CNC 2012
Programming Languages
Professor Sherri Shulman
Final
Owen Meyer

1.]

Abbreviated Concrete Syntax to add Structs to Clite

Program → **{ StructDefinitions Declarations Statements }**
StructDefinitions → **{ StructDefinition }**
StructDefinition → struct **Identifier** { **{ Declarations }** }
Declaration → … | **StructDeclaration**
StructDeclaration → struct **Identifier Identifier { , Identifier }**
Type → … | struct
Assignment → … | **Identifier . Identifier { . Identifier } = Expression**
Expression → …
Primary → … | **Identifier . Identifier { . Identifier }**

Abbreviated Abstract Syntax to add Structs to Clite

Program = **StructDefinitions Declarations Statements**
Declaration = … | **StructDecl**
StructDecl = **Variable** v **; StructType** t
Type = … | struct
StructType = **String** structName
Fields = **Declarations**
StructDefinitions = **StructDefinition ***
StructDefinition = **String** structName **; Fields** fs
StructRef = **String** id **; VariableRef** ref
VariableRef = … | **StructRef**
StructRef = … | **String** id **; VariableRef** vr

Note:
Included in my submission is my haskell version of these modifications. The haskell version, handles things differently with more or less the same ideas. It also includes modifications for put() statements with concatenate – able strings. There are some samples of simple Clite code to show the outcomes as far as parsing to my concrete and abstract syntax choices. The configurations on these pages are geared towards the java project. I realize that there are other ways (some better) to go. For example my representation of strings doesn't account for implementation using arrays or anything like that. If I call the struct's name it's **structType** and the reference's name the **refType**: I can easily handle the issues of getting and comparing types in the haskell version by adding to the abstract syntax Type a (Struct StructType) category. This does not work in the java version as the expansion doesn't jibe with how it handles types. Below is the java approach that I took, by adding in two extra hashMaps.

2.]

With the following ideas I believe I can provide adequate type checking:
Struct Definition: < name , fields > (String, Declarations) //as above, so below..
Struct Declaration: < v , structName > (Variable, StructType) //the structName is it's "struct type"
Struct Reference: < id , reference > (String, VariableRef)
//a reference can be a variable, an array, or another struct
a TypeMap is as in the java Clite project: a hashMap of < id , type >
additionally I add:
a StructDefMap of Struct Definitions and a StructTypeMap of Struct Declarations.
The StructDefMap (SDM) is used to get the refType of the actual value. [hashMap of Stuct Defs.]
The StructTypeMap (STM) is used to get the structName of the reference. [hashMap of Struct Decs.]
//the STM is needed because the existing TypeMap, as it is designed, cannot handle the structType.

Here is a rough sketch of changes to typeOf(..) to retrieve the refType for some stuctRef:

```
public Type typeOf (Expression e, TypeMap tm, StructDefMap sdm, StructTypeMap stm){
    …
    if (e instanceof VariableRef ){
        …
        if (e instanceof StructRef){
            StructRef sr = (StructRef) e;
            StructType st = structTypeOf(sr.id(), stm);
            check(sdm.containsKey(st) , "error: undefined struct");
            TypeMap tm' = typing(sdm.get(st) );      //makes a type map from sr's type's fields
            return typeOf (sr.ref( ) , tm' , sdm, stm );
        …
}//admittedly annoying to have those three hashMaps. Some better way?
```

Here is a sketch of structTypeOf:

```
public StructType structTypeOf(String id, StructTypeMap stm){
    check (stm.containsKey(id), "error: undeclared struct reference");
    return (StructType) stm.get(id);
}
```

Left Hand Side Assignment type checking:

1. ...
2. An assignment is valid if the following are all true:
    a. If target is Variable or ArrayRef, check that it is declared.
       If target is Variable and it's type is struct check that it's structType is defined*
       If target is structRef:
           -Check if the target's id is declared.
           -Check if the target's structType is defined.
           -Check that the target's variableRef is valid. **
    b. Check if the source is Valid. (Expressions)
    c. → d. // these are handled by the existing system as the new typeOf() gets the refType.

…

* This handles the case of a struct to struct assignment.

** Calling V (< target > . vr , tm , sdm , stm) //where < target > . vr is an expression (the variableRef field of a structRef); tm is a typeMap generated from the Fields of < target > (found in the SDM). – Both V for statements and expressions check a variable to be listed in the typeMap, while expressions reports undeclared variables and statements undefined target in assignment, so this doesn't quite work,

but can be easily set up to handle the recursive case where the < target > . vr is another structRef.

Right Hand Side Assignment type checking (Expressions):

…

2. If source is Variable or ArrayRef, check that it's id appears in the type map.
If source is StructRef, check that it's id appears in the type map and it's structType appears in the StructDefMap and that it's variableRef is valid.

…

3.]

I am choosing to find two structs equivalent when they both have the same structType. Just look up their structTypes and compare 'em. It's just that simple.

4.]

All declarations must have unique names and all declaration types must be an element of {int, float, bool, char}.

5.]

All declarations must have unique names and all declaration types must be an element of {int, float, bool, char, struct}. If a declaration has type struct then it's stuctType must be defined in the struct definitions.

7.] (out of sequence)

Given the following two samples:

```
struct s1 {
        int ate;
        int bye;
}
struct s2 {
        int ate;
        int bye;
        float cry;
}
struct s1 create;
struct s2 consume;
```

Struct s2 is a subset of s1 as it is less general. The less general struct can be cast to the more general case. This involves structType of the struct in question. This information cannot be found in a struct reference, but can be found in the STM. I am not clear yet about how this might happen dynamically (if typeMaps are used during runtime). An example of correct casting of a stuct: [create = struct s1 (consume);]. The meaning would depend on Clite semantics, whether this would entail deep or shallow copy. Casting the opposite direction would result in a type error as the cry field could not be copied to an s1 struct. I have not provided for struct casting in my EBNF, though this example follows the form of the existing cast system with the additional Identifier token.

