The Type System of PLAAY

Theodore S Norvell
Electrical and Computer Engineering
Memorial University

September 3, 2018

Abstract

The PLAAY's the thing — William Shakespear.

Contents

1 Quick introduction to the PLAAY language **3**
1.1 Expressions . . . 3
1.1.1 Abstract syntax . . . 3
1.1.2 Informal meaning . . . 4
1.1.3 Implicit fetches . . . 6
1.2 Notation . . . 7

2 An informal introduction to PLAAY's type system **7**
2.1 Values . . . 8
2.2 Extraordinary outcomes . . . 8
2.3 The bottom type . . . 9
2.4 Subtypes and soundness . . . 9
2.5 Meets and joins and top . . . 10
2.6 Subtypes of primitive types . . . 11
2.7 Tuple types . . . 11
2.8 Function types . . . 13
2.8.1 Meets of function types . . . 15
2.8.2 Joins of function types . . . 17
2.9 Field types . . . 17
2.10 Locations . . . 17
2.10.1 Some consequences . . . 19

2.10.2 Aliasing 20
2.11 Classes 22
2.12 Named types and recursion 22

3 Formalizing the type system **22**
3.1 Abstract syntax for types 22
3.2 Types 23
3.3 The length of types 23
3.4 Subtypes 24
3.5 Values and semantics of types 26
3.5.1 Values 27
3.5.2 Semantics of types 28
3.5.3 Soundness 29
3.6 Dynamic Type Checking 29
3.7 Static Typing of expressions 30
3.7.1 Typing rules 31
3.7.2 Soundness of the typing rules 35

List of varables and constants

e expression

a address

b boolean: either true or false.

c constness: either con or loc.

d declaration

e expression

f field

h context: either L or R

i, j identifier

k natural numbers

n numbers

m members: Either expressions or declarations

p primitive types

q type factors

r, s type terms

t, u, v types

x, y, z values

1 Quick introduction to the PLAAY language

1.1 Expressions

1.1.1 Abstract syntax

PLAAY is a visual programming language in which the programmer directly manipulates the abstract syntax of the program. Mostly this document is about the type system, but this section contains a quick introduction to some of the abstract syntax for expressions and declarations. Expressions e include

$$
\begin{array}{rcl}
e & : & := \mathsf{numberLiteral}\,[i] \\
 & | & \mathsf{stringLiteral}[i] \\
 & | & \mathsf{nullLiteral} \\
 & | & \mathsf{boolLiteral}[b] \\
 & | & \mathsf{var}[i] \\
 & | & \mathsf{dot}\,[i]\,(e) \\
 & | & \mathsf{tuple}\,(e_0, e_1, \ldots, e_{n-1}) \\
 & | & \mathsf{call}(e_0, e_1, \ldots, e_{n-1}) \\
 & | & \mathsf{callVar}[i](e_1, \ldots, e_{n-1}) \\
 & | & \lambda \\
 & | & \mathsf{if}(e, seq, seq) \\
 & | & \mathsf{while}\,(e, seq) \\
 & | & \mathsf{assign}(e_0, e_1) \\
 & | & \mathsf{loc}\,(e) \\
 & | & \mathsf{objectLiteral}\,\left(seqMember_0, seqMember_1, \ldots, seqMember_{n-1}\right) \\
 & | & \mathsf{expPH} \\
\lambda & : & := \mathsf{lambda}(params, optType, seq)
\end{array}
$$

where i ranges over strings. We have the following additional forms of trees

$$\begin{array}{rcl} \mathit{params} & : & := \mathsf{params}\,(d_0, d_1, ..., d_{n-1}) \\ d & : & := \mathsf{varDecl}[c](\mathsf{var}[i], \mathit{optType}, \mathit{optExp}) \\ c & : & := \mathsf{loc} \mid \mathsf{con} \\ \mathit{optType} & : & := \mathit{type} \mid \mathsf{noType} \\ \mathit{optExp} & : & := e \mid \mathsf{noExp} \\ \mathit{seq} & : & := \mathsf{expSeq}\,(m_0, m_1, \ldots, m_{n-1}) \\ m & : & := e \mid d \end{array}$$

The abstract syntax of of *type* is given in Section 3.1.

The set of expressions above roughly corresponds to the language as implemented in the summer of 2018.

1.1.2 Informal meaning

The meanings of most of these constructs are fairly straightforward:

- number, string, boolean, and null literal evaluate to number, string, boolean and null values respectively.
- $\mathsf{var}[i]$ is an occurrence of a variable. Its value is obtained from the run-time stack. If the variable has not been initialized, it is an error —but not a type error.
- Evaluating $\mathsf{dot}\,[i]\,(e)$ accesses a field of an object.
- Evaluating $\mathsf{tuple}\,(e_0, e_1, \ldots, e_{n-1})$ creates a tuple of values. In the case where $n = 1$, its value is the same as the value of its operand.
- Evaluating $\mathsf{call}(e_0, e_1, \ldots, e_{n-1})$ applies a function to an argument. If the argument is not in the domain of the function, it is a type error. When e_0 evaluates to a closure with a declared result type, the value of the function must belong to that type or is is a type error.
- $\mathsf{callVar}[i](e_1, \ldots, e_{n-1})$[1] is just an abbreviation for $\mathsf{call}(\mathsf{var}[i], e_1, \ldots, e_{n-1})$.
- Evaluating $\mathsf{lambda}(\mathit{params}, \mathit{optType}, \mathit{seq})$ creates a closure. The optional type is the return type. If the return type is missing, the return type is inferred from the sequence which is its body.

[1] Currently the implementation calls this callWorld. But that is a misnomer and will be changed.

- Evaluating $\mathsf{assign}(e_0, e_1)$ changes the value of one or more locations in the store. The value of an assign is the empty tuple. The expression on the left should evaluate to an location value or a tuple of location values or so on. The expression on the right should evaluate to an expression of the appropriate type.This distinction between L contexts and R contexts is discussed more below. Storing a value that does not belong to the location's type is a type error.

- Evaluating $\mathsf{if}(e, seq, seq)$ evaluates e and then evaluates one of two expression sequences, based on the value of e. If e evaluates to neither true nor false, it is a type error. Its value is the value of the chosen sequence.

- Evaluating $\mathsf{while}\,(e, seq)$ evaluates e and then the sequence repeatedly until the expression evaluates to false. If e evaluates to neither true nor false, it is a type error. The value of a loop that terminates is the empty tuple.

- Evaluating $\mathsf{loc}\,(e)$ evaluates e as if it were on the left side of an assignment. That is e is evaluated in an L context. The value of $\mathsf{loc}\,(e)$ is the same as the value of its operand evaluated in a L context.

- Evaluating $\mathsf{objectLiteral}\,(seqMember_0, seqMember_1, \ldots, seqMember_{n-1})$ evaluates a sequence of expressions and declarations (as below). Its value is the stack frame thus created.

- expPH is an expression place-holder. It is used during editing to hold the place for expressions yet to be added. Evaluating a place-holder is an error.

- $\mathsf{expSeq}\,(seqMember_0, seqMember_1, \ldots, seqMember_{n-1})$ is evaluated as follows. First a new stack frame is made, based on the variable declarations. Initially the variables are uninitialized (or, if you prefer, initialized to the error value). Second the sequence members are evaluated from left to right. As the variable declarations are evaluated, the variables become initialized. The value of the sequence is the value of its last member. If there are zero members, then the value is the empty tuple. PLAAY does not have a declaration before use rule in order to allow for mutually recursive function definitions.

- $\mathsf{varDecl}[c](\mathsf{var}[i], optType, optExp)$ declares a new variable. Evaluating a variable declaration initializes the variable. If the type is not given, it is inferred from the expression. If neither is given, the type defaults to the top type. If c is con, the value of the variable is the value of the expression (which must not be missing). If c is loc, a new location is allocated and the value of the variable is the location; in this case the expression (if given) is used to initialize the location. It is a type error if the value of the expression does not belong to the type of the field. The value of a declaration is the empty tuple.

- Variable declarations can also appear in parameter lists. In this case, the value for the variable comes from the corresponding argument in the case of con variable declarations. For loc declarations, a new location is created as part of the call and the argument is used to initialize that location. Initialization expressions can be used to provide default arguments.

1.1.3 Implicit fetches

For certain of the expressions listed above, the evaluation of the expression is immediately followed by an 'implicit fetch'. An fetch operation converts a location to its contents. Implicit fetches only happen when the value is a location. For example consider an assignment

$$\mathsf{assign}(\mathsf{var}[\mathsf{x}], \mathsf{callVar}[+](\mathsf{var}[\mathsf{y}], \mathsf{numberLiteral}[1]))$$

Suppose the value of var[x] is a location. There is no implicit fetch because the expression occurs in an L context. (The left side of an assignment is an L context.) The expression var[y] occurs in an R context. (The operands of a call are R contexts.) If var[y] represents a location, that location is implicitly fetched and the contents of the location will be the first argument to the function. Implicit fetching only happens if three things are true:

- The expression is a var expression, a dot expression, a call expression or a callVar expression.
- The expression occurs in an R context.
- The value of the expression is a location.

Implicit fetcing is done only once: if the result of an implicit fetch is a location, then that is the result of the expression. For example, suppose x and y are both variables whose values are locations that hold locations that hold numbers. Consider assign(var[x], var[y]). The left side will evaluate to a location that holds a location that holds a number. The right side will evaluate to a value of the same type, but as implicit fetch will change that to a location that holds a number. Thus the assignment will work without error.

As mentioned above, implicit fetches can be suppressed with the loc operator; this is in fact its only purpose. Supppose that variable x represents a location that holds a location that holds a number, as above, and variable z represents a location that holds a number. Consider:

$$\mathsf{assign}(\mathsf{var}[\mathsf{x}], \mathsf{loc}(\mathsf{var}[\mathsf{z}]))$$

here there is no implicit fetch, so the types of the left and right hand operands will be as in the previous example. The assignment will work without error.

A fetch can be forced by calling the identity function. For example, if, as above, variable x represents a location that holds a location that holds a number, we could write

$$\mathsf{assign}(\mathsf{callVar}[\mathsf{id}](\mathsf{var}[\mathsf{x}]), \mathsf{numberLiteral}[42])$$

Since the operand of a call is an R context, var[x] is implicitly fetched from; the function's argument (and hence its value) is a location that holds a number. The call itself occurs in an L context and so there is no implicit fetch after the call returns. Thus the left side will evaluate to a location that holds a number. The assignment will work without error.

