Closet: A Framework For Combining Scripting and Static Languages

Andy Boothe

andy.boothe@gmail.com

University of Texas at Austin

Department of Computer Sciences

Undergraduate Honors Thesis

Spring 2006

Abstract

Scripting languages like Python and Ruby allow rapid development of software, but make no guarantees about the type safety of a program at compile time. While static languages like Java do make strong guarantees about the type safety of a program at compile time, they are not perceived to be as agile as their more dynamic counterparts. Is it possible to develop a statically-typed language that makes strong guarantees about type safety, but looks and acts like a scripting language?

Table of Contents

1.0 Introduction

As computers have become more and more important in industry and in the lives of people around the world, the demand for software has grown with tremendous speed. Rapid growth continues to this day, and there's no reason to believe this growth will slow in the foreseeable future. As the demand for software increases, the supply of software must increase as well. Therefore, the ability to write programs to solve complex problems quickly and correctly can only become more important.

All modern programs are written in programming languages. Therefore, to write more powerful programs quickly and correctly, we will need more expressive programming languages that help us to write programs quickly and correctly. The basis for such languages exist today; indeed, some good examples are already well defined with reliable implementations and in wide use. Languages such as Ruby, PHP, Python, and Perl are touted as enabling programmers to write programs extremely quickly, whereas languages such as Java, C++, and C# are popular for writing large, mission-critical applications out of libraries of reusable components. However, there is not yet any one language suitable for both of these pursuits. In fact, quite the opposite is true. There are two distinct camps which embody one - but not both - of these goals: the scripting languages camp focuses on languages which are agile but unsafe, whereas the static languages camp focuses on languages which are safe but not too agile.

In the interest of completeness, there is one other language camp which bears mention: functional languages. While these languages are inspiring, their popularity pales in comparison to the popularity of imperative languages. However, functional languages have shown themselves to be a potent proving ground for new ideas in programming languages, and there are many ideas in modern functional languages that are very powerful, but still have not yet crossed over to the other language paradigms. These features and ideas must not be ignored, and instead should be adapted to new agile languages that are also safe.

Some canonical examples of modern scripting languages are Python, Ruby, and Perl. So-called scripting languages are largely a reaction to the so-called ``B&D"[1] languages which scripting proponents claim force the programmer to focus on unimportant details such as types, thereby distracting the programmer from his real task: reasoning about and writing correct and efficient programs. As a result, scripting languages are written to afford the programmer as much flexibility as possible and to free his mind from the mundane tasks of programming; the main focus is on writing programs quickly. Scripting languages are often dynamically typed, feature high-level language features like true closures and functions as objects, and have elaborate syntax which allows the programmer to perform complex computations with concise syntax. While these properties of scripting languages make programming in scripting languages faster, they greatly reduce the power of the compiler or interpreter to perform static analysis and identify errors. For example, in a dynamic typing system types are not known until runtime, so the language cannot make compile-time guarantees about type safety. This makes dynamically-typed programs susceptible to an entire class of bugs to which statically-typed languages are

1 ``B&D" is a colloquialism for ``bondage and disciple" and is usually applied to languages which are highly restrictive.

largely immune.

On the other end of the spectrum there are static languages. Some canonical examples of modern static languages are Java, C, C++, and C#. Static languages are designed to eliminate as many errors as possible without disallowing the programmer from performing arbitrary calculations; the main focus is on writing programs correctly. As a result of this philosophy, static languages are usually simpler, have fewer languages features, and are explicitly typed. Explicit typing is a form of static typing, so static languages are empowered to make strong guarantees about the type safety of programs. This combination of factors results in greatly reducing the number of programmer errors at runtime. However, explicit typing also requires the programmer to specify the type of every variable and function explicitly, which makes both the process of writing new code and the process of changing existing code longer and more tedious. Also, popular imperative languages often lack critical language features like closures, programmers must often duplicate this functionality in their programs by hand, resulting in both duplicated code and longer code, in addition to increased tedium.

One of the most striking differences between these two types of languages is the **type system** each uses, where the ``the fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program" [Cardelli97]. Type systems are implemented using a **type checker**, which is used to determine whether or not a program conforms to a language's type system. Scripting languages are typically dynamically typed, which means that the language uses a trivial compile-time type system system in which all values are of the same type, so the compile-time type system has no power to eliminate potential runtime errors. Static languages use static typing, so they use a more meaningful compile-time type system which knows intimate details about the type of each value in the program, which allows the language to eliminate entire classes of runtime errors.

Another important property of a language is its **safety**. ``A program fragment is **safe** if it does not cause any untrapped errors to occur. Languages where all program fragments are safe are called **safe languages**. Therefore, safe languages rule out the most insidious form of execution errors: the ones that may go unnoticed" [Cardelli97]. Safety in a high-level language is highly desirable, and both static and scripting languages tend to be safe.

Language	*Typing*	*Safe?*	*Inferred?*	*Paradigm*	*GC?*
C	Weak	No	No	Imperative	No
C++	Weak/Strong	Yes/No	No	Imperative, OO	Yes/No
C#	Strong	Yes/No	No	Imperative, OO	Yes
Java	Strong	Yes	No	OO	Yes
Python	Dynamic	Yes	No	Scripting, OO	Yes
PHP	Dynamic	Yes	No	Scripting, OO	Yes
Ruby	Dynamic	Yes	No	Scripting, OO	Yes
Haskell	Stronger	Yes	Yes	Functional	Yes
ML	Stronger	Yes	Yes	Functional	Yes
Lisp/Scheme	Dynamic	No	No	Functional	Yes

Figure 1.1: Examples of Programming Languages

Judging by the popularity of both types of language and the near-religious zeal with which people defend either philosophy, scripting languages and static languages both have gotten some things right. However, as proponents of each school are eager to point out about the other, both have also gotten things wrong. However, many of the drawbacks of both static and scripting languages are avoidable. This begs the question: is it possible to create a sort of hybrid language with the advantages of both types of language and the disadvantages of neither? Can a scripting language be made `safe,' or can a static language with strong typing be made `agile'?

2.0 What Do Programmers Like About Languages?

To better understand why programmers like scripting languages, contrast the following equivalent code written in two languages: Ruby, a scripting language; and Java, a static language.

2.1 Case Study: qsort

Consider these implementations of quicksort (`qsort`) in Ruby and Java:

```
def qsort(vs)
  if vs.length<=1
    return vs
  else
    pivot = vs.pop()
    return qsort(vs.find_all do |v| v < pivot end) +
```

```
            [pivot] +
            qsort(vs.find_all do |v| v >= pivot end)
  end
end
```

Code Listing 2.1.1: qsort in Ruby

```
public static List<Comparable> qsort(List<Comparable> vs) {
  if(vs.size() <= 1)
    return vs;
  else {
    Comparable pivot=vs.remove(vs.size()/2);
    List<Comparable> left=new ArrayList<Comparable>();
    List<Comparable> right=new ArrayList<Comparable>();

    for(Comparable v : vs)
      if(v.compareTo(pivot) < 0)
        left.add(v);
      else
        right.add(v);

    List<Comparable> result=qsort(left);
    result.add(pivot);
    result.addAll(qsort(right));

    return result;
  }
}
```

Code Listing 2.1.2: qsort in Java

To be precise, these two implementations are not *exactly* the same. (The Java version has one loop whereas the Ruby version has two, for example.) However, the behavior they implement is the same, so we will consider the two passages to be equivalent.

Both functions sort any collection of ordered objects in the most general way possible in the language. In Ruby, we get this polymorphism syntactically for free; because Ruby is dynamically typed, all checks to see if the arguments support the required operations `<`, `length`, `pop`, and so on are deferred until runtime; therefore, the compiler allows any value to be passed

to qsort, including values which will generate a runtime typing error. However, in Java we have to use the verbose generics syntax introduced in Java5. This added verbosity empowers the compiler to guarantee that no uses of qsort will generate a runtime typing error.