6.]

Abbreviated Concrete Syntax to add put statements to Clite

Statement → … | **PutStmt**
PutStmt → put (**Primary**)
Primary → **Identifier [** [**Expression**] **]** { <++< **Primary** } | **Literal** { <++< **Primary** }
Literal → … | **String**
String → … | " { Letter | Digit } "

Abbreviated Abstract Syntax to add put statements to Clite

Type = … | **STRING**
Statement = … | **Put**
Put = **Expression** str
Expr …
Operator = … | **StringOp**
StringOp = <++<
Value = … | **StringVal**
StringVal = **String** val

(again: a little bit different than the attached haskell version, but similar idea, geared for java version)

Rule 6.4:
...
6. A put statement is valid if it's Expression is valid and if it's Expression type is STRING.

Rule 6.5:
…
3. …
e. if op is StringOp then both expressions must be type STRING.

V for statements

```
…
    if (s instanceof Put) {
        Put p = (Put) s;
        V (p.str, tm)
    }
```

V for expressions

```
…
    if (e instanceof Binary){
        …
        if (b.op.stringOp( ) ){
            check (typ1 == STRING && typ2 == STRING, "type error for"+b.op);
```

typeOf( ) will have to have a type return for string literals.

8.]
a.)
“
Each of the four value sets includes not only the finite nonzero values that are ascribed to it above, but also NaN values and the four values positive zero, negative zero, positive infinity, and negative infinity.
...
Except for NaN, floating-point values are ordered; arranged from smallest to largest, they are negative infinity, negative finite nonzero values, positive and negative zero, positive finite nonzero values, and positive infinity.
...
Positive zero and negative zero compare equal; thus the result of the expression 0.0==-0.0 is true and

the result of 0.0 > -0.0 is false. But other operations can distinguish positive and negative zero; for example,1.0 / 0.0 has the value positive infinity, while the value of 1.0/-0.0 is negative infinity.
...
A floating-point operation that overflows produces a signed infinity.
“
I recognize the importance of these publications for the long term using/learning of programming languages. I am reminded of various building code publications, which once one is familiar with using them, become an indispensable source of answers to the interminable stream of questions that arise from applying plans to materials.
b.)
float j = infinity: Positive zero.
“
• Division of a finite value by an infinity results in a signed zero. The sign is determined by the rule stated above.” (“That is, the quotient produced for operands *n* and *d* that are integers after binary numeric promotion (§5.6.2) is an integer value *q* whose magnitude is as large as possible while satisfying $|d| \cdot |q| \leq |n|$; moreover, *q* is positive when $|n| \geq |d|$ and *n* and *d* have the same sign, but *q* is negative when $|n| \geq |d|$ and *n* and *d* have opposite signs.”)

int j = 0: ArithmeticException is thrown.
“...if the value of the divisor in an integer division is `0`, then an `ArithmeticException` is thrown.”

float j = 0: Positive infinity.
“• Division of a nonzero finite value by a zero results in a signed infinity. The sign is determined by the rule stated above.”

largest IEEE representable with available bitsize > float j > 0 : long answer / many cases.
“
• In the remaining cases, where neither an infinity nor NaN is involved, the exact mathematical quotient is computed. A floating-point value set is then chosen:
◆ If the division expression is FP-strict (§15.4):
❖ If the type of the division expression is `float`, then the float value set must be chosen.
❖ If the type of the division expression is `double`, then the double value set must be chosen.
◆ If the division expression is not FP-strict:
❖ If the type of the division expression is `float`, then either the float value set or the float-extended-exponent value set may be chosen, at the whim of the implementation.
❖ If the type of the division expression is `double`, then either the double value set or the double-extended-exponent value set may be chosen, at the whim of the implementation.
Next, a value must be chosen from the chosen value set to represent the quotient. If the magnitude of the quotient is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign. Otherwise, the quotient is rounded to the nearest value in the chosen value set using

IEEE 754 round-to-nearest mode. The Java programming language requires
support of gradual underflow as defined by IEEE 754 (§4.2.4).
Despite the fact that overflow, underflow, division by zero, or loss of information
may occur, evaluation of a floating-point division operator / never throws a
run-time exception"

and a few other cases...

>++> <++<

Whew!....

P.S.
I just flashed on how to avoid the extra hashMap (no need for STM) just have structType be a subclass of Type; duh... shows what a newbie I am. I still nee the SDM though..

```
public class StructType extends Type{
    private String structName;
    StructType (String sn){
        structName = sn;
        super.("struct");
    }
}

class Type {
    final static Type INT = new Type("int");
    …

    private String id;

    private Type (String t) { id = t; }

    public String toString (){
        return id;
    }
}
```

With this setup, the existing TypeMap can handle this new type.

```
public static TypeMap typing (Declarations d) {
    TypeMap map = new TypeMap();
    for (Declaration di : d){
        if (di instanceof VariableDecl)
        map.put ( ((VariableDecl)di).v, ((VariableDecl)di).t);
        else if  (di instanceof ArrayDecl){
                map.put ( ((ArrayDecl)di).v, ((ArrayDecl)di).t);
        }
        else if (di instanceof StructDecl){
                StructDecl id = (StrucDecl)di;
                String var = id.v;
                StructType st = new StructType(id.t);
                map.put ( var, (Type)st );
        }
}
```

The StuctType is cast to a Type to get into a TypeMap, but can be cast back to a StructType when it's id field is found to contain “struct” (the one Type object that there can be more than one of).