1.2 Notation

Since the abstract syntax can be verbose, I'll sometimes use a sort of ad-hoc pseudocode. E.g.,

$$\lambda\ \mathsf{y} : \mathsf{Loc}[\mathsf{Int}] \cdot \{\ \mathsf{y} := 12\ \}$$

instead of the abstract syntax

```
lambda( paramList( vardecl[con]( var[y],
                                 locType(intType),
                                 noExp ) )
        noType,
        exprSeq(assign( var[y],
                        numberLiteral[12] )) )
```

2 An informal introduction to PLAAY's type system

In PLAAY, values are organized into types. When variables have declared types, we try to ensure that only values of the given type are associated to them. This is done through a combination of static (i.e., edit-time[2]) and dynamic (i.e., run-time) checks. The basic principle is to report problems to the user as soon as possible. Thus we report errors at edit-time when possible. Errors reported at edit-time may be ignored by the user. That is, the user may run the program even if errors are being reported. So we still need run-time checks even for errors that would normally be caught at compile time. This is in contrast to languages like Java —where there is limited need for run-time type checking— and C —where run-time checking is arguably needed, but isn't done.

[2]The traditional term is "compile-time". But PLAAY doesn't have a compilation step. Instead errors are reported through the editor when program is being edited.

2.1 Values

The following is list of the values in the PLAAY language.

- String values – Sequences of 0 or more characters
- Boolean values.
- Number values.
- The null value — for indicating nullity.
- The empty tuple.
- Tuples of 2 or more values.
- Object values — Objects comprise named fields that refer to values.
- Closure values — These are the result of evaluating lambda expressions.
- Built-in function values —These represent functions implemented in other languages.
- Locations — locations are values that hold references to other values.

Let's start the type system by supposing that we have types String, Number, Bool, Null. I'll call these types **primitive** types. We'll leave tuples, objects, closures, built-in functions, and locations to later.

We will use $\text{Val}(t)$ to represent the set of values[3] that belong to type t. For example

$$\{0, 1, 2, 3.14, -42, \ldots\} \subseteq \text{Val}(\mathsf{Number})$$

2.2 Extraordinary outcomes

Usually a computation results in some value. But there are other possible **outcomes**.

- The error outcome ▼, representing a computation that has gone wrong. For example $12 \div 0 = ▼$.
- A nontermination outcome ▲, representing the result of a computation that never finishes.

[3]The idea that types correspond to sets is likely naive. However, it serves as a starting point.

We want division, for example, to be such that $x \div y : \mathsf{Number}$ when $x, y : \mathsf{Number}$. Since $12, 0 : \mathsf{Number}$.

Similarly, we can argue that ▼ should be in the image of every type. We want the type of any nonempty expression sequence to be the type of its last expression. Consider

$$\mathsf{ExpSeq}(\mathsf{while}(\mathsf{true}, \mathsf{ExpSeq}()), 12)$$

However we don't say that ▼ or ▲ are values of type Number, since they aren't values at all; they are outcomes. We will distinguish between the values of type, written as $\mathrm{Val}(t)$ and the possible outcomes of a computation of type t, which we call the type's image, $\mathrm{Im}(t)$. The relationship is simply that

$$\mathrm{Im}(t) = \mathrm{Val}(t) \cup \{\blacktriangledown, \blacktriangle\}$$

2.3 The bottom type

The minimal type we can form is "bottom" notated $\bot$. This type's image contains only ▲ and ▼. We have

$$\begin{aligned} \mathrm{Val}(\bot) &= \emptyset \\ \mathrm{Val}(\mathsf{Bool}) &= \{\mathsf{true}, \mathsf{false}\} \end{aligned}$$

and

$$\mathrm{Val}(\mathsf{Number}) = \{0, 1, 2, 3.14, -42, \ldots\}$$

and so on.

2.4 Subtypes and soundness

We write $t <: u$ to mean that t **is a subtype of** u. We can define this relation on types so that $\bot <: \mathsf{Number}$. In general we want that, for any types t and u

$$(t <: u) \Rightarrow (\mathrm{Val}(t) \subseteq \mathrm{Val}(u)) \qquad \text{(Soundness)}$$

Furthermore, we want that, for any types t, u, and v,

$$t <: t \qquad \text{(Reflexivity)}$$

$$(t <: u) \wedge (u <: v) \Rightarrow (t <: v) \qquad \text{(Transitivity)}$$

I.e., that subtyping is a preorder.

When $t <: u$ and $u <: t$, we say the types are **equivalent**. It doesn't follow that the types are equal. In fact we will regard types to simply be abstract syntax trees, so two types are equal only when they are written out exactly the same way. But it does follow that, when t and u are equivalent, they have the same values. We will write $t \equiv u$ to mean t and u are equivalent.

2.5 Meets and joins and top

Given types t and u, an upper bound of t and u is any type v such that $t <: v$ and $u <: v$. We can form the set U of all upper bounds $\{v \mid t <: v \wedge u <: v\}$. If there is a $w \in U$ such that $w <: v$ for all v in U, then w is the least upper bound or join of x and y, it is written

$$t \sqcup u$$

Dually, we can define the greatest lower bound or meet of x and y as that member of w of $L = \{v \mid v <: t \wedge v <: t\}$ such that $v <: w$ for all v in L, if such a member exists. We will write it as $t \sqcap u$.

We have

$$\begin{aligned} t &<: t \sqcup u \\ u &<: t \sqcup u \\ t \sqcap u &<: t \\ t \sqcap u &<: u \end{aligned}$$

From soundness we can prove that

$$\begin{aligned} \mathrm{Val}(t) \cup \mathrm{Val}(u) &\subseteq \mathrm{Val}(t \sqcup u) \\ \mathrm{Val}(t \sqcap u) &\subseteq \mathrm{Val}(t) \cap \mathrm{Val}(u) \end{aligned}$$

We will want to arrange that finite joins always exist. We want joins for example to deal with if-expressions. If $e_0 : t$ and $e_1 : u$ and $e_2 : \mathsf{Bool}$ then we want

$$\mathsf{if}(e_2, \mathsf{ExpSeq}(e_0), \mathsf{ExpSeq}(e_2)) : t \sqcup u$$

Also if we catenate a list of t and a list of u, we should expect a list of $t \sqcup u$. The basic intuition for a join is that we use join types when the type checker can't figure out exactly what type an expression is, but can narrow it down to either t or u . The two possibilities do not need to be disjoint. A value could be in both t and u; then it is also in $t \sqcup u$.

It's also good to insist that finite meets exist. For example if we want to require that a parameter implement two interfaces, we can do so by requiring that it is in the meet of the two interfaces. Meet types are also useful to deal with overloaded functions.

Since the only outcomes that Bool and Number share are error and nontermination, we would expect that the meet of these two is $\bot$. In general when the meet of two types is $\bot$ we say the types are disjoint, notated $t \# u$.

We can imagine that there is a type that represents all values, i.e. that has all values in its image. Call this the "top" type, denoted by $\top$.

So far we have a picture that looks Figure 1 (leaving out a some atomic types to keep the diagram reasonable).

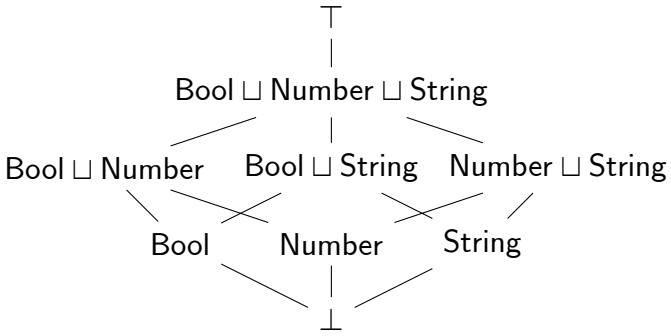

Figure 1: Subtype relations between primitive types

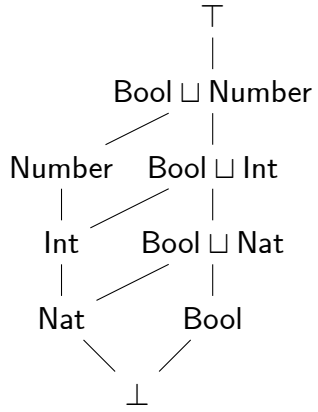

Figure 2: Adding Int and Number

2.6 Subtypes of primitive types

We introduce two new atomic types: Int and Nat with relationships Nat <: Int <: Number. Adding these to our diagram (but deleting String, to keep the diagram small) we get Figure 2.

2.7 Tuple types

To the set of values, we'll add tuples of values. If x and y are values, then so are (x, y), (y, x), (x, x, y), etc.

A tuple type is one of the following: The 0-tuple type, written as $\langle\rangle$. A 2-tuple, which is a pair of any types. written $\langle t, u\rangle$. And so on for 3, 4, and 5 tuples etc. We consider tuples of types to be types. Thus the type $\langle \mathsf{Int}, \mathsf{String}\rangle$ is a 2-tuple of types and hence a

type. There are no 1-tuple types.

The values of the type $\langle\rangle$ is the set $\{()\}$ containing only the empty tuple. The values of $\langle \mathsf{Int}, \mathsf{String}\rangle$ contains pairs of values such as (42, "hello"). But it does not contain pairs with extraordinary outcomes such as $(\blacktriangle, \blacktriangledown)$. The values of $\langle t, u\rangle$ is the set

$$\mathrm{Val}(t) \times \mathrm{Val}(u)$$

And so on for larger tuples. The reason is that, if we try to make a tuple using erroneous computation, then that is an error. Likewise if we make a tuple using a nonterminating computation we will get a nonterminating computation.

It follows that, for any t, $\mathrm{Val}(\langle t, \bot\rangle) = \emptyset = \mathrm{Val}(\bot) = \mathrm{Val}(\langle \bot, t\rangle)$. So we can expect that $\langle t, \bot\rangle \equiv \bot \equiv \langle \bot, t\rangle$.

Comparison of tuple types is item-wise, so

$$(\langle t_0, t_1\rangle <: \langle u_0, u_1\rangle) \Leftarrow (t_0 <: u_0 \wedge t_1 <: u_1)$$

We cannot make this implication an equivalence because $\langle t_0, \bot\rangle <: \langle u_0, u_1\rangle$ even if $t_0 <: u_0$ is not true.

Tuple types are used to represent types of argument lists when function calls have other than 1 argument in their argument lists.