When comparing these two implementations, the most obvious difference is the lengths of the two snippets. The Ruby implementation is significantly shorter than the Java implementation, weighing in at a third as many lines and less than half as many characters, partly due to some of Ruby's language features and partly due to the lack of type declarations. Also, the Ruby code looks much less complex than the Java code and is much more clear and readable. Furthermore, Ruby uses fewer symbols, preferring instead keywords (e.g. **`end`** instead of **`}`**), so the code actually reads like English. The aforementioned type-safety guarantee of the Java program, however, is also attractive. Equally as attractive, while it's not obvious from the code, the Java code will both start faster and run faster because Java is a byte-code compiled language whereas Ruby is an interpreted language (although Ruby could in principle be compiled to byte code as well).

For the sake of argument, let us assume that these differences represent the reasons why programmers choose one type of language over the other. People like the brevity of scripting languages and the for free flexibility offered by the dynamic typing system as well as the greater readability. On the other hand, people also appreciate the guarantees that a static language's compiler makes, even at the expense of longer code, and the speed it offers.

2.2 Can Scripting Languages and Static Languages be Combined?

There are many lessons to learn from both static and scripting languages:

- shorter programs are better
- more readable programs are better
- guarantees are a good thing
- flexibility is a good thing
- faster is better than slower
- more features are better than fewer features, within reason

How many of these pros can be combined into one language while avoiding as many cons as possible?

It is possible to achieve compile-time type safety guarantees without explicit typing using another static typing system called **type inferencing**. In type inferencing, the types of variables and expressions are determined at compile time by analyzing the way the variables are used instead of by explicit declaration. Therefore, the same type information is inferred without having to write it explicitly, so type safety guarantees are achieved without sacrificing conciseness or visual simplicity. It is important to note, however, that no type inferencing system can necessarily infer the type of all useful programming expressions, so some type information must be stated explicitly from time to time.

Garbage collection is also a feature of the majority of modern popular languages and it appears

in both scripting languages and in static languages. In fact, garbage collectors are critical features in languages. Garbage collectors are important in avoiding many varieties of memory management errors. Clearly, garbage collectors are a good thing in high-level languages.

There are a few other crucial features that have proved to be important in modern languages. One such feature is **closures**. Closures offer a sophisticated versatility while still preserving modularity, so they allow many programming idioms that are good solutions to difficult problems. A related feature is the **lambda expression**, which is borrowed from functional languages. Lambda expressions embody a functional abstraction of mapping values from one type to another in a straightforward manner, so they are a very clear way of expression many ideas in programming. Certain syntactic sugars, such as support for list literals, are also very useful.

3.0 Closet

Closet is an experimental language designed to explore how to marry the principles of both static languages and scripting languages together in a single language. Specifically, the goal for Closet was to make a language which looks and acts like a scripting language but can make all of the guarantees of a static language.

Closet has the following features:

- *Static Typing* the types of Closet programs are determined at compile time. Programs are guaranteed to be type-safe before they ever run, excluding downcasts, which are checked dynamically at runtime.
- *Type Inferencing* Closet programs are largely implicitly typed. While function parameters, return types, and member variables are explicitly typed, local variables are implicitly typed, eliminating much of the need for type annotations on variables.
- *Garbage Collection* memory is managed by a garbage collector, so user memory management is automatic.
- *Purely Object-Oriented* all values in Closet are objects. This includes user-defined types as well as built-in types such as integers and functions. Therefore, higher-order functions are possible in (and important to) Closet.
- *Functions as Closures* all functions in Closet are closures, which offers the basis of implementations for objects in Closet.

The unit of compilation in Closet is the file.

A grammar for Closet is in Appendix A of this paper. A synopsis of Closet's type structure and behavior are given in Appendices B and C.

3.1 Closet by Example

Closet is designed to be readable and safe while still being short and expressive. It is also

designed to be highly orthogonal, so a few short examples will offer a good working understanding of the language.

This is the trivial Hello World program in Closet:

```
function void main():
      puts("Hello, World!")
end
```

Code Listing 3.1.1: Hello World in Closet

The program consists of one function called `main` with type `()->void`, or a function with zero arguments and return type `void`, which is the entry point for Closet programs. The body of `main` contains one statement which calls `puts` passing the `String` value `"Hello, World!"` as an argument. The `puts` function writes its argument to the output. Note that statements do not require an explicit statement terminator; a newline is sufficient.

```
function Boolean evenp(n:Integer):
  return n%2==0
end
```

Code Listing 3.1.2: Predicate for Evenness of Integers in Closet

This example defines a function called `evenp` with type `(Integer)->Boolean`, or a function with one argument of type `Integer` and return type `Boolean`, that returns `true` if its `Integer` argument is even (i.e. has remainder zero when divided by two) and `false` otherwise.

```
function List qsort(xs:List):
  var result
  if xs.size() <= 1:
    result = xs
  else: do
    var pivot=(cast Integer) xs[0]
    var l=qsort(xs[1:].filter(lambda(x:Object)->pivot>=(cast
          Integer)x))
    var r=qsort(xs[1:].filter(lambda(x:Object)->pivot<(cast
          Integer)x))
    result = l + [pivot] + r
  end
  return result
```

```
end
```

Code Listing 3.1.3: quicksort in Closet

This code snippet defines a function `qsort` that implements the recursive quicksort algorithm. It is important to note that this quicksort is not generic as the quicksort algorithms from the introduction are; this quicksort will only work on lists of `Integer`s. A discussion of an augmentation to the type system to include generics, which would allow a type-safe polymorphic `qsort` is included in the Future Work section. This snippet also demonstrates Closet's type inferencing system. The locked variable `xs` has type `List` by explicit typing. The types of variables `pivot`, `l`, and `r` are `Integer`, `List`, and `List` .Somewhat more interesting is that the variable `result` has the type `List`, which is the result of `unify(List, List)`. Closet's type inferencing system is discussed in section 3.4.

```
function ()->Integer generator():
  var n = 1
  function Integer g():
    var result = n
    n = n+1
    return result
  end
  return g
end
```

Code Listing 3.1.4: Sequence generator

This code snippet further illustrates Closet's typing system and shows an example of Closet's closures and of a higher-order function in Closet. The function `generator` is a higher-order function with the type `()->()->Integer` and the function `g` has type `()->Integer`. The variable `n` receives the type `Integer` via its initialization.

```
function Integer geni():
        return 5
end

function Object geno():
        return geni()
end

var f
```

```
f = geni
f = geno
```

Code Listing 3.1.5: Function Type Inferencing

This snippet illustrates the how Closet's type system handles function types. The function `geni` has type `()->Integer` and `geno` has the type `()->Object`; since `f` receives the value of both, it receives the most general type, `()->Object`.

```
function Object id(x:Object):
      return x
end
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].map(id)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].map(lambda(x:Object) -> x)
```

Code Listing 3.1.6: Function Type Inferencing

This code example shows the use of functions as first-class objects and introduces the Closet `List` type. A function `id` is defined, which implements the identity function, and is used as an argument to the `List#map` function, which takes one argument of type `(Object)->Object`. An example of using a lambda expression as a first-class object is also given. The lambda argument implements the same function as `id`.

```
function String printHierarchy(C:Class):
  var result
  if C.getParent() === null: do
    puts("  "+C)
    result = "-"
  else: do
    var pfx = printHierarchy(C.getParent())
    puts("  |"+pfx+C)
    result = pfx+"--"
  end
  return result
end
printHierarchy(Object)
printHierarchy(5.getClass())
```

Code Listing 3.1.7: Example of Closet's Pure Object-Oriented Nature

This example shows the treatment of everything, even integer literals and classes, as an object. The function `printHierarchy` will print the path along the inheritance hierarchy from the given class to the root class, `Object`. Any object may be queried for its class using the `getClass` method.

