msysypi

Micro SYSY comPIler

基本功能

```
SysY -> Eeyore -> RISC-V
        Tigger -> RISC-V
```

特色

是用 Rust 写的。

结构

SysY -> Eeyore

先解析成 AST, 然后用类似求值的做法生成代码。

算法: 对于每个函数进行编译，处理语句的时候旁边开一个"变量作用域"栈，栈中的值是HashMap<String,VarDecl>，记录着这个作用域下声明的变量，而查找就往回找直到找到。

常量初始化语句

```
type VarScope = HashMap<String, VarDecl>;
```

在生成AST的时候可能只知道初始化表达式而不是实际的值，因此初始化的VarInit如下定义，可以包括未计算的值

```
type VarDecls = Vec<Box<VarDecl>>;
struct Decl(DeclType, VarDecls);
struct VarDecl{
    dtype: DeclType,
    name: String,
    dims: Vec<i32>,
    init: VarInit
}
type Exp = Box<Expr>;
type Con = Box<CondExpr>;
enum VarInit{
    Nil,
    I(ExprInit),
    E(ValueType)
}
// 我们需要在VarInit中表达是否完成了求值的变量
// VarInit本质上装的应该是一个高维列表，如果求完了值就是整数(i32)，因此我们构造一种通用数据结构
enum InitCont<T>{
    Nil,
    Val(T,Box<InitCont<T>>), // 该值是一个单值
    Lis(Box<InitCont<T>>, Box<InitCont<T>>) // 该值是一个列表
```

```
}
type ValueType = InitCont<i32>;
type ExprInit = InitCont<Exp>;
```

那么表达式求值的模块就应该是

```
trait Evaluatable{
    fn parseEval(&self, &mut Vec<HashMap<String,VarDecl>>) -> ValueType;
}
```

最后，高维数组的初始化列表需要把值填到位置上，这需要一个函数

```
struct posVal{
    pos: i32,
    val: i32
}
fn toPosVec(Vec<i32> dims, ValueType) -> Vec<posVal>
```

事实上我的做法是把维度和值类型一起递归下去填，

```
    fn cz(&self, a: Var, off: i32, dim:&[i32], b:&mut VStack)->VI{
        if dim.len() == 1{
            // one-dimension left that's a single value
            match self{
                InitCont::Val(t) => AR(*t).cz(LVal::SymA(a,RVal::Int(off)), b),
                InitCont::Vax(t) => panic!("dimension mismatch")
            }
        }else{
            match self{
                InitCont::Val(t) => AR(*t).cz(LVal::SymA(a,RVal::Int(off)), b),
                InitCont::Vax(t) => {
                    let da = dim[0];
                    let db = dim[1];
                    let mut ptr = 0;
                    let mut vx=VI::new();
                    let ps = |vx:&mut VI,mut t:i32|{
                        if t%db==0 {t} else {
                            let r = (t+db)%db;
                            while t < r{

 vx.push_back(Inst::Ass(LVal::SymA(a,RVal::Int(off+t*4)),RVal::Int(0)));
                                t+=1;
                            }
                            r
                        }
                    };
                    for i in t.iter(){
                        match i.as_ref(){
                            &InitCont::Val(_) => {
                                vx=mdq(vx,i.cz(a,off+ptr*4,&dim[1..],b));
                                ptr+=1;
                            },
                            &InitCont::Vax(_) => {
                                ptr = ps(&mut vx,ptr);
                                vx=mdq(vx,i.cz(a,off+ptr*4,&dim[1..],b));
                                ptr += db;
```

```
                            }
                        }
                    }
                    while ptr < da{

 vx.push_back(Inst::Ass(LVal::SymA(a,RVal::Int(off+ptr*4)),RVal::Int(0)));
                        ptr+=1;
                    }
                    vx
                }
            }
        }
    }
```

Expr

一般 Expr 都具有一个返回值 t，除非 Expr 是单个 void 函数调用。为了减少一次赋值，在生成 Expr 对应的代码时将返回值存的变量传进去；不过需要特判一下 void 函数调用，因为这个时候继续生成 `t=CallFn f_xxx` 会RE。

void 函数

void 函数可能没写 `return` 语句；解决方法很简单，在每个 void 函数结尾加一条 `return` 就行。

其它语句

变量记录如同初始化时候的`VarScope`,需要加一个符号表,Eeyore有三种变量

```
struct VScope{
    map:HashMap<String,Vmeta>,
    local:i32,
    temp:i32,
    param:i32,
    retired:VecDeque<Inst>
}
struct VStack{
    st:Vec<VScope>,
    lab:i32
}
```