- If a call has one argument in its argument list, that is the argument. For example a function call $\mathsf{call}(f, 12)$ is a call with one argument, which is of type Nat.
- If a call has 0 arguments in its argument list, then the argument is the unit value () and the type of the argument is $\langle\rangle$. For example a function call $\mathsf{call}(f)$ is syntactic sugar for $\mathsf{call}(f, ())$.
- And, if the call has two or more arguments in its argument list, they are tupled to make a single argument. For example, a call $\mathsf{call}\,(f, 42,$ "hello") is syntactic sugar for a call $\mathsf{call}\,(f, \mathsf{tuple}\,(42,$ "hello")).

This way all function are officially functions of one argument, but the syntax makes it easy to pack multiple arguments.

[At some point we might want to think about allowing matching of arguments to parameters by name. We will also want to deal with omitted arguments and perhaps variable length argument lists. All of these considerations I'm going to put off to later.]

Joins of a tuple type and a nontuple type produce a new type that is different from either of them. So $\langle \mathsf{Int}, \mathsf{String}\rangle \sqcup \mathsf{Int}$ is simply itself. The meet of a tuple type and a nontuple type is $\bot$.

Likewise the join of two tuple types of different lengths is a new type. E.g., $\langle \mathsf{Int}, \mathsf{String}\rangle \sqcup \langle \mathsf{Int}, \mathsf{String}, \mathsf{Nat}\rangle$ is a new type that is a supertype of each argument. And the meet of such types is bottom $\bot$. E.g.,$\langle \mathsf{Int}, \mathsf{String}\rangle \sqcap \langle \mathsf{Int}, \mathsf{String}, \mathsf{Nat}\rangle \equiv \bot$.

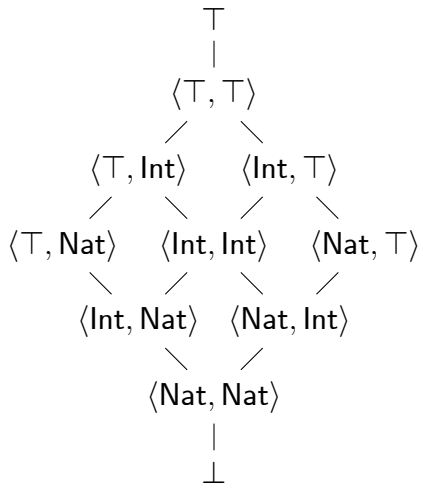

Figure 3: Tuples

When two tuple types of the same length are joined or meeted, we do the meet or join itemwise. Thus

$$\begin{aligned} \langle t_0, t_1 \rangle \sqcap \langle u_0, u_1 \rangle &\equiv \langle t_0 \sqcap u_0, t_1 \sqcap u_1 \rangle \\ \langle t_0, t_1 \rangle \sqcup \langle u_0, u_1 \rangle &\equiv \langle t_0 \sqcup u_0, t_1 \sqcup u_1 \rangle \end{aligned}$$

Figure 3 shows some tuple types and their relationships.

2.8 Function types

Suppose t is a type and u is a type. Then $t \to u$ is a function type.

For example $\mathsf{Int} \to \mathsf{String}$ represents all function values (i.e., built-in functions and closures) that, applied to a value in Int return a value in String. This includes impure functions that have side effects. There is no attempt in the type system to control side effects.

Methods in PLAAY are simply functions that are bound to objects. I won't go into exactly how methods work here. I'll just say that whether a function is a method or not is not something apparent from its type. This makes it easy to use methods as call-backs, for example.

The basic idea of subtyping for function types is that they are monotone in result and anti-monotone in the argument type. For example, suppose we have a location x of type $\mathsf{Int} \to \mathsf{Int}$. What values can we reasonably assign to x besides values that are obviously of

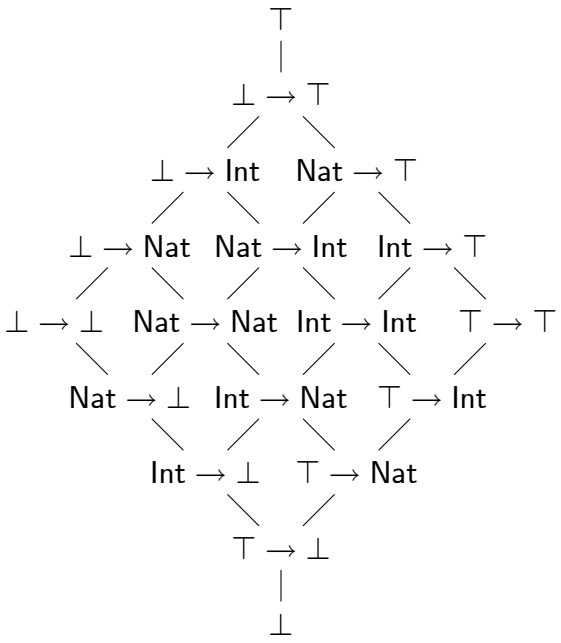

Figure 4: Function types

type $\mathsf{Int} \to \mathsf{Int}$. Well we can't assign a function of type $\mathsf{Nat} \to \mathsf{Int}$, since then a call

$$\mathsf{call}(e, \mathsf{numLit}[-1]) \qquad ,$$

where e evaluates to x, would cause a run-time error that should have been caught at design time. But it would be perfectly reasonable to assign a value of type $\mathsf{Number} \to \mathsf{Int}$. So we expect that

$$\mathsf{Number} \to \mathsf{Int} <: \mathsf{Int} \to \mathsf{Int}$$

On the other hand, it would also make sense to assign x a value of type $\mathsf{Int} \to \mathsf{Nat}$, since any result we get back from that is going to be an integer. So we expect that

$$\mathsf{Int} \to \mathsf{Nat} <: \mathsf{Int} \to \mathsf{Int}$$

In general we want

$$t_0 \to u_0 <: t_1 \to u_1, \text{ if } t_1 <: t_0 \text{ and } u_0 <: u_1 \qquad \text{(Arrow-Subtype)}$$

as illustrated in Figure 4.

2.8.1 Meets of function types

The meet of two function types represents overloaded functions.[4] For example a function in the image of

$$(\mathsf{Int} \to \mathsf{Nat}) \sqcap (\mathsf{Bool} \to \mathsf{Bool})$$

is a function that accepts an argument of either type Int or type Bool and returns a Nat, if the argument is an integer, and returns a Bool, if the argument is boolean. If the argument is both an Int and a Bool (which would mean an error or a nontermination) the answer must be in both Nat and a Bool (which would mean an error or a nontermination). We would expect this type to be a subtype of each of the following types

$$(\mathsf{Int} \to \mathsf{Nat})$$
$$(\mathsf{Bool} \to \mathsf{Bool})$$
$$(\mathsf{Int} \sqcup \mathsf{Bool} \to \mathsf{Nat} \sqcup \mathsf{Bool})$$
$$(\mathsf{Int} \sqcap \mathsf{Bool} \to \mathsf{Nat} \sqcap \mathsf{Bool})$$

Certain built-in functions are best described by meets. For example, the addition function could be described by an infinite meet.

$$(\langle\rangle \to \mathsf{Nat})$$
$$\sqcap (\mathsf{Nat} \to \mathsf{Nat}) \sqcap (\mathsf{Int} \to \mathsf{Int}) \sqcap (\mathsf{Number} \to \mathsf{Number})$$
$$\sqcap (\langle \mathsf{Nat}, \mathsf{Nat}\rangle \to \mathsf{Nat}) \sqcap (\langle \mathsf{Int}, \mathsf{Int}\rangle \to \mathsf{Int}) \sqcap (\langle \mathsf{Number}, \mathsf{Number}\rangle \to \mathsf{Number})$$
$$\sqcap (\langle \mathsf{Nat}, \mathsf{Nat}, \mathsf{Nat}\rangle \to \mathsf{Nat}) \sqcap (\langle \mathsf{Int}, \mathsf{Int}, \mathsf{Int}\rangle \to \mathsf{Int}) \sqcap (\langle \mathsf{Number}, \mathsf{Number}, \mathsf{Number}\rangle \to \mathsf{Number})$$
$$\sqcap \cdots$$

This is an example of an infinite meet that should exist. For technical reasons this paper assumes only that finite meets exist. For practical purposes, we could cut off the type of addition after some large number of arguments. There are also no doubt ways around the technical limitations in the special cases that we need.

Function meets with the same argument type If we have a function with both type $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Int} \to \mathsf{Bool}$, we can conclude that supplying the function with an Int will result in an object that is both a Int and a Bool. And since these result types are disjoint, this function call can only end in error or nontermintion. It seems reasonable to guess that

$$(\mathsf{Int} \to \mathsf{String}) \sqcap (\mathsf{Int} \to \mathsf{Bool}) \equiv (\mathsf{Int} \to \bot)$$

[4] Currently the only mechanism for overloading is the presence of default arguments. E.g., an expression

$$\lambda\mathrm{con}\ \mathrm{x} : \mathsf{Nat} := 0 \to \mathsf{Int} \cdot \ldots$$

should have both type $\mathsf{Nat} \to \mathsf{Int}$ and $\langle\rangle \to \mathsf{Int}$. But other mechanisms may be added in the future.

and more generally that

$$(t \to u_0) \sqcap (t \to u_1) \equiv (t \to u_0 \sqcap u_1)$$

But we are not forced to this conclusion. From the (arrow-subtype) rule, we can see that

$$(t \to u_0 \sqcap u_1) <: (t \to u_2) \text{ if and only if } u_0 \sqcap u_1 <: u_2$$

Taking $u_2 = u_0$ we get

$$(t \to u_0 \sqcap u_1) <: (t \to u_0)$$

and likewise

$$(t \to u_0 \sqcap u_1) <: (t \to u_1)$$

From the last two and that $a <: b$ and $a <: c$ implies $a <: b \sqcap c$ we get that

$$(t \to u_0 \sqcap u_1) <: (t \to u_0) \sqcap (t \to u_1)$$

The question then is: is it also the case that

$$(t \to u_0) \sqcap (t \to u_1) <: (t \to u_0 \sqcap u_1) \qquad ?$$

From the proposed definition of image above, it seems the right thing to do to decree that this is true.

Function meets with disjoint argument types When the argument types are disjoint as in $(\mathsf{Int} \to \mathsf{Nat}) \sqcap (\mathsf{Bool} \to \mathsf{Int})$ we can see that this should be a subtype of $(\mathsf{Int} \sqcap \mathsf{Bool} \to \mathsf{Nat} \sqcap \mathsf{Int})$ which is the same as $\bot \to \mathsf{Nat}$. If we insist that functions are monotonic in the sense that they always map error arguments to error results and nonterminating arguments to nonterminating results, we might go further and expect that $(\mathsf{Int} \to \mathsf{Nat}) \sqcap (\mathsf{Bool} \to \mathsf{Int})$ is a subtype of $\bot \to \bot$. To obtain, this we would need a rule that $t \to u <: \bot \to \bot$, for all types t and u.