3.2 Closet Features

Closet is a purely Object-Oriented language, so all values in Closet, including classes and functions, are objects.

The unit of compilation of Closet is the file. Therefore, all code for a Closet program must be included in a single file since there is no provision at this time for including code from another file. (It is important to note that this is purely a reflection of the experimental nature of Closet.)

Closet uses a single-inheritance class-based model for types. Except for the class `Object`, every class in Closet has exactly one parent. `Object` has zero parents because it is the root of the inheritance hierarchy. Because it is the root of the inheritance hierarchy, `Object` is an ancestor of every class in Closet, be it user-defined or built-in. Overriding methods in Closet must have the same parameter types, but are allowed to have more specific return types, so a method `foo` with type `()->Integer` in a daughter class could override a method `foo` with type `()->Object` in its parent class, for example.

Functions in Closet are first-class objects, so higher-order functions in Closet are both common and encouraged. All functions (and methods) in Closet are closures.

Since functions are closures, name lookup in functions is performed in typical closure fashion: name lookup begins in the outermost scope, then continuing to the next outermost scope, and so on until either the name is found, which indicates success, or the list of active scopes is exhausted, which is a compile-time error. Local variable lookup in methods is the same. It is important to note that instance variables and methods are only available to methods through the const variable `self`, which is the object on which a method was invoked (or to which a method was bound in the case of bound methods, which are discussed below). Values in `self` are resolved by looking in the class of `self`, then in the parent class of `self`, and so on. Lookup in `self` is performed according to the dynamic type of `self`, not the static type.

Method invocation on objects perform the same in Closet as they do in Java. If an object `a` of type `A` exists with the method `foo()` defined on it, invoking `a.foo()` invokes `foo()` with the name `self` bound to the value of `a`. However, an additional syntax for specifying methods as values is available. Because methods are meaningless without an object upon which to act, though, method values in Closet do not exist. Instead, if a method is specified as a value, a special function called a **bound method** is returned. For example, the way to specify the function `foo` invoked on `a` is `a.foo` instead of something like `lambda(a:A)->a.foo()`. To implement this behavior, the runtime system detects the conditions where a method is specified as a value and automatically creates a clone of the method specified and binds the name `self` to the object from which the method was specified by adding an `Env` to the function's closure environment. (This is always acceptable

behavior because all functions in Closet are closures.) So, in the case of `a.foo`, the runtime copies `foo`, binds `self` to `a` in the closure environment of this copy of `foo`, and returns the result, which is a stand-alone function. (This behavior is the same as bound method behavior in Python except that binding takes place only when a method is specified as a stand-alone value instead of every time a method is called.)

Objects in Closet have two kinds of members: methods and member variables. Function members and class member variables are notably absent.

Objects in Closet are passed by reference, and references are passed by value. Since everything in Closet is an object, all variables and all values in Closet are references to objects.

Operators like + are implemented using method calls. Taking a hint from Python, the Closet parser parses each operator as a call to a function with a special name. For example, the expression `a+b` parses to `a.__add__(b)`.

3.3 Closet Expressions

There are twenty types of expressions in Closet:

1. `And`
2. `Assign`
3. `Call`
4. `Cast`
5. `Dot`
6. `Get`
7. `Integer`
8. `IsA`
9. `IsSame`
10. `Lambda`
11. `List`
12. `New`
13. `Not`
14. `Null`
15. `Or`
16. `Self`
17. `Slice`
18. `String`
19. `Super`
20. `Sym`

The definitions of each of Closet's expressions are defined in Appendix B.

3.4 Closet's Type System

In order to allow the programmer to ignore variable type declarations as much as possible (in deference to scripting languages) while still offering static type checking (in deference to static languages), Closet uses a type inferencing system to infer the types of variables.

3.4.1 Overview

In Closet's type system, each variable has exactly one type for its lifetime. It is tempting to have the type system remember exactly what type a variable has at any given time, thus allowing a variable to have many types through its lifetime, but because it is impossible to determine when closures which capture variables will execute this is not practical. Consider the example:

```
var a
a = 5
function Integer bar():
  return a
end
a = new Object()
function Object foo():
  return a
end
bar()
```

Code Listing 3.4.1: The Intractability of Alternative Type System

It is tempting to allow `a` to have type `Integer` in `bar` and type `Object` in `foo`, but in this code snippet `a` would only have type `Integer` until `a=new Object()` is evaluated, after which point `a` would have type `Object`, so the expression `foo()` would be a type error because it returns `a`, which has type `Object`, as an `Integer`. Because it is impossible to decide all the times when `foo` is called and guarantee that all calls occur before the evaluation of `a=5` in general, this type system is intractable. (It is interesting to note, however, that in the absence of closures, or when closures only capture fixed-type variables, this approach is tractable.) Therefore, all variables have exactly one type during their lifetime.

3.4.2 Variables

Closet allows three kinds of variables: **free**, **locked**, and **const**. A free variable is a variable whose type is determined by the type inferencing system. A locked variable is a variable whose type is explicitly specified in the program text. A **const** variable is a variable whose value is fixed for the entirety of the program; because the value of a const variable never changes after initialization, const variables are also necessarily locked, and const variables cannot be the lvalue of an assignment expression. Functions and classes are automatically const.

```
var a                    # Free variable
var c:String             # Locked variable; type String
function void d() {}     # Const variable; type ()->void
```

Code Listing 3.4.2.1: Variable declarations in Closet

3.4.3 Types

There are four kinds of types in Closet: **simple**, **compound**, **null**, and **unit**.

Simple types are class types (e.g. `Object`).

Compound types represent function types. They have a domain and a range, with the domain being a tuple of zero or more types representing argument types and the range being a single type representing the result type of the function. Only compound types are callable. All compound types are child types of the simple type `Function`.

The null type is a special, singleton type which represents the absence of type information, as in the case of an uninitialized free variable which has not yet been the target of an assignment.

The unit type is another special, singleton type which represents the void type.

3.4.4 Type Inferencing Algorithm

At the top level, the program is a list of statements. The interpreter infers the types of each of these top-level statements, such as function statements, which causes the recursive inferring of any substatements within these statements, which causes recursive inferring of any substatements within these statements, and so on. Eventually, the inferring of a statement involves the inferring of one or more expressions, at which point the interpreter uses the typing rules for expressions given in Appendix B to build the typing environment, which it then uses to determine the types of the program's variables. Because of the way the typing rules are laid out, the types of free variables may become more general as the typing process moves from statement to statement. As a result, the type inferencing process may need to an arbitrarily large – but finite – number of passes to fully resolve types.

At a high level, the type inferencing system propagates type information from typed expressions to free variables through assignment statements. The type assigned to a free variable is the most specific type which accepts the values of all assignments. The type system uses the function `unify` to calculate this most specific type.