用 `retired` 记录从函数内的 block 里定义的变量，到生成函数的时候一口气dump出来。

生成 Eeyore 指令的时候使用 backpatch 来处理控制流，这个实现比构成控制流图再 dfs 简单，因为 rust 没有方便好用的指针……

```
pub struct Segment{
    pub ins: VecDeque<Inst>,
    list: Vec<Vec<i32>>,
    ret: Option<RVal>
}
```

`list` 就是指向`ins`里需要backpatch的跳转指令的下标，`(true/false/break/continue)list`。本来可以写`[Vec<i32>;4]`，但是 rust 数组类型太难用了。如果对于每个分量命名则多出一万行代码，真的要死。

剩下的工作都是模式匹配。

Eeyore -> Tigger

我没有输出 Tigger，直接输出的 RISC-V Assembly，但是中间也是经过 Tigger 这一步的。

因为来不及实现寄存器分配，所以每个变量读写都会写回内存，那么只需要实现 `store LVal = Reg` 和 `load Reg = RVal`，剩下的都是 `Reg` 和 `Reg` 的操作，就平凡了。

store 和 load 其实就是大分类讨论。

```
    let load = |lv:&Ldef, gv:&Gdef, nfn:&mut ti::Fn, a:RVal, b:Reg| {
        // load a:RVal into b
        let b = b.tu();
        match a{
            RVal::Int(t) => nfn.ps(Li(b,t)),
            RVal::Sym(v) => {
                let c = lv.fnd(v);
                if let Some((i,j)) = c{
                    if j==0 {
                        // load local int
                        nfn.ps(Sld(i>>2,b))
                    }else{
                        // load local array address
                        nfn.ps(Sla(i>>2,b))
                    }
                }else{
                    let (i,j) = gv.fnd(v);
                    if j==0{
                        // load global int
                        nfn.ps(Vld(i,b))
                    }else{
                        // load global array address
                        nfn.ps(Vla(i,b))
                    }
                }
            }
        }
    };
    let store = |lv:&Ldef, gv:&Gdef, nfn:&mut ti::Fn, a:Reg, b:LVal|{
        // store a:Reg into b:LVal
        // note: extra register needed: s4, s5
        let a = a.tu();
        let S4 = s4.tu();
        let S5 = s5.tu();
        let (bvname,bind,r) = match b { LVal::Sym(t) => (t,RVal::Int(0),false),
LVal::SymA(t,a)=> (t,a,true)};
        let c = lv.fnd(bvname);
        if let Some((i,j)) = c{
            if !r{
                // local int
                nfn.ps(Sst(a,i>>2));
            }else{
                // local array
                match bind{
                    RVal::Int(t) =>{
                        if j == 0{
```

```
                        nfn.ps(Sld(i>>2,S4));
                        nfn.ps(St(S4,t,a));
                    }else{
                        // addressed with int
                        nfn.ps(Sst(a,(i+t)>>2));
                    }
                },
                RVal::Sym(t) =>{
                    if j == 0{
                        load(lv,gv,nfn,RVal::Sym(t),s5);
                        nfn.ps(Sld(i>>2,S4));
                        nfn.ps(Op(S4,S4,Add,S5));
                        nfn.ps(St(S5,0,a));
                    } else {
                        // addressed with sym
                        load(lv,gv,nfn,RVal::Sym(t),s4);
                        nfn.ps(Spa(S4,S4));
                        nfn.ps(St(S4,i,a));
                    }
                }
            }
        }
    }else{
        let (i,j) = gv.fnd(bvname);
        if !r{
            // global int
            nfn.ps(Vla(i,S4));
            nfn.ps(St(S4,0,a));
        }else{
            // global array
            match bind{
                RVal::Int(t) =>{
                    // addressed with int
                    if j == 0 {nfn.ps(Vld(i,S4));} else {nfn.ps(Vla(i,S4));}
                    nfn.ps(St(S4,t,a));
                },
                RVal::Sym(t) =>{
                    // addressed with sym
                    if j == 0 {nfn.ps(Vld(i,S4));} else {nfn.ps(Vla(i,S4));}
                    load(lv,gv,nfn,RVal::Sym(t),s5);
                    nfn.ps(Op(S4,S4,Add,S5));
                    nfn.ps(St(S4,0,a));
                }
            }
        }
    }
};
macro_rules! load{
    ($a:expr, $b:expr) => {load(&lv,&gv,&mut nfn,$a,$b)}
}
macro_rules! store{
    ($a:expr, $b:expr) => {store(&lv,&gv,&mut nfn,$a,$b)}
}
```