Function meets with common result types Consider $(\mathsf{Bool} \to \mathsf{String}) \sqcap (\mathsf{Nat} \to \mathsf{String})$. The two argument types are disjoint. We might expect that

$$(\mathsf{Bool} \to \mathsf{String}) \sqcap (\mathsf{Nat} \to \mathsf{String}) \equiv (\mathsf{Bool} \sqcup \mathsf{Nat} \to \mathsf{String})$$

and more generally that

$$(t_0 \to u) \sqcap (t_1 \to u) \equiv (t_0 \sqcup t_1 \to u)$$

From (arrow-subtype) we get $(t_0 \to u) <: (t_0 \sqcup t_1 \to u)$ and we also have $(t_1 \to u) <: (t_0 \sqcup t_1 \to u)$. From this we get

$$(t_0 \to u) \sqcap (t_1 \to u) <: (t_0 \sqcup t_1 \to u)$$

However to get the other half of the equivalence, we must decree that

$$(t_0 \sqcup t_1 \to u) <: (t_0 \to u) \sqcap (t_1 \to u)$$

2.8.2 Joins of function types

[to be done]

2.9 Field types

We now consider objects. An object is a collection of fields. Each field has a name (unique within the object), a type, and a value.

If i is an identifier and t is a type, then $i\colon t$ is a **field type**. The image of this type is all objects that have a field named i of type t. These objects can have other fields too.

Consider $\mathsf{a}\colon \mathsf{Bool}$ this type's image includes the objects

$$\{\mathsf{a}\colon \mathsf{Bool} \mapsto \mathsf{true}\}$$

and

$$\{\mathsf{a}\colon \mathsf{Bool} \mapsto \mathsf{true}, \mathsf{b}\colon \mathsf{Nat} \mapsto 12\}$$

The latter of these is also in the images of $\mathsf{b}\colon \mathsf{Nat}$, $\mathsf{b}\colon \mathsf{Int}$, and $\mathsf{a}\colon \mathsf{Bool} \sqcap \mathsf{b}\colon \mathsf{Int}$.

We can see then that the following laws seem reasonable

$$i\colon t <\colon i\colon u, \text{ if } t <\colon u$$

and

$$i\colon t \sqcap u <\colon i\colon t$$

for any u.

A field type or a meet of field types is called an **interface type**. These correspond roughly to interfaces in languages like Java. The type $\mathsf{a}\colon \mathsf{Bool} \sqcap \mathsf{b}\colon \mathsf{Int}$ represents all objects that have both a field named "a" that contains only values in the image of type Bool and a field "b" that contains only values in the image of type Int.

[Suppose, for a moment, we consider that for each individual string there is a corresponding type, so "a" $<\colon$ String and $\mathrm{Val}(\text{"a"}) = \{\text{"a"}\}$. Then $\mathsf{a}\colon \mathsf{Bool}$ can be considered to be equivalent to "a" $\to$ Bool. I'm not yet willing to take this step.]

Sequences can be thought of as a combination of a function and a length field. Thus (Following Reynolds [[1988]]) $\mathsf{Seq}[t]$ is a synonym for $(\mathsf{Nat} \to t) \sqcap (\mathsf{length} : \mathsf{Nat})$. We have $\mathsf{Seq}[t] <\colon \mathsf{Seq}[u]$ whenever $t <\colon u$.

2.10 Locations

A short comment on wording: In PLAAY —or at least in this document— we use the word **variable** in the mathematical sense of a name for a value. Variables may be parameters, local variables, or fields of objects. (In fact, since activation frames are simply

objects, all three kinds of variable are fields.) Anyway for the lifetime of a variable its "value" does not change, except to change from uninitialized to initialized. In order to accommodate imperative programming, we need assignment. In PLAAY, the thing we assign to is a **location**. Locations can be thought of as addresses that index a store. Assignment alters the store. Locations are themselves considered values and so we can use variables to name locations. Earlier documentation may use these words differently. **end comment**

If we have an assignment command $a := e$, where a is a Number location, e could be of type Number or Int or Nat. If we think of $a :=$ as a function f so that $a := e$ can be written as $f(e)$, we find that f is a function of type $\mathsf{Number} \to \langle\rangle$.[5]

At first I considered three approaches to locations:

- Following Reynolds' [[1988]] design for Forsythe, we can think of a location of type t as being an object of type $t \sqcap (t \to \langle\rangle)$. Then $a := e$ is literally an abbreviation for $a(e)$. Under this approach we define $\mathsf{Loc}[t] \equiv t \sqcap (t \to \langle\rangle)$. Thus $\mathsf{Loc}[t] <: t$, for all t.

- A second approach is to think of a location of type t as being an object of type $t \sqcap \mathsf{set}\colon\ (t \to \langle\rangle)$ with $a := e$ being an abbreviation for $a.\mathsf{set}\,(e)$. Under this approach we define $\mathsf{Loc}[t] \equiv t \sqcap \mathsf{set}\colon\ (t \to \langle\rangle)$ (or at least $\mathsf{Loc}[t] <: t$) $\mathsf{Loc}[t] <: t \sqcap \mathsf{set}\colon\ (t \to \langle\rangle)$).

- A third approach is to think of a location of type t as being an object of type $\mathsf{get}\colon\ (\langle\rangle \to t) \sqcap \mathsf{set}\colon\ (t \to \langle\rangle)$ with the application of .get being implicit. In this approach the assignment $a := e$ is an abbreviation for $a.\mathsf{set}\,(e)$. With this approach we define $\mathsf{Loc}[t] \equiv \mathsf{get}\colon\ (\langle\rangle \to t) \sqcap \mathsf{set}\colon\ (t \to \langle\rangle)$. In this case it is not true that $\mathsf{Loc}[t] <: t$. This means that at certain places in the code (called R contexts), locations will need to be dereferenced.

We take an approach similar to the third. $\mathsf{Loc}[t]$ is a class that implements $\mathsf{get}\colon\ (\langle\rangle \to t) \sqcap \mathsf{set}\colon\ (t \to \langle\rangle)$.

We can abbreviate $\mathsf{get}\colon\ (\langle\rangle \to t)$ as $\mathsf{Source}[t]$ and $\mathsf{set}\colon\ (t \to \langle\rangle)$ as $\mathsf{Acceptor}[t]$. So

$$\mathsf{Loc}[t] <: \mathsf{Source}[t] \sqcap \mathsf{Acceptor}[t]$$

The rationale for rejecting the first two approaches is this. Both these approaches have the seeming advantage that $\mathsf{Loc}[t] <: t$. This seems to simplify, for example, the type checking of assignments. For example if x and y are both expressions of type $\mathsf{Loc}[\mathsf{Int}]$ then the assignment $x := y$ will type check for the same reason that the assignment $x := 12$ type checks, i.e., that the type of the right-side is a subtype of Nat. This can be summed up in the type checking rule

$$\frac{x : v \qquad y : t \qquad v <: \mathsf{Acceptor}[u] \qquad t <: u}{x := y : \langle\rangle}$$

[5] As mentioned above, functions can have side effects.

(Where $\mathsf{Acceptor}[u]$ is defined as $u \to \langle\rangle$ for the first choice and as $\mathsf{set}\colon (t \to \langle\rangle)$ for the second and third.) So up to now everything looks good. But consider an expression z of type $\mathsf{Loc}[\mathsf{Loc}[\mathsf{Int}]]$. This is a subtype of $\mathsf{Loc}[\mathsf{Int}]$ which is a subtype of Int and so, by transitivity $\mathsf{Loc}[\mathsf{Loc}[\mathsf{Int}]]$ is a subtype of Int and we are forced to conclude that $x := z$ should also type check. This would be OK if the PLAAY followed the lead of Gedanken [[Reynolds, 1970]] and made it part of the semantics of assignments that the right-hand side is repeatedly fetched from until a nonlocation is found. However that would make assignment to variables of type $\mathsf{Loc}[\mathsf{Loc}[t]]$ impossible (Gedanken has a second assignment operator for the purpose of allowing assignments of locations. Forsythe does not have locations that can hold locations.) The intention in PLAAY is that the right hand side of an assignment will be fetched from only once if possible and otherwise not at all. In PLAAY, our example $x := z$ does not type check since there is no u such that $\mathsf{Loc}[\mathsf{Int}] <: \mathsf{Acceptor}[u]$ and either $\mathsf{Loc}[\mathsf{Loc}[\mathsf{Int}]] <: u$ or $\mathsf{Loc}[\mathsf{Loc}[\mathsf{Int}]] <: \mathsf{Source}[u]$; this is what we want. The price is that the rules for type checking assignments are a bit more complex.

In the previous paragraph, I've used assignment as an example. Similar remarks apply to operands of other operators, such as the call operator. In assignment, the left operand is said to be in an L context. The right operand is said to be in an R context. Expressions in an R context may be evaluated slightly differently from expressions in an L context; var, dot and call expressions in an R context are evaluated in two phases; in the first, the expression is evaluated as if it were in an L context. In the second, if the result of the first phase is a source, the source is fetched from, otherwise the result is just the result of the first phase. Every expression in a program can be considered to be in an R context or an L context. Arguments to calls are in an R context, as are the first operands of 'if' and 'while' expressions and the bodies of lambda expressions. The operands of the tupling operation inherit their context from the tuple operation itself. For example in the tree

$$\mathsf{assign}(\mathsf{tuple}(\mathsf{var}[\mathsf{a}], \mathsf{var}[\mathsf{b}]), \mathsf{tuple}(\mathsf{var}[\mathsf{c}], \mathsf{numberLiteral}[12]))$$

the first tuple is in an L context and so its children are also; the second tuple is in an R context and so its children are also.

2.10.1 Some consequences

We have that $\mathsf{Loc}[t] \equiv \mathsf{Loc}[u]$ is true when $t \equiv u$.

When $t \not\equiv u$ and neither is equivalent to $\bot$, we should expect that on semantic grounds $\mathsf{Loc}[t]$ and $\mathsf{Loc}[u]$ are disjoint, i.e. that the intersection of their value is empty. This can be justified on semantic grounds as follows. Suppose $t \not\equiv u$. Then there may be at least one value in the image of t that is not in the image of u or the other way around. Without loss of generality, suppose x is in the image t and not in the image of u. Of course x is a value. Consider an ordinary member y that is in the image of $\mathsf{Loc}\,[u]$ which has a set field which gives an error when applied to x and, more importantly, does not update the contents of

the location. y should not be in the image of $\mathsf{Loc}[t]$. Now consider any ordinary member z of the image of $\mathsf{Loc}[t]$. After we set z to x it has a value that is not in u, a location that might hold a value not in u should not be in the image of $\mathsf{Loc}\,[u]$.