Given two types, the `unify` function will either return the most specific type which can hold references to values of both types or signal an error. `unify(t1, t2)` represents the assignment of a value with type `t2` to a free variable which currently has type `t1`. `unify` has the following definition:

```
unify(Null t1, Simple t2) = t2
unify(Null t1, Compound t2) = t2
unify(Null t1, Null t2) = t2
unify(Unit t1, Unit t2) = t1
unify(Simple t1, Simple t2) = mostSpecificParentOf(t1, t2)
unify(Simple t1, Compound t2) = unify(t1, Function)
```

```
unify(Compound t1, Simple t2) = unify(Function, t2)
unify(Compound t1, Compound t2) =
  if accepts(t1, t2)
    t1
  else
  if accepts(t2, t1)
    t2
  else
    Function
```

Code Listing 3.4.4.1: Pseudocode for unify

The `accepts` function is used to determine if one type accepts assignments from another type. `accept(t1, t2)` is `true` if `t1` accepts values of `t2`, or `false` otherwise. The definition of `accepts` is:

```
accepts(Simple t1, Simple t2) = isParentOf(t1, t2)
accepts(Simple t1, Compound t2) = accepts(t1, Function)
accepts(Null t1, Simple t2) = true
accepts(Null t1, Compound t2) = true
accepts(Compound t1, Simple t2) = false
accepts(Compound t1, Compound t2) =
  if t1.args.len != t2.args.len
    false
  else
  if for i in 0..t1.args.len, accepts(t1.args[i], t2.args[i])
    if accepts(t1.result, t2.result)
      true
    else
      false
  else
  if for i in 0..t2.args.len, accepts(t2.args[i], t1.args[i])
    if accepts(t2.result, t1.result)
      true
    else
      false
  else
```

```
    false
```

Code listing 3.4.4.2: Pseudocode for accepts

The null type is the type of an uninitialized variable, so using the null type as the right-hand side of an assignment is an error because it indicates the use of an uninitialized variable, which is not safe because Closet makes no guarantees about the values of uninitialized local variables and so is therefore an error. However, using the null type on the left-hand side is simply the first assignment to an uninitialized free variable, so the resulting type should be the right-hand side type.

The unit type is the void type, so using the unit type as the right-hand side of an assignment is an error because it is an assignment from void, which is semantically illegal.

Unifying two simple types results in the most specific type which both types share, whereas unifying a simple type `t1` and a compound type `t2` is equivalent to unifying the simple type `Function` and type `t1` since compound types and simple types are not compatible as-is.

Unifying two compound types results in either a new compound type if the two compound types are compatible, or the simple type Function if they are not. Two function types `t1` and `t2` are compatible if either `t1` accepts `t2` or `t2` accepts `t1`. `t1` accepts `t2` if `t1` and `t2` have the same number of arguments, the return type of `t1` accepts the return type of `t2`, and the each argument type of `t1` accepts the corresponding argument type of `t2`. If `t1` accepts `t2`, then the result type is `t1`.

Because if unify returns it returns the most specific type which accepts assignments from both types, the resulting type will always be at least as general as each of the parameters. Therefore, because the type of a variable receives its ultimate type from iterative applications of unify, the type of a free variable can only get more general with time.

Functions in Closet are const variables, so function parameters are always locked variables and return types are specified explicitly. Methods are treated in the same way.

Consider the example:

```
function void main():
      var a=0
      var b=0
      var c=0
      var d=0

      d = c
      c = b
```

```
        b = a
        a = new Object()
end
```

Code Listing 3.4.3: Contrived Type Inferencing Example

This is a good example of a program that takes the type inferencing algorithm many passes to resolve. Because of the order of the assignment statements and the assignment relationships among the variables, this snippet takes 5 passes to fully resolve. Because each of the variables `a`, `b`, `c`, and `d` receive both `Integer` and `Object` values, the correct type for each variable is the type `Object`. After the first pass, `a` has the correct type `Object`; after the second, `b` has the correct type of `Object`, after the third, `c`, after the fourth, `d`, and the last pass makes no changes to the types of any variables in the program, indicating that the algorithm has fully resolved all types and may terminate. The reason that this program requires so many passes to resolve is that type information can only trickle downward, not upward, through a series of statements. The type inferencing algorithm infers types beginning with the first statement in a list of statements and moving towards the end. Therefore, if the type t_a of variable `a` is generalized to type t_a' at statement `n`, then any statement `m` that occurs before statement `n` and involves `a` must be revisited, so another pass must be made because such statements will not be visited on the current pass. It is possible to create a program that has arbitrarily many statements arranged in this way, so a program may require arbitrarily many passes to resolve all of its types. However, in practice almost no program requires more than two or three passes.

3.5 Closet Implementation

Closet's implementation is fairly straightforward. Complexity was avoided by implementing an interpreter instead of a compiler or virtual machine.

3.5.1 Design

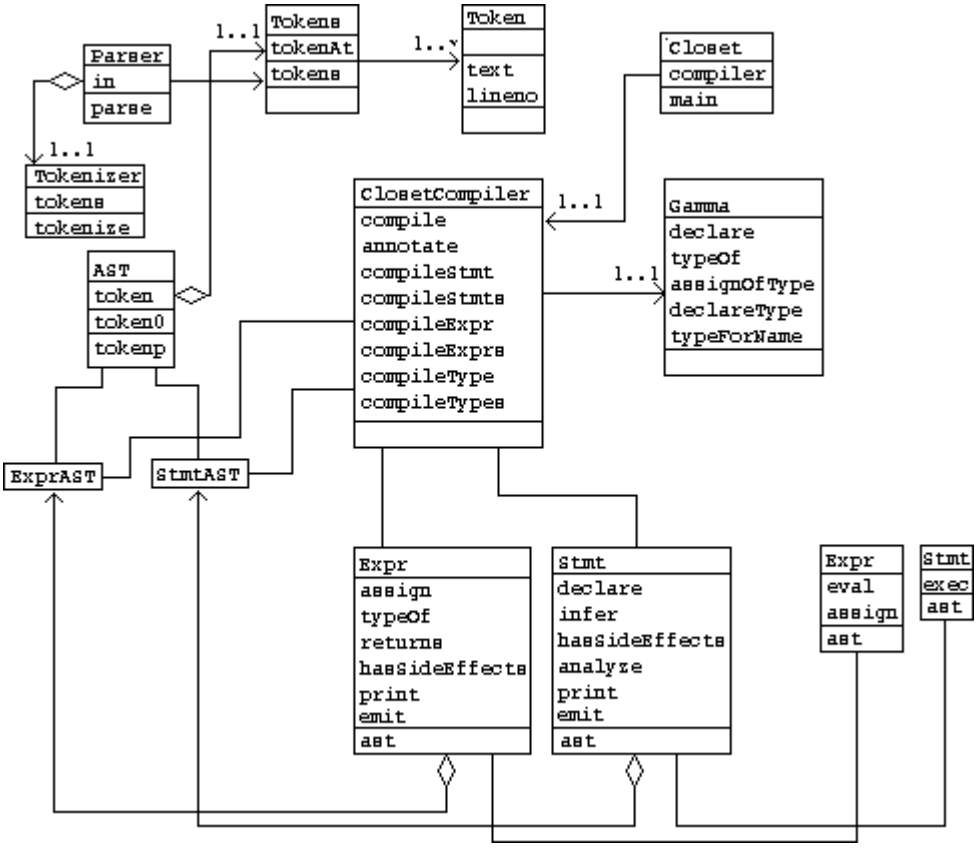

Figure 3.5.1.1: UML Diagram of Closet Implementation

The Closet interpreter is a set of approximately 200 Java classes which implement the Closet language. The Closet interpreter was designed and implemented to gain insight into both the process of designing languages in general and the mechanics of Closet.

There is a class for the compile-time version, run-time version, and abstract syntax tree of each of Closet's twenty expression types and ten statement types.

3.5.1.1 Program Execution

The process of running a Closet program can be broken down into several phases:

- lexical analysis
- parsing
- compilation
- inference
- analysis
- emission

- interpretation.

Phase 1: Lexical Analysis

The lexing phase consists of the lexical analysis of the program source file. The `Tokenizer` class takes a `java.io.InputStream` and converts it into an array of `Token` objects. There are three categories of tokens: atoms, keywords, and operators.