由于 rust 所有权的限制还不能直接写 Lambda，必须得套层宏。整体上就是大分类讨论，LVal 和 RVal 的访问模式和 RISC-V 的内存访问差好远，真难受。

Tigger -> RISC-V

这步基本上就是查表了，最大的出错点在于 Tigger 一下子要 *4 一下子不 *4。然后单个函数栈占用也可能超过 int12，所以需要自己写一份大个的。

过程中嵌入的 shenanigans

SysY -> Eeyore

多做了一遍求值，结果还无法复用，权当简化生成的 Expr 了。这部分应当可以嵌入代数变换啥的优化。

Eeyore -> RISC-V

把全局的初始化读了进来，然后直接生成 `.data` 段，写成一堆 `.word`。这样也避免了在 Tigger `main` 开头加一遍初始化。

心路历程

看到讲义大力推荐 Rust，就想着 Rust 这种 Memory safe! Pattern matching! Cross-platform compilable! 的语言真是爽得批爆，遂开始学习 Rust，并用 Rust 写这个 Project。不成熟的决定便是痛苦的开端；等我实际花了好几天时间学完了一些基础工具链后，我发现我可能已经没法回头了。

Rust 确实让这个编译器少了一万个 Segment Fault，但是这是以我在写代码的时候绞劲脑汁理解所有权、借用和绕过 borrow checker 为代价的，这部分思考占用了我太多的大脑，可能直接让我写代码的效率不止减了半。事实证明，太多在平时写 C++ 培养出来的设计模式在 Rust 中是不可用的；我完全应该一开始就写 C++。

最痛苦的部分其实来源于调试的时候，对于 C++ 的模式我已经很适应了，但是 Rust 我完全没有头绪；特别是通过模式匹配逐级下降的函数里，我甚至没法打印全局信息，完全没办法定位错误是怎么发生的。在结构体中多加个域可能就会影响结构体的性质，因此也不能随便嵌入调试信息。除了能 Copy 的结构，其它的数据调用基本都套层引用，还要考虑所有权问题，在容器内也不能随便move出去，如此种种，感觉烦透了。

甚至，Rust 的模式匹配也没有那么美好，因为递归的数据结构至少会套一层 Box，然后就又是所有权问题啥的层出不穷。

最难以忍受的便是缺乏全局的表，虽然到后面靠着 lazy_static + mutex 勉强算是搞上了，但是这相当于从一开始就 Ban 掉了一条设计思路，很多涉及到全局操作，典型的比如 scope，需要全局递归传下去；这特别是 Ban 掉了一条直接从 rule file 生成完整 translator 的路，搞得几乎必须把 AST 读经来，写上一万行重复的模式匹配代码。

然后就是始终 WA 点，又调不出来，不知道哪里漏考虑或者错考虑了情况，本地的有输出的测试是都过了的，剩下的我就真的没头绪了。

调试方法

我自己写了一点批量运行测试数据的脚本，首先是生成：

```
@echo off
for /r %%i in (*.eeyore) do (
    msysypi.exe -S %%i -o %%i.s
)
```

然后是运行

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <vector>
#include <set>
#include <string>
using namespace std;
#include <sys/types.h>
#include <dirent.h>
#define run(...) ({sprintf(z,##__VA_ARGS__); system(z);})
char z[100];
int main (void){
    DIR *dp;
    struct dirent *ep;
    dp = opendir ("./");
    set<string> st;
    if (dp != NULL){
        while (ep = readdir (dp)){
            st.insert(ep->d_name);
        }
        (void) closedir (dp);
    }
    else
        perror ("Couldn't open the directory");

    for(auto t: st){
        int tr = t.length();
        const int ql = 9;
        if(tr>ql && t.substr(tr-ql,ql) == ".eeyore.S"){
            string q = t.substr(0,tr-ql);
            string in = q + ".in";
            string ou = q + ".out";
            string an = q + ".ans";
            bool inExist = st.count(in);
            int ret = 0;
            int tz = run("riscv32-unknown-linux-gnu-gcc %s -o ou -L/root/sysy-
runtime-lib -lsysy -static",t.c_str());
            if(tz){
                printf("CTE at %s\n",q.c_str());
                getchar();
            }
            if(inExist)
                sprintf(z,"qemu-riscv32-static ou < %s >
%s",in.c_str(),an.c_str());
            else
                sprintf(z,"qemu-riscv32-static ou > %s",an.c_str());
            int zr = system(z);
            printf("\nReturnValue: %d\n",zr/256);
            sprintf(z,"echo %d >> %s",zr/256,an.c_str());
            system(z);
            sprintf(z,"diff -w %s %s",ou.c_str(),an.c_str());
            if(system(z)){
                printf("Difference found at testpoint %s\n",q.c_str());
                getchar();
            }
        }
    }
```