What about the case that $t \not\equiv u$, but one of t and u is equivalent to $\bot$. Suppose, without loss of generality that $t \equiv \bot$ and $u \not\equiv \bot$. An value of type $\mathsf{Loc}[t]$ would be an address of a part of the store that can not hold any values. Do such values exist? We don't have to decide yet.

- Suppose we decide that $\mathsf{Loc}[t]$ has no values. Then we have that $\mathrm{Val}(\mathsf{Loc}[t]) = \emptyset$.So $\mathrm{Val}(\mathsf{Loc}[t]) \cap \mathrm{Val}(\mathsf{Loc}\,[u]) = \emptyset$.
- On the other hand, if we decide that there are values in $\mathrm{Val}(\mathsf{Loc}[t])$, they can not accept values of $\mathrm{Val}(u)$ and so are not in $\mathsf{Loc}\,[u]$. Any ordinary values of $\mathsf{Loc}\,[u]$ would be able to store and later produce ordinary values of $\mathrm{Val}(u)$, which an ordinary value in $\mathrm{Val}(\mathsf{Loc}[t])$ would not be able to do. And so, again, we conclude that $\mathrm{Val}(\mathsf{Loc}[t]) \cap \mathrm{Val}(\mathsf{Loc}\,[u]) = \emptyset$.

Either way, if $t \not\equiv u$ then the intersection of their values should be empty.

2.10.2 Aliasing

By admitting variables as values, we open a can of worms, which is aliasing. However this can was, in fact, already open in the Winter 2018 implementation of PLAAY. (In PLAAY an object can end up as part of the stack chain and so an assignment to a location a might also change a field $p.a$ in an object accessed through a pointer. Likewise an assignment to $p.a$ might change location a.) As this genie is already partway out of the bottle the choice is whether to find a way to stuff it back in, or to let it out all the way. Java stuffs the genie back in the bottle by not allowing any aliasing between stack variables and stack variables nor between stack variables and heap variables. C++ allows unfettered aliasing. Allowing this aliasing may make automated verification more tricky. On the other hand Occam's razor seems to lead us to location types.

On the plus side, pass-by-reference parameters (reference parameters for C++ers, var parameters for Pascal fans) are now perfectly natural; for example, we have a type $\mathsf{Loc}[t] \to u$.[6] Mutable fields in interfaces also make sense for example, $\mathsf{f}\colon \mathsf{Loc}[t]$.

[6]There is a catch here because at the call site side we typically need to suppress the normal value conversion. E.g., if x is a $\mathsf{Loc}(\mathsf{Int})$ and f is a $\mathsf{Loc}[\mathsf{Int}] \to \mathsf{Int}$ then $\mathsf{call}(f, \mathsf{var}[x])$ will normally pass an Int,leading to a type error. So either we need to do something fancy on semantics of the call, or we need for the caller to indicate which parameters are passed by reference. I don't want to do anything in the semantics that causes types to changes the meaning of a legitimate program. (Technically we want an erasure property in which the types of a correct program can be erased without changing its meaning.) That leaves it to the programmer to mark reference parameters at the call site. For this we have the

It is tempting to think that location declarations are just abbreviations for constant declarations. For example we might expect that

$$\textbf{loc}\ \mathsf{x} : \mathsf{Int} := 12$$

is just an abbreviation for

$$\mathsf{x} : \mathsf{Loc}[\mathsf{Int}] := 12$$

but this is not right, since the type of 12 is Nat and Nat is not a subtype of Loc[Int],so the latter has a type error, while th former should not. A location declaration creates (at runtime) a new location while a constant declaration simply gives a name to a value. We can think of

$$\textbf{loc}\ \mathsf{x} : \mathsf{Int} := 12$$

as an abbreviation for

$$\mathsf{x} : \mathsf{Loc}[\mathsf{Int}] := \mathsf{new}\ \mathsf{Loc}(12)$$

where Loc is a constructor of locations.[7]

For similar reasons, the pass-by-reference parameters of a function are usually written as constant declarations. For example this lambda expression

$$\lambda\ \mathsf{y} : \mathsf{Loc}[\mathsf{Int}] \cdot \{\ \mathsf{y} := 12\ \} \tag{1}$$

has type $\mathsf{Loc}[\mathsf{Int}] \to \langle\rangle$. The lambda expression

$$\lambda\ \textbf{loc}\ \mathsf{y} : \mathsf{Int} \cdot \{\ \mathsf{y} := \mathsf{y} + 1\ ;\ y\ \} \qquad , \tag{2}$$

on the other hand, has type $\mathsf{Int} \to \mathsf{Int}$. It is a pure function with no side effects. The fact that the parameter is declared as a location is purely a matter that is internal to the function's implementation. The function (2) has the same type and does the same thing (although in a slightly different way) as the function

$$\lambda\ \mathsf{y} : \mathsf{Int} \cdot \{\ \mathsf{y} + 1\ \}$$

The lambda expression

$$\lambda\ \textbf{loc}\ \mathsf{y} : \mathsf{Loc}[\mathsf{Int}] \cdot \{\ \mathsf{y} := 12\ \}$$

operator loc whose operand is an L-context. Then the call would look like $\mathsf{call}(f, \mathsf{loc}\,(\mathsf{var}[x])\,)$. I don't mind this, as it makes the code more readable and the semantics simple.

[7] The construction new Loc(−) looks a bit like the construction loc (−) proposed in an earlier foot note but they are different new Loc(−) evaluates its operand for value (i.e., its operand is an R-context) and wraps a location around the value. The construction loc (−) evaluates its operand for location. For example if x is a Loc(Int), then new Loc(x) is a new Loc(Int) that initially shares the same value as x, but loc (x) simply evaluates to original location named x itself.

would have the same type as (1), but it has a type error in the assignment: the type of y is Loc[Loc[Int]] and the type of 12 is Nat and Nat is not a subtype of Loc[Int]. A correct, but gratuitously complex, equivalent to (1) is

$$\lambda \textbf{ loc } \mathsf{y} : \mathsf{Loc}[\mathsf{Int}] \cdot \{\ \mathsf{id}(\mathsf{y}) := 12\ \}$$

where id is the identity function.

Arrays can be thought of as sequences of locations. A declaration

$$a := \textbf{ new } \mathsf{Array}[\mathsf{Int}](10)$$

declares a new variable of type Seq[Loc[Int]]. Consider the assignment $a(i) := b(i)$ where a and b are both arrays. The call on the left side returns a location. The call on the right side also returns a location, but this location is implicitly fetched.

2.11 Classes

Classes are blueprints for objects. A class is a type, but it is more than a type in the sense that there are things we can do with classes that we can not do with other types. For example we can use a class to create new objects.

I'm going to ignore classes for now and come back to them later. When I come back to them we will find that a class k that defines a field i of type t will be such that $i\colon t <\colon k$.

2.12 Named types and recursion

As a convenience to the programmer, we should allow complex types to be abbreviated with a name and that name to be used in place of the type. However we will not allow recursion. (At least not yet.) For example a declaration like this

$$\textbf{typename } \mathsf{fred} = (\mathsf{head} : \mathsf{int} \sqcap \mathsf{tail} : \mathsf{fred})$$

which introduces a new name fred and then uses that name in its definition, would not be allowed.

3 Formalizing the type system

3.1 Abstract syntax for types

[To be done]

3.2 Types

We will define the types of PLAAY to be all trees finitely generated by the following abstract grammar which defines types

$$\begin{array}{rcl} t, u, v ::= & & r \\ & | & t \sqcup u \\ & | & \bot \end{array}$$

type terms

$$\begin{array}{rcl} r, s ::= & & q \\ & | & r \sqcap s \\ & | & \top \end{array}$$

type factors

$$\begin{array}{rcl} q ::= & & p \\ & | & \langle\rangle \mid \langle t_0, t_1\rangle \mid \langle t_0, t_1, t_3\rangle \mid \cdots \\ & | & t \to u \\ & | & i\colon t \\ & | & \mathsf{Loc}[t] \end{array}$$

and primitive types

$$p ::= \mathsf{Bool} \mid \mathsf{String} \mid \mathsf{Number} \mid \mathsf{Int} \mid \mathsf{Nat} \mid \mathsf{Null}$$

where i is any identifier. This formalization ignores the use of identifiers as types, since that is easily dealt with in other ways. Note that we assume that types are always in join normal form; that is a type always has its joins outside of meets; $(\mathsf{Bool} \sqcup \mathsf{Int}) \sqcap \mathsf{Null}$ is not a type according to the above grammar, but we can rewrite it as a type $(\mathsf{Bool} \sqcap \mathsf{Null}) \sqcup (\mathsf{Int} \sqcap \mathsf{Null})$.

3.3 The length of types

For types factors, we can define a length of the type factor

$$\begin{array}{rcl} \#\langle\rangle & = & 0 \\ \#p & = & \#(t \to u) = \#(i\colon t) = \#\mathsf{Loc}[t] = 1 \\ \#\langle t_0, t_1\rangle & = & 2 \\ \#\langle t_0, t_1, t_3\rangle & = & 3 \\ & & \cdots \end{array}$$

3.4 Subtypes

We wish to formally define the subtype relation $t <: u$. We do that by first defining a relationship between finite sets of types. We'll define

$$\Theta <: \Delta$$

where Θ and Δ are finite sets of types and then define $t <: u$ as $\{t\} <: \{u\}$. The intended interpretation is that $\Theta <: \Delta$ should hold only if the intersection[8] of the values of types in Θ is contained in the union of values in types in Δ:

$$\Theta <: \Delta \qquad \Rightarrow \qquad \bigcap_{t \in \Theta} \mathrm{Val}(t) \subseteq \bigcup_{u \in \Delta} \mathrm{Val}(u)$$

This is soundness. Completeness of the subtype relation is the converse implication. Fans of logic may recognize that $\Theta <: \Delta$ is analogous to a sequent in the sequent calculus.

We can define the relation $\Theta <: \Delta$ as the least relation that obeys the following rules. In these rules and below, we abbreviate a singleton set $\{t\}$ by its sole occupant t and $\Theta \cup \{t\}$ with Θ, t.