Atoms are the syntactic representations of the atomic values of Closet: `String`s, `Integer`s, and variable names.

Keywords are combined to produce Closet's syntactic forms. Closet's keywords are **`function`**, **`return`**, **`if`**, **`else`**, **`while`**, **`for`**, **`self`**, **`null`**, **`var`**, **`type`**, **`void`**, **`class`**, **`extends`**, **`method`**, **`cast`**, **`new`**, **`isa`**, **`true`**, **`false`**, **`super`**, **`lambda`**, **`do`**, **`end`**.

Operators are both Closet's separators and operators. Closet's operators are **`+`**, **`(`**, **`)`**, **`,`**, **`{`**, **`}`**, **`=`**, **`;`**, **`.`**, **`-`**, **`*`**, **`/`**, **`%`**, **`->`**, **`:`**, **`===`**, **`==`**, **`!==`**, **`!=`**, **`<`**, **`<=`**, **`>=`**, **`>`**, **`&&`**, **`||`**, **`[`**, **`]`**.

If the lexer cannot lex the given input stream, it will issue a tokenization error and exit.

Phase 2: Parsing

Parsing is the process of assigning syntactical form to the unprocessed list of tokens produced by the lexer.

Closet uses a hand-written top-down recursive-descent parser `Parser` to transform the `Tokens` object produced by the `Tokenizer` into an array of `StmtAST` objects. The `Parser` object implements the BNF in Appendix A.

If the parser cannot understand the given `Tokens`, it will issue a parsing error and exit.

Phase 3: Compilation

In the compilation phase, the syntactic forms are transformed into an in-memory representation of the program.

The Closet interpreter uses a `ClosetCompiler` object to transform the `StmtAST` objects into compile-time `Stmt` objects.

Phase 4: Inference

In the inference phase, the type inferencing algorithm is run against the in-memory representation of the program to determine the types of the variables in the program.

The `ClosetCompiler` object from the compile phase is responsible for invoking the `declare`

method exactly once on each element of the list of program statements, which will build the initial typing environment for the inference step, and then for iteratively invoking the `infer` method on the list of program statements, which will populate the typing environment according to the typing rules in Appendix B and recursively invoke `declare` and then `infer` on their substatements, until the type inferencing process is complete.

If the inference phase encounters any fatal errors, such as undeclared read errors, then it will issue an error and exit.

Phase 5: Analysis

In the analysis phase, the in-memory representation of the program produced during the compilation phase is analyzed for proper inheritance, reachability of code, and various other semantical errors.

The `ClosetCompiler` object from the compilation phase is responsible for invoking the `analyze` method on the compile-time `Stmt` objects, which implement the checks themselves.

If the analysis phase uncovers any fatal errors, such as a control path in a function which does not return a value, then it will issue an error and exit.

Phase 6: Emission

The emission phase sees the transformation of the in-memory, compile-time representation of the program to the in-memory, run-time representation of the program.

The `ClosetCompiler` object from the compilation phase is responsible for invoking the `emit` method on the compile-time `Stmt` objects, which then return the corresponding run-time `Stmt` object, usually by calling `emit` recursively on substatements.

Phase 7: Interpretation

The interpretation phase is when the program code is actually executed.

The `Closet` class, which is the main class of the Closet interpreter, takes the set of run-time `Stmt` objects which the `ClosetCompiler` object creates and invokes the `exec` method of each one in turn and then calls the `main` method of the program.

If the program performs an illegal operation, like an illegal dynamic cast or a division by zero, then an exception will be signaled and the interpreter will exit.

3.5.2 Closet Errors

The Closet interpreter recognizes 31 compile-time errors, 3 compile-time warnings, and 5

runtime errors.

3.5.3 Performance

Since Closet is a framework for experimentation, the focus in designing Closet was on maximizing ease of experimentation. However, an effort was made to design Closet so that its performance ultimately could be comparable to the performance of other static languages. To be clear, the performance of this implementation is not at all comparable, but the language was designed so that such an implementation *could* be made.

Because variables are completely statically scoped, general variable lookup can be performed on a set of nested `Env` structures built at runtime by using a pair of integer indexes `depth` and `slot` determined at compile time, where `depth` is the number of `Env`s back to go from the top `Env` and `slot` is the index in that `Env` to use, and trivial local-scope lookup can be done with a single index. Because the vast majority of variable fetches will be either fetches from an object or trivial local fetches, variable fetching performance in Closet will be close to the performance of that of other languages.

Method resolution in Closet can be done in the same way method resolution is performed in Java: methods are resolved expensively only the first time they are called and then trivially thereafter using thunking.

4.0 Discussion

4.1 Closet Design

The design of Closet was a significant undertaking. No other purely object-oriented languages embracing type inferencing exist, so there was no prior art to study for inspiration or direction. Including so-called dynamic features in a static language also offered another important challenge in terms of deciding what to include and what to leave out.

The complexity of a language should lie in its library, not in its core. Therefore, the overall design philosophy for Closet was simple:

1. Keep the language core as small as possible
2. Include only critical dynamic features.

Ultimately, Closet's core did end up very small: functions are closures; everything is an object; all variable lookups occur hierarchically in an Env. The only really dynamic feature included in Closet is closures. The small number of dynamic features included is a result of the primary litmus test used to determine whether or not a feature is critical: `does anybody ever use this feature?' Most languages contain features which are rarely, if ever, used because language authors are paranoid about leaving out a key feature. (Break labels in Java and continuations in Ruby leap to mind, for example.) In Closet, a feature was included only if it was determined to

be important and it could not be produced by the other language features already included. For example, many of the dynamic-looking features of Closet are actually just syntactic sugar: List literals, for example, and list indexing using [].

Closet was also designed with performance in mind as a final steering mechanism, although this was far from the foremost concern.

4.2 Successes

Closet was successful in showing that a language that looks and acts like a scripting language but has the guarantees of a static language can be made. The use of type inferencing to determine types at compile time without explicit type annotation was shown not only to be possible but also to be very effective and practical in real-life programs.

4.3 Failures

While Closet did fulfill its role as an investigation of the marriage of static and scripting languages, it is still only a prototype and therefore lacks many desirable features. There are obvious shortcomings, such as a package system and exceptions. Solutions to these problems are well-known, so these omissions only reflect Closet's status as a proof-of-concept language. However, a more critical error, the weakness of the type system, must be corrected before Closet is even a really effective proof of concept. While the type system works very well for simple types and function types, it does not support generics, which are crucial to enabling the programming style which is so popular in scripting languages. Before Closet can be considered a unilateral success, generics at the minimum must be added.

4.4 Future Work

There are several pieces still missing from Closet, as explained in the sections below.

4.4.1 Package System

The first, most significant improvement would be a package or module system which allows a program to be organized into a hierarchical set of files as opposed to a single file. The ability to include functionality from another body of code is of critical importance because external libraries cannot exist in any practical sense without it. Furthermore, Java showed that organizing code into a hierarchical set of separate namespaces is highly effective in organizing code and dealing with name collisions, in both the programmatic and the human, cognitive sense.

4.4.2 Generics

Generic typing is a critical next step. One of the key features of dynamic languages is their wholly polymorphic nature: any operation can be applied to any value in source and the runtime

system sorts out if these operations are valid. Polymorphism is also possible in Closet, although verbose casts are required to satisfy the type checker and Closet can make few static checks on such casts. The introduction of generics to the type inferencing system would allow Closet to have the same polymorphic behavior of dynamic languages while still maintaining static typing and therefore type safety.