```
    return 0;
}
```

非常好用。然后便是对于每个出错的数据查看 codegen 因为什么原因出错。但是 open test case 测完了，我就没主意了。

工具使用

Compiler compiler

我使用的是 lalrpop，lalrpop 最大的优点就是规则文件清晰好读，而且语法错误提示很好，比如 LR(1) reduce conflict 它会给出具体的例子和 Parse tree。lalrpop 最大的问题可能是生成的表太大了，SysY 将近 8000 行，Eeyore 和 Tigger 都有 3000 行。

```
use crate::sysY::ast::
{CompUnit,DeclType,FunctType,UOper,Oper,VarDecls,Decl,VarDecl,Exp,Con,VarInit,In
itCont,ExprInit,Func,BlockItem,Block,Stmt,LVal,Expr,CondExpr};
grammar;
match{
    "&&",
    "||",
    "==",
    "!=",
    "<=",
    ">=",
    "<",
    ">",
    "+",
    "-",
    "*",
    "/",
    "%",
    "!",
    ",",
    "{",
    "}",
    "[",
    "]",
    "(",
    ")",
    ";",
    "=",
    "const",
    "int",
    "void",
    "while",
    "if",
    "break",
    "continue",
    "return",
    "else"
}else{
    r"[a-zA-Z_][a-zA-Z0-9_]*", // keyword / identifier
    r"(0[xX][0-9a-fA-F]*)", // hex number
    r"(0[0-7]*)", // oct number
    r"([1-9][0-9]*)", // decimal
```

```
    r"//[^\n\r]*[\n\r]*" => { }, // comment
    r"/\*([^\*]*\*+[^\*/])*([^\*]*\*+|[^\*])*\*/" => { }, // comment
    r"\s+" => {} // space
}
/// Macro
Comma<T>: Vec<T> = { // (1)
    <mut v:(<T> ",")*> <e:T?> => match e { // (2)
        None => v,
        Some(e) => {
            v.push(e);
            v
        }
    }
};
Oper<Op,T>:Box<Expr> = {
    Oper<Op,T> Op T => Box::new(Expr::Op(<>)),
    T
};
/// Program
pub Program:Vec<Box<CompUnit>> = {
    CompUnit+
};
CompUnit:Box<CompUnit> = {
    Decl => Box::new(CompUnit::Decl(<>)),
    Func => Box::new(CompUnit::Func(<>))
};
/// Decls
Decl:Box<Decl> = {
    ConstDecl => Box::new(Decl(DeclType::Const,<>)),
    VarDecl => Box::new(Decl(DeclType::Var,<>))
};
ConstDecl:VarDecls = {
    "const" "int" <cdef:Comma<ConstDef>> ";" => cdef
};
VarDecl:VarDecls = {
    "int" <vdef:Comma<VarDef>> ";" => vdef
};
Dims = ("[" <Exp> "]")*;
ConstDef : Box<VarDecl> = {
    <name:Ident> <dims:Dims> "=" <init:InitVal> => Box::new(VarDecl{
        dtype:DeclType::Const, name, dims, init:VarInit::I(init)
    })
};
VarDef : Box<VarDecl> = {
    <name:Ident> <dims:Dims> <init:("=" <InitVal>)?> => Box::new(VarDecl{
        dtype:DeclType::Var, name, dims, init:match init {
            None => VarInit::Nil,
            Some(e) => VarInit::I(e)
        }
    })
};
InitVal : ExprInit = {
    Exp => Box::new(InitCont::Val(<>)),
    "{" <Comma<InitVal>> "}" => Box::new(InitCont::Vax(<>))
};
/// Functions
Func : Box<Func> = {
```