- Reflexive rule

$$\Theta <: \Theta$$

- Subset rules

$$\frac{\Theta <: \Delta}{\Theta <: \Delta \cup \Pi}$$

and

$$\frac{\Theta <: \Delta}{\Theta \cup \Pi <: \Delta}$$

- Cut Rule

$$\frac{\Theta <: \Delta \cup \Pi \qquad \Pi \cup \Theta <: \Delta}{\Theta <: \Delta}$$

- Top and bottom rules:

$$\bot <: \Delta \qquad \frac{\Theta <: \emptyset}{\Theta <: \bot}$$

and

$$\Theta <: \top \qquad \frac{\emptyset <: \top}{\top <: \Delta}$$

[8] The interssection of 0 sets here is considered to be the universe of all values. This universe V and will be defined in a later section along with the Val function.

- Primitive rules

$$\mathsf{Int} <: \mathsf{Number} \qquad \mathsf{Nat} <: \mathsf{Number} \qquad \mathsf{Nat} <: \mathsf{Int}$$

- Tuple rule. For tuples of equal length

$$\frac{t_0 <: u_0 \qquad t_1 <: u_1 \qquad \cdots}{\langle t_0, t_1, ... \rangle <: \langle u_0, u_1, ... \rangle}$$

- Function rule

$$\frac{u_0 <: t_0 \qquad t_1 <: u_1}{(t_0 \to t_1) <: (u_0 \to u_1)}$$

- Field rules

$$\frac{t <: u}{i : t <: i : u}$$

- Location rules

$$\frac{t <: u \qquad u <: t}{\mathsf{Loc}[t] <: \mathsf{Loc}[u]}$$

  and

$$\frac{\mathsf{get} : (\langle\rangle \to t), \mathsf{set} : (t \to \langle\rangle) <: \Delta}{\mathsf{Loc}[t] <: \Delta}$$

- Meet rules

$$\frac{r_0, r_1 <: \Delta}{r_0 \sqcap r_1 <: \Delta}$$

  and

$$\frac{\Theta <: u_0 \qquad \Theta <: u_1}{\Theta <: u_0 \sqcap u_1}$$

- Join rules

$$\frac{\Theta <: u_0, u_1}{\Theta <: u_0 \sqcup u_1}$$

  and

$$\frac{t_0 <: \Delta \qquad t_1 <: \Delta}{t_0 \sqcup t_1 <: \Delta}$$

The following additional rules allow us to prove disjointness. A set of types Θ is **disjoint** if $\Theta <: \emptyset$. As set of types Θ is **pair-wise disjoint** if for every two different types $t, u \in \Theta$, $\{t, u\} <: \emptyset$.

- Length disjointness rule. This rule relies on the definition of length given above.

$$\frac{\#q_0 \neq \#q_1}{q_0, q_1 <: \emptyset}$$

- Primitive disjointness rules.

$$\begin{aligned} \mathsf{Bool}, \mathsf{Number} &< : \emptyset \\ \mathsf{Bool}, \mathsf{String} &< : \emptyset \\ \mathsf{Bool}, \mathsf{Null} &< : \emptyset \\ \mathsf{Number}, \mathsf{String} &< : \emptyset \\ \mathsf{Number}, \mathsf{Null} &< : \emptyset \\ \mathsf{String}, \mathsf{Null} &< : \emptyset \end{aligned}$$

- Tuple disjointness rules

$$\frac{t_0, u_0 <: \emptyset}{\langle t_0, t_1 \rangle, \langle u_0, u_1 \rangle <: \emptyset} \qquad \frac{t_1, u_1 <: \emptyset}{\langle t_0, t_1 \rangle, \langle u_0, u_1 \rangle <: \emptyset}$$

and

$$\frac{t_0, u_0 <: \emptyset}{\langle t_0, t_1, t_2 \rangle, \langle u_0, u_1, u_2 \rangle <: \emptyset} \qquad \frac{t_1, u_1 <: \emptyset}{\langle t_0, t_1, t_2 \rangle, \langle u_0, u_1, u_2 \rangle <: \emptyset} \qquad \frac{t_2, u_2 <: \emptyset}{\langle t_0, t_1, t_2 \rangle, \langle u_0, u_1, u_2 \rangle <: \emptyset}$$

and so on.

- Other disjointness rules. Recall that p ranges over primitive types.

$$\begin{aligned} &p, (t \to u) <: \emptyset \\ &p, (i : t) <: \emptyset \\ &p, \mathsf{Loc}[t] <: \emptyset \\ &(t \to u), \mathsf{Loc}[t] <: \emptyset \end{aligned}$$

- [[Perhaps we need a rule that location types are disjoint from objects that don't implement the location type's interface.]]

3.5 Values and semantics of types

In this section we give a semantics for types. It uses sets rather than domains. I think this is adequate for the purpose of showing the subtype relation sound and complete and providing a basis for dynamic and static type checking. The use of sets for semantics is common in logic and I think it works here for the same reasons it works there.[9]

[9] The semantics given here might not be adequate as a basis for providing a semantics to the expressions of the language. But that is not my current purpose. That is a bridge to be crossed later. The semantics of expressions will use either an operational approach or the predicative programming approach of Hehner and Kassios; perhaps both. An operational semantics is obviously useful for guiding (or confirming) the implementation of language. An predicative semantics would provide the basis for contracts and (automated) proofs of programs. Having both would provide a chain from design-by-contract to the implementation.

3.5.1 Values

Values are trees following this abstract grammar

$$
\begin{array}{rcl}
x, y, z \in \mathcal{V} & : & := \mathsf{boolv}\,(b) \\
& | & \mathsf{stringv}\,(str) \\
& | & \mathsf{numberv}\,(n) \\
& | & \mathsf{tuplev}\,(x_0, x_1, \ldots x_k) \\
& | & \mathsf{nullv} \\
& | & \mathsf{obv}\,(f_0, f_1, \ldots f_k) \\
& | & \mathsf{locv}\,(t, a) \\
& | & \mathsf{closurev}\,(\lambda, \eta, t \rightarrow u, a) \\
& | & \mathsf{builtinv}(t \rightarrow u, a) \\
f & : & := \mathsf{field}(i, t, a) \\
\lambda & : & := \mathsf{lambda}(params, optType, seq)
\end{array}
$$

where: b is a boolean in the set $\{\mathsf{true}, \mathsf{false}\}$; str is a string from a set of strings $\mathcal{S}$; n is a rational number from a set of rational numbers $\mathcal{Q}$[10]; t and u are types; a is an address from a countable and infinite set of addresses $\mathcal{A}$; η is an environment, which is a finite sequence of values each of which is an obv; and i is an identifier from a countable, infinite, totally ordered set of identifiers $\mathcal{I}$. In any obv, no two fields may have the same identifier and we restrict the fields of an obv to being in order by identifier. In tuplev, the number of values must not be 1.

Values are trees in the sense that there are no loops. Equality of values is completely straight forward. Two values are equal if they are the same tree.[11]

We'll also assume that each address is associated with exactly one type. For example

$$x = \mathsf{locv}\,(t_0, a) \wedge y = \mathsf{locv}\,(t_1, a) \Rightarrow t_0 = t_1 \wedge \mathsf{locv}\,(t_0, a) = \mathsf{locv}\,(t_1, a)$$

[10] $\mathcal{Q}$ could be the set of all rational numbers or some subset that is conveniently represented on the computer; in the current implementation it is the set of numbers that javascript can represent and so includes positive and negative infinities and various NaN values in addition to a subset of the rationals.

[11] In contrast to most OOPLs there is (technically) no object identity. However fields do have identity, since they have addresses. Thus if we evaluate the expression

```
objectLiteral( vardecl[con](  var[a],
                              intType,
                              numberLiteral[123] ) )
```

twice we will get unequal objects since we will get objects with unequal fields, since the semantics of field creation demands that a new address is used for each field creation. This is an artifact of the evaluation semantics and not baked in to the structure of values.

We can imagine that each element of $\mathcal{A}$ is associated with just one type and that each time we create a location, we are sure to pick an address associated with the appropriate type. Similarly for fields, closures, and built-ins.

Note that location values exist even if the type is equivalent to $\bot$. In practice such values are not useful and we can make it a dynamic or static error to create one. But in terms of defining our set of values we will say that they exist. A similar comment applies to fields.

3.5.2 Semantics of types

For each type t, we'll define a set of values $\mathrm{Val}(t)$. We define

$$\mathrm{Im}(t) = \mathrm{Val}(t) \cup \{\blacktriangle, \blacktriangledown\}$$

Next we define the Val function.

$$\begin{aligned}
\mathrm{Val}\,(\top) &= \mathcal{V} \\
\mathrm{Val}\,(\bot) &= \emptyset \\
\mathrm{Val}\,(r \sqcap s) &= \mathrm{Val}\,(r) \cap \mathrm{Val}\,(s) \\
\mathrm{Val}\,(t \sqcup u) &= \mathrm{Val}\,(t) \cup \mathrm{Val}\,(u)
\end{aligned}$$

For tuple types we have

$$\begin{aligned}
\mathrm{Val}(\langle\rangle) &= \{\mathsf{tuplev}()\} \\
\mathrm{Val}(\langle t_0, t_1\rangle) &= \{\mathsf{tuplev}(x_0, x_1) \in \mathcal{V} \mid x_0 \in \mathrm{Val}(t_0) \wedge x_1 \in \mathrm{Val}(t_1)\} \\
&\ldots
\end{aligned}$$

Since the members of $\mathcal{V}$ are trees, we can not have tuple values that contain themselves.