Generics are possible under Closet's current type model. Closet's type inferencing system uses assignment statements to propagate type information from typed expressions to free variables; this general idea can be extended to implement generics. By propagating type information to generic type variables through the parameter types of function calls and through being the target of the return statement of a function, flexible and effective generics should be possible.

4.4.3 Overloading and Exceptions

Function overloading is important for the ease-of-use of the language. Exceptions are important for the same reason. These features are well-known and well-understood; adding these features in Closet should not pose a significant problem.

4.4.4 Object-Oriented Features

Various Object-Oriented programming constructs would be useful in Closet. One example of such a construct is a privilege system allowing the programmer to assign privilege levels (`public`, `protected`, `private`) to class members. A more developed notion of constructors, in particular the guaranteed calling of superclass constructors, is important for safety and object semantics. Abstract classes would also be very useful.

4.4.5 Multiple Inheritance

Controlled multiple inheritance has also been shown to be effective by languages such as Ruby and Java. Adding **interfaces**, or classes with only abstract methods as members, to Closet's typing system is possible, although it does somewhat complicate the `unify` function and raises the possibility of a typing conflict. Consider the following example:

```
class A:
end


interface I:
end


class B extends A implements I:
end
```

```
class C extends A implements I:
end


function void main():
  var a=new B()
  a = new C()
end
```

Code Listing 4.4.5.1: Example including interfaces

The type of `a` should be the most specific type which accepts values of type `B` and `C`, so what should the type of variable `a` in function `main` be: `A` or `I`? The type system could choose an arbitrary policy about resolving such conflicts, issue a fatal error, or try to determine which type is best using the remainder of the variable's scope. Regardless, Closet should print a notice to the user about the conflict.

Another possible addition to Closet's typing system would be **mixins**, or abstract classes which contain only methods, with or without implementations, but no data members.

4.5 Cosmetic Changes

Various other syntactic sugars should be added to Closet to make Closet look and act more like a dynamic language. One good example is a `Range` type with a syntactic literal like `0..10`. Additional looping constructs would be useful, in particular a `foreach` loop. A simple syntax for anonymous functions and classes would also be useful.

5.0 Conclusions

Closet has proved to be a good medium for exploring the marriage of static and scripting languages. The use of type inferencing combined with explicitly typed functions and member variables is an effective method for inferring types under a single-inheritance class-based type model. Furthermore, generics and multiple inheritance appear to be possible under the same approach to typing, although the feasibility is not known for certain. In light of the work done on this project, it is clear that there is potential for combining these two paradigms for programming languages.

6.0 References

1. Cardelli, Luca. *Type Systems*. In the Handbook of Computer Science and Engineering, Allen B. Tucker Jr. Ed., CRC Press 1997.
2. Java 1.4.2 API Documentation. `http://java.sun.com/j2se/1.4.2/docs/api/index.html`.
3. Pierce, Benjamin. *Types and Programming Languages*. MIT Press, London. 2002.
4. Edwards, Stephen. *Tiger Language Reference Manual*. Online. `http://www.cs.columbia.edu/~sedwards/classes/2002/w4115/tiger.pdf`.
5. Guy Steele. *Growing a language*. In Proc. of OOPSLA'98, 13th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications, October 1998. key note.

Appendix A: Closet BNF

```
prgm ::= prgm_stmt*
prgm_stmt ::= class_stmt
            | function_stmt
eval_stmt ::= expr ;
declare_stmt ::= var id (: type)? (= expr)? ;
return_stmt ::= return expr ;
function_stmt ::= function type id ( arg_list ) {
                    function_stmts* }
if_stmt ::= if expr : block
          | if expr : do function_stmt* else : block
while_stmt ::= while ( expr ) block
class_stmt ::= class id (extends simple_type)? :
                class_stmts* end
method_stmt ::= method type id ( arg_list ) :
                  function_stmts* end
mdeclare_stmt ::= var id : type ;
expr ::= expr + expr
       | expr - expr
       | expr * expr
       | expr / expr
       | expr % expr
       | expr === expr
       | expr !== expr
       | expr == expr
       | expr != expr
       | expr < expr
       | expr <= expr
       | expr > expr
       | expr >= expr
       | expr && expr
       | expr || expr
       | ! expr
       | expr isa stype
       | ( cast stype ) expr
       | value
value ::= id
        | self
        | value . id
        | value ( param_list )
        | ( expr )
        | int
        | string
        | bool
        | new stype ( param_list )
        | lambda ( arg_list ) -> expr
        | null
arg_list ::= id : type (, id : type)* | ϵ
param_list ::= expr (, expr)* | ϵ
type_list ::= type | (, type)* | ϵ
```

```
class_stmts ::= method_stmt
              | mdeclare_stmt
function_stmts ::= function_stmt
                 | class_stmt
                 | if_stmt
                 | while_stmt
                 | eval_stmt
                 | return_stmt
                 | declare_stmt
bool ::= true
       | false
type ::= stype
       | ctype
stype ::= id | void
ctype ::= ( type_list ) -> type
```

NOTES:

- Single-line comments begin with `#' and continue to the end of the line. There is no block comment.
- End-of-line can be used in place of a semicolon to terminate a statement

Appendix B: Closet Expressions

This appendix describes each of Closet's twenty expression types in detail. Each expression type has a listing in the following format:

ExpressionType
Syntax: *BNF*

Typing: **Type Rule**

Inference Step: *Any compile-time errors issued during the inference phase of compilation.*

Runtime Errors: *Any runtime errors which evaluating this expression may signal*

L-value: *Yes if the expression is a valid l-value; no otherwise*

Side-Effects: *Yes if the evaluating this expression can have side-effects; no otherwise*

Summary of the Expression including evaluation order for sub-expressions and any special scoping information.

AndExpr
Syntax: `expr && expr`

Typing: $$\frac{\Gamma \vdash t_1 : Boolean, t_2 : Boolean}{\Gamma \vdash (t_1 \&\& t_2) : Boolean}$$

Inference Step: If either of the expressions does not have the type `Boolean`, then an `IllegalBooleanOpError` is thrown.

Runtime Errors: If an evaluated expression has the value null, a `NullDereferenceError` is signaled.

L-value: No.

Side-Effects: If either subexpression has a side effects, then yes; otherwise, no

`AndExpr`s are evaluated using short-circuit logic, as they are in C and Java. If the value of the left expression is `false`, then the value of the entire expression is `false` and the right expression is never evaluated. Otherwise, the value of the entire expression is the value of the right expression.

AssignExpr

Syntax: `expr = expr`

Typing:
$$\frac{\Gamma \vdash x : T_1, t : T_2}{\Gamma \vdash x : unify(T_1, T_2), x = t : unify(T_1, T_2)}$$

Inference Step: If the expression on the left is not an l-value, an `IllegalAssignError` is issued. If the expression on the left is an l-value but is const, an `IllegalConstAssignError` is issued. If the left-hand expression is a locked variable and the right-hand expression has an improper type, then an `IllegalLockedAssignError` is issued. If the left-hand side expression is an undeclared variable, then an `UndeclaredWriteError` is issued. If this assignment causes the type of a free variable to unify to `Object` from a more specific type, an `AssignUnifiesToObjectWarning` is issued.

Runtime Errors: None.

L-value: No. (`AssignExpr`s are right-associative, though, so they may be stacked.)

Side-Effects: Yes.

The value of an assign expression is the value of the right-hand side of the expression. As a side-effect, the lvalue on the left-hand side is given the value of the right-hand side. The value on the right is evaluated before the assignment takes place.

Boolean Expression

Syntax: **`true`** | **`false`**

Typing:
$$true : Boolean$$
$$false : Boolean$$

Inference Step: None.

L-value: No.

Runtime Errors: None.

Side-Effects: No.

The value of a `Boolean` expression is a `Boolean` object with the value of the specified literal.