```
    "int" <name:Ident> "(" <param:Comma<FuncParam>> ")" <body:Block> =>
Box::new(Func{<>,ret:FunctType::Int}),
    "void" <name:Ident> "(" <param:Comma<FuncParam>> ")" <body:Block> =>
Box::new(Func{<>,ret:FunctType::Void})
}
FuncParam : Box<VarDecl> = {
    "int" <name:Ident> <dims:ParDim?> => Box::new(VarDecl{
        dtype:DeclType::Var, name, dims:match dims{
            None => Vec::new(),
            Some(e) => e
        },init:VarInit::Nil
    })
};
ParDim : Vec<Exp> = {
    "[" "]" <mut dim:Dims> => {
        dim.insert(0,Box::new(Expr::Nil));
        dim
    }
};
/// Block
Block : Block = "{" <BlockItem*> "}";
BlockItem : Box<BlockItem> = {
    Decl => Box::new(BlockItem::Decl(<>)),
    Stmt => Box::new(BlockItem::Stmt(<>))
};
Stmt : Box<Stmt> = {
    OSt,
    CSt
}
OSt : Box<Stmt> = {
    "if" "(" <Cond> ")" <Stmt> => Stmt::If1(<>),
    "if" "(" <Cond> ")" <CSt> "else" <OSt> => Stmt::If2(<>),
    "while" "(" <Cond> ")" <OSt> => Stmt::Whi(<>)
};
CSt : Box<Stmt> = {
    SSt => Box::new(<>),
    "if" "(" <Cond> ")" <CSt> "else" <CSt> => Stmt::If2(<>),
    "while" "(" <Cond> ")" <CSt> => Stmt::Whi(<>)
}
SSt : Stmt = {
    <LVal> "=" <Exp> ";" => Stmt::Assign(<>),
    <Exp?> ";" => Stmt::Expr(match <>{
        None => Box::new(Expr::Nil),
        Some(e) => e
    }),
    Block => Stmt::Block(<>),
    "break" ";" => Stmt::Break,
    "continue" ";" => Stmt::Continue,
    "return" <Exp?> ";" => Stmt::Ret(match <>{
        None => Box::new(Expr::Nil),
        Some(e) => e
    })
};
/// Expression
Exp = AddExp;
Cond = LOrExp;
LVal:LVal = <name:Ident> <ind:Dims> => LVal{<>};
Prim:Exp = {
```

```
    "(" <Exp> ")",
    LVal => Box::new(Expr::LVal(<>)),
    Number => Box::new(Expr::Num(<>))
};
UnaryExp:Exp = {
    Prim,
    <Ident> "(" <Comma<Exp>> ")" => Box::new(Expr::FnCall(<>)),
    <UnaryOp> <UnaryExp> => Box::new(Expr::UOp(<>))
};
MulExp = Oper<MulOp,UnaryExp>;
AddExp = Oper<AddOp,MulExp>;
RelExp = Oper<RelOp,AddExp>;
EqExp  = Oper<EqOp,RelExp>;
ConEqExp:Con = EqExp => Box::new(CondExpr::Comp(<>));
LAndExp:Con = {
    ConEqExp,
    <LAndExp> "&&" <ConEqExp> => Box::new(CondExpr::And(<>))
};
LOrExp:Con = {
    LAndExp,
    <LOrExp> "||" <LAndExp> => Box::new(CondExpr::Or(<>))
};
//// Tokens
UnaryOp:UOper = {
    "+" => UOper::Pos,
    "-" => UOper::Neg,
    "!" => UOper::Not
};
MulOp:Oper = {
    "*" => Oper::Mul,
    "/" => Oper::Div,
    "%" => Oper::Mod
}
AddOp:Oper = {
    "+" => Oper::Add,
    "-" => Oper::Sub
};
RelOp:Oper = {
    "<" => Oper::Lt,
    ">" => Oper::Gt,
    "<="=> Oper::Le,
    ">="=> Oper::Ge
};
EqOp:Oper = {
    "=="=> Oper::Eq,
    "!="=> Oper::Ne
};
Ident:String = {
    r"[a-zA-Z_][a-zA-Z0-9_]*" => String::from(<>)
};
Number:i32 = {
    r"(0[xX][0-9a-fA-F]*)" => i64::from_str_radix(&<>[2..],16).unwrap() as i32,
    r"(0[0-7]*)" => i64::from_str_radix(&<>,8).unwrap() as i32,
    r"([1-9][0-9]*)" => i64::from_str_radix(<>,10).unwrap() as i32
};
```

lalrpop的语法规则长得像这样，非常像 rust 本身，所有的 Action 都有明确的语义和类型，可读性很好。

MiniVM

MiniVM 跑得比谁都快，巨好评。不过我没有用 MiniVM 自带的调试工具。

RISC-V 模拟器

编译链接巨慢，运行也不快，难顶。

心得体会

不要拿自己之前没学过的语言写大工程。由于是第一次写 Rust，踏了太多的坑，在实现编译器之外的方面搞得很累。

有些遗憾没有亲手实现编译器优化，其实感觉这部分是比较有趣的，但是时间不够写不出来。

建议

希望spec能写得更加具有指导性一些，有一些难以阅读的地方，比如常量数组初始化语法那块很难理解；还有就是比如说明void函数可以不通过return语句返回之类，这些在语法上成立但是在语义上并不明确成立的要素。