For function types we have

$$\begin{aligned}
\mathrm{Val}\,(t_0 \to u_0) &= \{\mathsf{closurev}\,(t_1 \to u_1, \lambda, \eta, a) \in \mathcal{V} \mid \mathrm{Val}(t_0) \subseteq \mathrm{Val}(t_1) \wedge \mathrm{Val}(u_1) \subseteq \mathrm{Val}(u_0)\} \\
&\quad \cup \{\mathsf{builtinv}\,(t_1 \to u_1, a) \in \mathcal{V} \mid \mathrm{Val}(t_0) \subseteq \mathrm{Val}(t_1) \wedge \mathrm{Val}(u_1) \subseteq \mathrm{Val}(u_0)\}
\end{aligned}$$

For fields

$$\begin{aligned}
\mathrm{ObVal}\,(i : t) &= \{\mathsf{obv}\,(\ldots, \mathsf{field}(j, u, a), \ldots) \in \mathcal{V} \mid i = j \wedge \mathrm{Val}(u) \subseteq \mathrm{Val}(t)\} \\
\mathrm{Val}\,(i : t) &= \begin{cases}
\mathrm{ObVal}\,(i : t) \cup \{\mathsf{locv}(u, a) \mid \mathrm{Val}(u \to \langle\rangle) \subseteq \mathrm{Val}(t)\} & \text{if } i = \mathsf{set} \\
\mathrm{ObVal}\,(i : t) \cup \{\mathsf{locv}(u, a) \mid \mathrm{Val}(u) \subseteq \mathrm{Val}(t)\} & \text{if } i = \mathsf{get} \\
\mathrm{ObVal}\,(i : t) & \text{otherwise}
\end{cases}
\end{aligned}$$

For location types

$$\mathrm{Val}\left(\mathsf{Loc}[t]\right) = \{\mathsf{locv}(u, a) \mid \mathrm{Val}(t) = \mathrm{Val}(u)\}$$

For primitive types we have

$$\begin{aligned}
\mathrm{Val}(\mathsf{Bool}) &= \{\mathsf{boolv}\left(\mathsf{true}\right), \mathsf{boolv}\left(\mathsf{false}\right)\} \\
\mathrm{Val}(\mathsf{String}) &= \{\mathsf{stringv}\left(str\right) \mid str \in \mathcal{S}\} \\
\mathrm{Val}(\mathsf{Number}) &= \{\mathsf{numberv}\left(n\right) \mid n \in \mathcal{Q}\} \\
\mathrm{Val}(\mathsf{Int}) &= \{\mathsf{numberv}\left(n\right) \mid n \in \mathcal{Q} \wedge n \text{ is an integer}\} \\
\mathrm{Val}(\mathsf{Nat}) &= \{\mathsf{numberv}\left(n\right) \mid n \in \mathcal{Q} \wedge n \text{ is a nonnegative integer}\} \\
\mathrm{Val}(\mathsf{Null}) &= \{\mathsf{nullv}\}
\end{aligned}$$

3.5.3 Soundness

At this point we have formally defined types, subtyping, and images; so we can ask the questions of whether the subtyping relation is sound

$$\forall t, u \cdot (t <: u) \Rightarrow (\mathrm{Val}(t) \subseteq \mathrm{Val}(u)) \qquad \text{(Soundness)}$$

and complete

$$\forall t, u \cdot (\mathrm{Val}(t) \subseteq \mathrm{Val}(u)) \Rightarrow (t <: u) \qquad \text{(Completeness)}$$

3.6 Dynamic Type Checking

The most important operation for dynamic type checking is determining whether a given value x is a value of a given type t. This is used for example to check the initialization of variables, initialization of locations, assignment to locations, and results from functions. We have the following algorithm.

$$\begin{aligned}
\mathrm{In}(x, \bot) &= \mathsf{false} \\
\mathrm{In}(x, t \sqcup u) &= \mathrm{In}(x, t) \vee \mathrm{In}(x, u) \\
\mathrm{In}(x, \top) &= \mathsf{true} \\
\mathrm{In}(x, r \sqcap s) &= \mathrm{In}(x, r) \wedge \mathrm{In}(x, s)
\end{aligned}$$

$$\begin{aligned}
\mathrm{In}(x, \langle\rangle) &= (x = \mathsf{tuplev}()) \\
\mathrm{In}(x, \langle t_0, t_1 \rangle) &= (\exists x, y \cdot x = \mathsf{tuplev}(x, y) \wedge \mathrm{In}(y, t_0) \wedge \mathrm{In}(z, t_1)) \\
&\ldots
\end{aligned}$$

$$\begin{aligned}\mathrm{In}\,(x, t_0 \to u_0) \;=\; & \left(\exists t_1, u_1, \lambda, \nu, a \cdot \begin{array}{ll} & x = \mathsf{closurev}\,(t_1 \to u_1, \lambda, \eta, a) \\ \wedge & t_0 <: t_1 \wedge u_1 <: u_0 \end{array}\right) \\ & \vee \left(\exists t_1, u_1, a \cdot \begin{array}{ll} & x = \mathsf{builtinv}\,(t_1 \to u_1, a) \\ \wedge & t_0 <: t_1 \wedge u_1 <: u_0 \end{array}\right)\end{aligned}$$

$$\begin{aligned}\mathrm{In}\,(x, i : t) \;=\; & \left(\exists u, a \cdot \begin{array}{ll} & x = \mathsf{obv}\,(\ldots, \mathsf{field}(i, u, a), \ldots) \\ \wedge & u <: t \end{array}\right) \\ & \vee \left(i = \mathsf{set} \wedge \exists u, a \cdot \begin{array}{ll} & x = \mathsf{locv}\,(u, a) \\ \wedge & u \to \langle\rangle <: t \end{array}\right) \\ & \vee \left(i = \mathsf{get} \wedge \exists u, a \cdot \begin{array}{ll} & x = \mathsf{locv}\,(u, a) \\ \wedge & u <: t \end{array}\right)\end{aligned}$$

$$\mathrm{In}\,(x, \mathsf{Loc}[t]) = \exists u, a \cdot \begin{array}{ll} & x = \mathsf{locv}\,(u, a) \\ \wedge & t \equiv u \end{array}$$

$$\begin{aligned}\mathrm{In}(x, \mathsf{Bool}) &= (x = \mathsf{boolv}\,(\mathsf{true}) \vee x = \mathsf{boolv}\,(\mathsf{false})) \\ \mathrm{In}(x, \mathsf{String}) &= \exists str \in \mathcal{S} \cdot x = \mathsf{stringv}\,(str) \\ \mathrm{In}(x, \mathsf{Number}) &= \exists n \in \mathcal{Q} \cdot x = \mathsf{numberv}\,(n) \\ \mathrm{In}(x, \mathsf{Int}) &= \exists n \in \mathcal{Q} \cdot x = \mathsf{numberv}\,(n) \wedge n \text{ is an integer} \\ \mathrm{In}(x, \mathsf{Nat}) &= \exists n \in \mathcal{Q} \cdot x = \mathsf{numberv}\,(n) \wedge n \text{ is a nonnegative integer} \\ \mathrm{In}(x, \mathsf{Null}) &= (x = \mathsf{nullv})\end{aligned}$$

3.7 Static Typing of expressions

[[This is still work in progress. But it is getting better.]]

Judgements. Recall that expressions are evaluated in either an L context or an R context. For example the first operand of an assignment is evaluated in an L context and the second in an R context. When type checking we need to be sensitive to wether an expression is in an L or R context. Let Γ be a partial function from identifiers to types and h be either L or R. For an expression or declaration m and type t, we write $\Gamma \vdash_h m : t$ to mean that m has type t if evaluated in an h context with an environment described by Γ. The rules are intended to convey a partial function in that, for a given Γ, h, and e, there should be at most one t so that $\Gamma \vdash_h e : t$. E.g. we'll have $\Gamma \vdash \mathsf{numberLiteral}[12] : \mathsf{Nat}$, but not $\Gamma \vdash \mathsf{numberLiteral}[12] : \mathsf{Int}$. We aim to get the smallest type that is consistent with the meaning of the code.

3.7.1 Typing rules

Here are some of the rules for type judgement.

- Number literals:

$$\Gamma \vdash_h \mathsf{numberLiteral}[i] : \mathrm{numberType}(i)$$

where

$$\mathrm{numberType}(i) = \begin{cases} \mathsf{Nat} & \text{if } i \text{ is a natural number} \\ \mathsf{Int} & \text{if } i \text{ is a negative natural} \\ \mathsf{Number} & \text{otherwise} \end{cases}$$

We should also check that i actually represents a number. E.g. $\mathsf{numberLiteral}[\text{“abc”}]$ should be an error. Although it is not a type error.

- String literals

$$\Gamma \vdash_h \mathsf{stringLiteral}[i] : \mathsf{String}$$

- Boolean literals

$$\Gamma \vdash_h \mathsf{booleanLiteral}[b] : \mathsf{Bool}$$

- Null literals

$$\Gamma \vdash_h \mathsf{nullLiteral} : \mathsf{Null}$$

- Tuples

$$\frac{\Gamma \vdash_h \bar{e} : \bar{t}}{\Gamma \vdash_h \mathsf{tuple}\,(\bar{e}) : \langle \bar{t} \rangle}$$

- Variable rule

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash_h \mathsf{var}[x] : \mathrm{fetch}(h, \Gamma x)}$$

where

$$\begin{aligned} \mathrm{fetch}(\mathrm{L}, q) &= q \\ \mathrm{fetch}(\mathrm{R}, q) &= \begin{cases} u & \text{if } q = Loc[u] \\ q & \text{otherwise} \end{cases} \\ \mathrm{fetch}(h, p \sqcap q) &= \mathrm{fetch}(h, p) \sqcap \mathrm{fetch}(h, q) \\ \mathrm{fetch}(h, \top) &= \top \\ \mathrm{fetch}(h, t \sqcup u) &= \mathrm{fetch}(h, t) \sqcup \mathrm{fetch}(h, u) \\ \mathrm{fetch}(h, \bot) &= \bot \end{aligned}$$

- Lambda rules. Declaring the return type of a lambda is optional. If the type is declared, we check that the return type is a subtype the body.

$$\frac{\Gamma \vdash \mathsf{paramList}(\bar{d}) : \bar{t} \qquad \mathrm{extend}(\Gamma, \mathrm{vars}(\bar{d}), \bar{t}) \vdash_{\mathrm{R}} b : u \qquad u <: v}{\Gamma \vdash_h \mathsf{lambda}(\mathsf{params}(\bar{d}), v, b) : \langle \bar{t} \rangle \to v}$$

  If the return type is not declared, it is inferred from the body.

$$\frac{\Gamma \vdash \mathsf{paramList}(\bar{d}) : \bar{t} \qquad \mathrm{extend}(\Gamma, \mathrm{vars}(\bar{d}), \bar{t}) \vdash_{\mathrm{R}} b : u}{\Gamma \vdash_h \mathsf{lambda}(\mathsf{params}(\bar{d}), \mathsf{noType}, b) : \langle \bar{t} \rangle \to u}$$

  The extend function extends the environment with additional bindings, shadowing existing bindings if need be. Type-checking the parameter list yields a vector $\bar{t}$ of types as described later. In the case that $\bar{t}$ is a vector of length one, we take $\langle \bar{t} \rangle$ to mean the sole item of $\bar{t}$.