CallExpr

Syntax: `value ( param_list )`

Typing: $$\frac{\Gamma \vdash t_f : (T_1, T_2, \ldots, T_n) \rightarrow T_R, t_1 : T_1, t_2 : T_2, \ldots, t_n : T_n}{\Gamma \vdash t_f(t_1, t_2, \ldots, t_n) : T_R}$$

Inference Phase: If the static type of value is not a `CompoundType`, then an `IllegalCallError` is issued. If the wrong number of parameters are specified, then an `IllegalArityError` is issued. If a actual argument type does not match an expected argument type, then an `IllegalArgumentError` is issued. If value is `null`, then a `NullDerefenceError` is issued.

Runtime Errors: None.

L-value: No.

Side-Effects: Yes.

A `CallExpr` is a call to a value with a static type of some `CompoundType`. The arguments are evaluated from left to right.

CastExpr

Syntax: `( cast stype ) expr`

Typing: $$\frac{\Gamma \vdash t : T}{\Gamma \vdash ((castT')t) : T'}$$

Inference Step: If the static type of the given expression is not related to the casting type, then an `IllegalCastError` is issued. If the cast type is not known, an `UndeclaredTypeError` is issued.

Runtime Errors: If the dynamic type of the given expression is not compatible with the casting type, an `IllegalCastError` is issued.

L-value: No.

Side-Effects: No.

The value of the cast expression is the value of the expression on the right-hand side, but the type of the whole expression is the type specified in the cast.

DotExpr

Syntax: `value . id`

Typing: $$\frac{\Gamma \vdash t : T, T.\Gamma \vdash id : T'}{\Gamma \vdash t.id : T'}$$

Inference Phase: If the specified value does not exist within static type of the given object, then an `UndeclaredDotReadError` will be issued. If the value on the left has the null type, then a `TypelessDereferenceError` is thrown.

Runtime Errors: If the left-hand value is null, a `NullDereferenceError` is signaled.

L-value: If the given value is a `SelfExpr`, then yes; otherwise, no.

Side-Effects: No.

The value of a dot expression is the value of looking up `id in value`. Dot expressions with a `SelfExpr` as the left-hand side are lvalues, again only if the expression evaluates to a non-const variable. In the case of methods, the returned value is a bound method.

GetExpr

Syntax: `value [ expr ]`

Typing: $$\frac{\Gamma \vdash t_v : T_v, t_i : T_i, T_v.\Gamma \vdash \_\_get\_\_ : (T_i) \rightarrow T_R}{\Gamma \vdash t_v[t_i] : T_R}$$

Inference Step: Because of the semantics of the `GetExpr`, any errors which are issued when inferencing a `CallExpr` may be issued when inferencing a `GetExpr`.

Runtime Errors: If the specified index is out of range, an `IndexOutOfRange` error is signaled. What values are out of range is defined by the object implementing `__get__`.

L-value: Yes.

Side-Effects: If evaluating the specified value or the given expression has side-effects, then yes; otherwise, no.

The `GetExpr` is syntactic sugar for accessing the elements of an aggregate structure.

The expression `a[i]` is evaluated as `a.__get__(i)`, so any class can use this syntax by defining a `__get__` method. The interpreter does not check the signature of user-defined `__get__` methods, but only get methods that take exactly one argument will be useful for using

this special syntax. Any argument types or return type may be used.

The expression `a[i] = b` is evaluated as `a.__set__(i, b)`, so any class can use this syntax by defining the `__set__` method. The interpreter does not check the signature of user-defined `__set__` methods, but only set methods that take exactly two arguments will be useful for using this special syntax. Any argument types or return type may be used.

IntegerExpr

Syntax: `int`

Typing:

$$0 : Integer$$

$$\frac{\Gamma \vdash t : Integer}{\Gamma \vdash succ(t) : Integer}$$

Inference Step: None.

Runtime Errors: None.

L-value: No.

Side-Effects: No.

The value of an `IntegerExpr` is an `Integer` object with the value of the specified literal.

IsAExpr

Syntax: `expr isa type`

Typing: $tisaT : Boolean$

Inference Step: If the static type of the expression and the specified testing type are unrelated, then an `IllegalIsAError` is thrown. If the testing type is not known, and `UndeclaredTypeError` is thrown.

Runtime Errors: None.

L-value: No.

Side-Effects: No.

The value of the expression is `true` if the dynamic type of the value of the given expression is assignment-compatible with the specified type at runtime, or `false` otherwise.

IsSameExpr

Syntax: `expr === expr`

Typing: $t_1 === t_2 : Boolean$

Inference Step: If the specified expression are of incompatible types, then an `IllegalIdentityCheckError` is issued.

Runtime Errors: None.

L-value: No.

Side-Effects: No.

The value of the expression is `true` if the reference on the left is the same reference as the reference on the right, or `false` otherwise. The expression on the left is evaluated before the expression on the right.

LambdaExpr

Syntax: `lambda ( arg_list ) -> expr`

Typing: $$\frac{\Gamma, t_1 : T_1, t_2 : T_2, \ldots, t_n : T_n \vdash t_R : T_R}{\Gamma \vdash (lambda(t_1 : T_1, t_2 : T_2, \ldots, t_n : T_n) -> t_R) : (T_1, T_2, \ldots, T_n) \rightarrow T_R}$$

Inference Step: If an argument type is not declared, an `UndeclaredTypeError` is issued.

Runtime Errors: None.

L-value: No.

Side-Effects: No.

The value of a `LambdaExpr` is a function with a body of exactly one return statement which has the value of the given expression. The type of a lambda is a `CompoundType` with the domain part specified explicitly in the argument list and the range part induced by the type of the right-hand side expression.

ListExpr

Syntax: `[ expr_list ]`

Typing: $[t_1, t_2, \ldots, t_n] : List$

Inference Step: None.

Runtime Errors: None.

L-value: No.

Side-Effects: If any of the subexpressions has side effects, then yes; otherwise, no.

The value of the expression is a new `List` with the leftmost expression specified at index `0`, the next leftmost expression at index `1`, and so on. Expressions are evaluated from left to right.

NewExpr

Syntax: `new stype ( param_list )`

Typing: $$\frac{\Gamma \vdash (T.\Gamma \vdash \_\_init\_\_ : (T_1, T_2, \ldots, T_n) \rightarrow void), t_1 : T_1, t_2 : T_2, \ldots, t_n : T_n}{\Gamma \vdash newT(t_1, t_2, \ldots, t_n) : T}$$

Inference Step: If the specified type has no `__init__` method defined, an `IllegalDotRead` error is issued. If the wrong number of arguments are specified to the constructor, then an `IllegalArityError` is issued. If a specified and expected argument type do not match, an `IllegalArgumentError` is issued.

Runtime Errors: None.

L-value: No.

Side-Effects: Yes.

The value of the expression is a reference to a new instance of an object of the given type. The parameters are passed to the object's `__init__` method, which is called before the reference is returned.

NotExpr

Syntax: `! expr`

Typing: $$\frac{\Gamma \vdash t : Boolean}{\Gamma \vdash !t : Boolean}$$

Inference Step: If the specified expression does not have type `Boolean`, then an `IllegalBooleanOpError` error is issued.

Runtime Errors: None.

L-value: No.

Side-effects: If the specified expression has side effects, then yes; otherwise, no.

If the value of the given expression is `true`, the value of the whole expression is `false`, or `true` otherwise.

NullExpr

Syntax: **`null`**

Typing: $null : NullType$

Inference Step: None.

Runtime Errors: None.

L-value: No.

Side-Effects: No.

The value of the given expression is the special `null` reference.

