**}**

如果这个测试用例能正确通过，那么就说明所撰写的代码是正确无误的了。

1. http://en.wikipedia.org/wiki/Abstract\_syntax\_tree [↑](#footnote-ref-1)
2. 见文档Name Mangling Syntax [↑](#footnote-ref-2)
3. 因此我的《实用编译器构建指南》也只好先放一放。 [↑](#footnote-ref-3)
4. 所有未注明出处的文档，都在源代码目录的<salvia>/doc文件夹下。 [↑](#footnote-ref-4)
5. http://llvm.org/docs/LangRef.html [↑](#footnote-ref-5)
6. http://llvm.org/docs/doxygen/html/IRBuilder\_8h\_source.html [↑](#footnote-ref-6)