  When a return type is declared, we can determine the type of a lambda expression without type checking the body. This is useful in the first pass over an expression sequence, as described later:

$$\frac{\Gamma \vdash \mathsf{params}(\bar{d}) : \bar{t}}{\Gamma \vdash_h \mathsf{lambda}(\mathsf{params}(\bar{d}), v) : \langle \bar{t} \rangle \to v}$$

- Call rules. Any call expression with more or fewer than 2 operands can be treated as if it had 2. In particular $\mathsf{call}(e_0)$ is treated as $\mathsf{call}(e_0, \mathsf{tuple}())$ and $\mathsf{call}(e_0, e_1, \ldots, e_{n-1})$ is treated as $\mathsf{call}(e_0, \mathsf{tuple}(e_1, \ldots, e_{n-1}))$ when $n > 2$.

$$\frac{\Gamma \vdash_{\mathrm{R}} e_0 : t \qquad \Gamma \vdash_{\mathrm{R}} e_1 : u \qquad \mathrm{applicable}(t.u)}{\Gamma \vdash_h \mathsf{call}(e_0, e_1) : \mathrm{fetch}(h, \mathrm{returnType}(t.u))}$$

  where applicable and the returnType function are defined by

$$\begin{aligned}
\mathrm{applicable}(\bot, u) &= \mathit{false} \\
\mathrm{applicable}(t, \bot) &= \mathit{false} \\
\mathrm{applicable}(t \sqcup u, v) &= \mathrm{applicable}(t, v) \wedge \mathrm{applicable}(u, v) \\
\mathrm{applicable}(t, u \sqcup v) &= \mathrm{applicable}(t, u) \wedge \mathrm{applicable}(t, v) \\
\mathrm{applicable}(\top, s) &= \mathit{false} \\
\mathrm{applicable}(r_0 \sqcap r_1, s) &= \mathrm{applicable}(r_0, s) \vee \mathrm{applicable}(r_1, s) \\
\mathrm{applicable}(p, s) &= \mathit{false} \\
\mathrm{applicable}(\langle\rangle, s) &= \mathrm{applicable}(\langle t_0, t_1 \rangle, s) = \cdots = \mathit{false} \\
\mathrm{applicable}(t \to u, s) &= s <: t \\
\mathrm{applicable}(i\colon t, s) &= \mathit{false} \\
\mathrm{applicable}(\mathsf{Loc}[t], s) &= \mathit{false}
\end{aligned}$$

$$
\begin{array}{rcl}
\mathrm{returnType}(\bot, u) & = & \bot \\
\mathrm{returnType}(t, \bot) & = & \bot \\
\mathrm{returnType}(t \sqcup u, v) & = & \mathrm{returnType}(t, v) \sqcup \mathrm{returnType}(u, v) \\
\mathrm{returnType}(t, u \sqcup v) & = & \mathrm{returnType}(t, u) \sqcup \mathrm{returnType}(t, v) \\
\mathrm{returnType}(\top, s) & = & \top \\
\mathrm{returnType}(r_0 \sqcap r_1, s) & = & \mathrm{returnType}(r_0, s) \sqcap \mathrm{returnType}(r_1, s) \\
\mathrm{returnType}(p, s) & = & \top \\
\mathrm{returnType}(\langle\rangle, s) & = & \mathrm{returnType}(\langle t_0, t_1\rangle, s) = \cdots = \top \\
\mathrm{returnType}(t \to u, s) & = & \begin{cases} u & \text{if } s <: t \\ \top & \text{otherwise} \end{cases} \\
\mathrm{returnType}(i\colon t, s) & = & \top \\
\mathrm{returnType}(\mathsf{Loc}[t], s) & = & \top
\end{array}
$$

For example, if f has type $(\mathsf{length}\colon \mathsf{Nat} \sqcap (\mathsf{Nat} \to \mathsf{String}))$ and x has type Nat, then $\mathsf{call}(f, x)$ has type String; although $\mathsf{length}\colon \mathsf{Nat}$ is not applicable to Nat, $(\mathsf{Nat} \to \mathsf{String})$ is applicable to Nat. We use meet to combine the return type of $\mathsf{length}\colon \mathsf{Nat}$ applied to Nat, which is $\top$ with the result type of $(\mathsf{Nat} \to \mathsf{String})$ applied to Nat; the result is $\top \sqcap \mathsf{String} = \mathsf{String}$.

- An expression $\mathsf{callVar}[i](e_1, \ldots, e_{n-1})$ is typechecked the same as $\mathsf{call}(\mathsf{var}[i], e_1, \ldots, e_{n-1})$
- The dot rule

$$\frac{\Gamma \vdash_{\mathrm{R}} x : t \qquad \mathrm{fType}\left(\mathrm{rType}(t), i\right) = u}{\Gamma \vdash_h \mathsf{dot}\left[i\right](x) : \mathrm{fetch}(h, u)}$$

- The loc rule

$$\frac{\Gamma \vdash_{\mathrm{L}} x : t}{\Gamma \vdash_h \mathsf{loc}(x) : t}$$

- The assignment rule

$$\frac{\Gamma \vdash_{\mathrm{L}} x : t \qquad \Gamma \vdash_{\mathrm{R}} y : u \qquad \mathrm{assignable}(t, u)}{\Gamma \vdash \mathsf{assign}(x, y) : \langle\rangle}$$

where

$$
\begin{array}{rcl}
\mathrm{assignable}(\bot, v) & = & \mathit{false} \\
\mathrm{assignable}(t \sqcup u, v) & = & \mathrm{assignable}(t, v) \wedge \mathrm{assignable}(u, v) \\
\mathrm{assignable}(r, v) & = & \begin{cases} v <: t & \text{if } \mathrm{fType}\left(r, \mathsf{set}\right) = (t \to \langle\rangle) \not\equiv \top \text{ for some } t \\ \mathit{false} & \text{otherwise} \end{cases}
\end{array}
$$

[[Note. This rule needs to be amended to allow patterns such as tuples of location on the left.]] [[Apart from the need to deal with tuples, an assignment $\mathsf{assign}(x, y)$ is essentially an abbreviation for $\mathsf{call}(\mathsf{dot}\,[\mathsf{set}]\,(\mathsf{loc}(x)), y)$, so I should check that the type rules agrees.]]

- Expression Sequences. Expression sequences contain a mixture of variable declarations and expression. As long as all the variables are given an explicit type, there is no problem. However variables that don't have explicit types are assigned a type based on their initialization expression. One solution is to ban forward references, i.e., not allow a variable to be used before it is declared. However, that prevents mutually recursive subroutines and so is undesirable. Another approach is to try to do Hindley-Milner style inference; we won't do that yet; maybe someday. Instead we start by making two passes over the declarations. We start by assigning all variables declared in the initialization sequence a peudo-type of 'unknown'. Next we attempt to assign each variable (from left to right) a type based either on its explicit type or the type of its initialization expression. If determining the type of an initialization expression requires knowing the type of a variable declared later, there is an error. In particular the variables used in the body of a lambda do not need to be of known type if the lambda has an explicit return type. Let's call the result of this process $\mathrm{extend}(\Gamma, \mathsf{exprSeq}\,(\bar{m}))$. Once this new environment has been successfully computed, we can type check the expression sequence with the following rule, when there is at least one member.

$$\frac{\Gamma' = \mathrm{extend}(\Gamma, \bar{m}) \qquad \Gamma' \vdash_{\mathrm{L}} m_0 : t_0 \qquad \Gamma' \vdash_{\mathrm{L}} m_1 : t_1 \qquad \cdots \qquad \Gamma' \vdash_h m_{n-1} : t_{n-1}}{\Gamma \vdash_h \mathsf{exprSeq}\,(\bar{m}) : t_{n-1}}$$

  In the case of an empty sequence we have

$$\Gamma \vdash_h \mathsf{exprSeq}\,() : \langle\rangle$$

  Note that type checking does not catch use before definition errors. Such errors need a data-flow analysis.

- Variable declarations. Since the first pass of an expression sequence or object literal deals with determining the types of variables. The second pass is simply to check the type of the initialization expression

$$\frac{\Gamma \vdash_{\mathrm{R}} x : u}{\Gamma \vdash_h \mathsf{varDecl}[c]\,(\mathsf{var}[i], \mathsf{noType}, x) : \langle\rangle}$$

$$\frac{\Gamma \vdash_{\mathrm{R}} x : u \qquad u <: t}{\Gamma \vdash_h \mathsf{varDecl}[c]\,(\mathsf{var}[i], t, x) : \langle\rangle}$$

When there are no initialization expressions we have

$$\Gamma \vdash_h \mathsf{varDecl}[c]\,(\mathsf{var}[i], \mathsf{noType}, \mathsf{noExp}) : \langle\rangle$$

$$\Gamma \vdash_h \mathsf{varDecl}[c]\,(\mathsf{var}[i], t, \mathsf{noExp}) : \langle\rangle$$

[Note that the case of no type and no initializtion will actually be flagged as an error in the first pass over the expression list. Thus there is no need to flag the error in the typing pass.]

- If
$$\frac{\Gamma \vdash_{\mathrm{R}} e : t_0 \qquad t_0 <: \mathsf{Bool} \qquad \Gamma \vdash_h seq_0 : u_0 \qquad \Gamma \vdash_h seq_1 : u_1}{\Gamma \vdash_h \mathsf{if}(e, seq_0, seq_1) : u_0 \sqcup u_1}$$

- While loops
$$\frac{\Gamma \vdash_{\mathrm{R}} e : t_0 \qquad t_0 <: \mathsf{Bool} \qquad \Gamma \vdash_h seq : u}{\Gamma \vdash_h \mathsf{while}\,(e, seq) : \langle\rangle}$$

- Object literal. We make the same first pass and second pass as for an expression sequence (except that there is no need to evaluate the last member in anything other than an L context)
$$\frac{\Gamma' = \mathrm{extend}(\Gamma, \bar{m}) \qquad \Gamma' \vdash_{\mathrm{L}} \bar{m} : \bar{t}}{\Gamma \vdash_h \mathsf{objectLiteral}\,(\bar{m}) : \mathrm{extract}(\bar{m}, \Gamma')}$$
The $\mathrm{extract}(\bar{e}, \Gamma')$ function simply retrieves the name/type pairs for each variable declaration in $\bar{m}$, forms a field type for each, and then meets those field types togther. [[Details to be filled in, perhaps]]

- Place Holders. No rule. Hence place holder are always an error.

3.7.2 Soundness of the typing rules

The static typing rules are sound if whenever

$$\Gamma \vdash_h m : t$$

evaluating the member m, with respect to an environment that agrees with Γ and in an h context, results in either nontermination, error, or a value v such that $v \in \mathrm{Val}(t)$. Of course to prove soundness, I'd have to give a formal definition of evaluation. Maybe someday.