OrExpr

Syntax: `expr || expr`

Typing: $$\frac{\Gamma \vdash t_1 : Boolean, t_2 : Boolean}{\Gamma \vdash (t_1 \| t_2) : Boolean}$$

Inference Step: If either of the expressions does not have the type `Boolean`, then an `IllegalBooleanOpError` is issued. If one of the evaluated expressions is null, a `NullDereferenceError` is thrown.

Runtime Errors: None.

L-value: No.

Side-Effects: If either expression has a side effects, yes; otherwise, no

`OrExpr`s are also evaluated using short-circuit logic. If the value of the left expression is `true`, then the value of the entire expression is `true` and the right expression is never evaluated. Otherwise, the value of the entire expression is the value of the right expression.

SelfExpr

Syntax: **`self`**

Typing: $$\frac{self : T \in \Gamma}{\Gamma \vdash self : T}$$

Inference Phase: If the expression is not in the scope of a method, an `IllegalSelfError` will be thrown.

Runtime Errors: None.

L-value: Yes.

Side-Effects: No.

In a method, the value of a `SelfExpr` is the object on which a method was invoked. `SelfExprs` are only legal within the scope of a method body.

SliceExpr

Syntax: `value [ expr? : expr? ]`

Typing:
$$\frac{\Gamma \vdash t_v : T_v, t_i : T_i, t_j : T_j, T_v.\Gamma \vdash \_slice\_\_ : (T_i, T_j) \rightarrow T_R}{\Gamma \vdash t_v[t_i : T_j] : T_R}$$

Inference Phase: Due to the implementation of the `SliceExpr`, any errors which may be issued by inferencing a `CallExpr` may be issued by inferencing a `SliceExpr`.

Runtime Errors: An `IndexOutOfRangeError` may be thrown.

L-value: No.

Side-Effects: If the specifed value or one of the subexpressions has side-effects, then yes; otherwise, no.

A `SliceExpr` is used as syntactic sugar for retrieving multiple elements from an aggregate at once. There is no hard specification for how a `SliceExpr` should behave, but the convention is that the return type should be itself an aggregate type and that the two expressions should have the key type for the aggregate type and that the result of `a[i:j]` should contain all the values corresponding to the keys in the open interval `[i, j)`. If `i<=j`, then the result should contain the value corresponding to `i` first moving to `j` last, or vice versa if `j<i`. So, `[0,1,2,3][0:2] => [0,1]` and `[0,1,2,3][2:0] = [2, 1]`.

Super Expression

Syntax: **`super`**

Typing:
$$\frac{self : T \in \Gamma}{\Gamma \vdash super : parent(T)}$$

Inference Phase: If the `SuperExpr` appears in a class without a parent, an

`IllegalSelfError` is issued. If the super expression appears on its own, a `StandaloneSuperError` is issued.

Runtime Errors: None.

L-value: No.

Side-Effects: No.

In a method, the value of a `SuperExpr` is the object on which the method was invoked. `SuperExprs` are only valid within the scope of a method. In `DotExprs` with a `SuperExpr` as the left-hand side, value lookup starts with the parent of the static type of self, which distinguishes it from all other expressions, which starts lookup dynamic type of the value.

SymExpr

Syntax: `id`

Typing: $$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

Inference Phase: If the specified variable has not been declared, an `UndeclaredReadException` will be thrown. If the specified variable has not been initialized, an `UninitializedReadError` will be thrown.

Runtime Errors: None.

L-value: Yes.

Side-Effects: No.

The value of a `SymExpr` is the value of a given variable at the time of evaluation. Syntactically, `SymExprs` are valid lvalues, although semantically only `SymExprs` evaluating to non-const variables are l-values.

StringExpr

Syntax: `string`

Typing: $\text{“}\ldots\text{”} : String$

Inference Step: None.

Runtime Errors: None.

L-value: No.

Side-Effects: No.

The value of a string expression is a `String` object with the value of the specified literal.

Appendix C: Closet Statements

This appendix describes each of Closet's ten statement types in detail. Each statement type has a listing in the following format:

StatementType
Syntax: *BNF*

Inference Phase: *Any compile-time errors issued during the inference phase of compilation*

Analysis Phase: *Any compile-time errors issued during the analysis phase of compilation*

Runtime Errors: *Any runtime errors which executing this statement may signal*

Summary of the Statement's effects and uses.

ClassStmt
Syntax: `class id (extends simple_type)? : class_stmts* end`

Inference Phase: If a non-`void __init__` method is specified, then a `NonVoidConstructorError` is issued. If an overriding method has a signature which cannot override its parent method, then a `MismatchedOverrideError` is issued. If the parent type is not declared, then an `UndeclaredTypeError` is issued. If the return type or an argument type in a method is not known, an `UndeclaredTypeError` is issued. If the type of a member variable is not known, an `UndeclaredTypeError` is issued.

Analysis Phase: None.

Runtime: None.

A `ClassStmt` is the syntactic form for defining a new class type. Methods are closures, so class methods capture the variables in the scope where the class is defined.

DeclareStmt
Syntax: `var id (: type)? (= expr)?`

Inference Phase: If an initialization value was specified which does not match the type specified for a locked variable, an `IllegalLockedInitializationError` will be thrown. If a variable with this name was declared in this scope, not an enclosing scope, then a|

`RedefinitionError` is thrown. If the type is specified but not declared, an `UndeclaredTypeError` is thrown.

Analysis Phase: None.

Runtime: None.

A `DeclareStmt` is the syntactic form for introducing a new variable into the current scope. A type may be specified explicitly to indicate a locked variable, and an initial value may be specified as well.

EvalStmt

Syntax:
```
expr ;
```

Inference Phase: None.

Analysis Phase: The interpreter will generate a `StmtHasNoSideEffect` warning if the evaluation of the expression can have no side-effects.

Runtime: None.

An `EvalStmt` evaluates the given expression. `EvalStmt`s should have a side-effect.

FunctionStmt

Syntax:
```
function type id ( arg_list ) { function_stmts* }
```

Inference Phase: If the return type or an argument type is not known, an `UndeclaredTypeError` is thrown.

Analysis Phase: In a non-void function, the interpreter will generate a `NotAllControlPathsReturnError` if not all return paths return a value.

Runtime: None.

A `FunctionStmt` is used to introduce a new named function into the current scope. Functions in Closet are closures, so function statements may be nested.

IfElseStmt

Syntax:
```
if expr : block |
if expr : do function_stmt* else : block
```

Inference Phase: If the condition does not have type Boolean, an `IllegalConditionError` will be issued.

Analysis Phase: None.

Runtime: If the condition is `null`, a `NullDereferenceError` is signaled.

An `IfElseStmt` is one of the syntactic forms for making a decision based on a `Boolean` truth value. If the condition is `true`, the first block is executed; if the condition is `false`, the second block (or nothing, if no second block is given) is executed.

ReturnStmt

Syntax: `return expr? ;`

Inference Phase: If the return value of this return statement does not match the return type of its enclosing function or method, an `IllegalReturnError` will be issued.

Analysis Phase: None.

Runtime: None.

A `ReturnStmt` is the syntactic form for returning a value from a function.

WhileStmt

Syntax: `while expr : block`

Inference Phase: If the condition does not have type `Boolean`, an `IllegalConditionError` will be thrown.

Analysis Phase: None.

Runtime: If the condition is `null`, a `NullDereferenceError` is signaled.

A `WhileStmt` is another construct for choosing a control path based on a `Boolean` expression and is the only syntactic form for iteration. The condition is evaluated. If the condition is `true`, then the block is executed and the condition is evaluated again in preparation for possibly running the block again; if the condition is `false`, then the block is skipped. Therefore, the expression is evaluated at least once and the block is executed zero or more times.