Evaluating the Expressiveness of a Multimethod Object-Oriented Programming Language

Sevtap Otles Karakoy
TR #98-12
July 1998

Keywords: Multimethods, generic functions, object-oriented programming languages, encapsulation, information hiding, block structure, subtyping, inheritance, TSTBC language.

1994 CR Categories: D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics; D.3.2 [*Programming Languages*] Language Classifications — object-oriented languages; D.3.3 [*Programming Languages*] Language Constructs and Features — abstract data types, control structures, procedures, functions, and subroutines; D.3.m [*Programming Languages*] Miscellaneous — multimethods, generic functions; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — operational semantics; F.3.3 [*Logics and Meanings of Programs*] Studies of Program Constructs — control primitives.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Evaluating the Expressiveness of a Multimethod Object-Oriented Programming Language

Sevtap Otles Karakoy
July 16, 1998
Under the supervision of Professor GaryT. Leavens

Abstract

Although most popular object oriented languages use single dispatch, multiple dispatch programming languages offer more expressiveness over single dispatch languages. We developed an interpreter for a multimethod object-oriented language, TSTBC, to study the expressiveness of these languages.

Compared with other multimethod languages, TSTBC features: objects that can act as classes, instances, and first-class generic functions, as well as method update. TSTBC has scoped inheritance, and the inheritance relation of an object is set at the time of its creation. The generic functions can be extended in the scope they are declared in, but only way to extend a generic function in a nested scope is to create a new generic function, assign the old generic function's methods to the new generic function and extend the new generic function.

1 Introduction

Object-oriented programming languages offer many useful features for designing computer programs. However not all object-oriented languages support exactly the same approach to software design and development.

Object-oriented languages can be separated into two main groups [Castagna 96]: Single dispatch languages and multiple dispatch languages. In single dispatch languages methods belong to a unique class, and messages are always sent to a single object. Many widely used languages such as Smalltalk [Goldberg & Robson 83], C++ [Stroustrup 91] and Java [Arnold & Gosling 98] are single dispatch languages. In multiple dispatch languages, such as CLOS [Steele 90, Paepcke 93], Dylan [Shalit 97, Feinberg 97], and Cecil [Chambers 92, Chambers 95], methods do not belong to a single class, rather they are global, and messages can be sent to more that one object at once. Depending on the language, methods can specialize on all or subset of their arguments. With multiple dispatching, method selection is based on the runtime classes of several arguments, as opposed to a runtime class of a single receiver in single dispatch languages [Chambers & Leavens 95].

Since the multiple dispatch mechanism is a more generalized version of the single dispatch mechanism, and single dispatch languages are not able to address the binary-method problem [Bruce *et. al.* 93], we have decided to study the semantics of multiple dispatch to investigate its expressiveness. The multiple dispatch mechanism is implemented by using generic functions. A *generic function* groups a set of methods that might have different number of arguments and different specializers, which are the types of its formals. When a generic function is called, it decides which method to execute dynamically. Generic functions will be explained in more detail in later sections.

Another feature that is studied in this work is scoped inheritance. Instead of maintaining a global inheritance graph that keeps all inheritance relations in the program, an inheritance relation is maintained within given scope, just like one might do with variables in a block-structured language. Maintaining a scoped inheritance relation also provides better encapsulation by making only part of the whole inheritance graph visible to clients. The inheritance relation in a given scope includes all the information from that scope plus the surrounding scopes. At the same time this inheritance relation is enclosed in every object to assure that an object that is accessed outside of its immediate environment still remembers its inheritance relationship, in the same way that it remembers bindings for variables.

Chambers and Leavens have developed a block structured, multimethod object-oriented language, BeCecil, to study the type system and semantics of such languages [Chambers & Leavens 96]. BeCecil has a prototype-based object model with first-class generic functions. It also has a static type system that separates subtyping from inheritance. However, this type system had many problems, some of which are caused by the complications of separating subtyping from code inheritance.

The purpose of this study is to investigate how the expressiveness of BeCecil is affected if we keep subtyping and inheritance parallel. TSTBC (To Simple To BeCecil) is a language that takes BeCecil as a model and simplifies it by collapsing inheritance and subtyping together while maintaining the prototype-based object model of BeCecil.

To accomplish this, we have developed an interpreter for TSTBC. The interpreter has allowed us to test TSTBC programs, and to see how expressive the language is. The features of the interpreter and the implementation is explained in detail in section 3.

Since BeCecil has been taken as a model for TSTBC, we list the main similarities and differences between these two languages below.

Both TSTBC and BeCecil support the following programming ideas:

- **Prototype-Based:** not distinguishing between the inheritance and "instance-of" relationships makes the language simpler. The inheritance relation is defined between objects. To create instances of an object, one simply creates an object that inherits from it.
- **Multimethods:** methods are collected in objects that are called generic function. Each method in the generic function may specialize on different objects. Generic functions are first-class objects.
- **Multiple Inheritance:** objects can have more than one parent.
- **Scoping and Encapsulation:** Visibility of all declarations are limited to a particular section of code. This allows programmers to be able to encapsulate some object that might otherwise clutter the interface.

Below are some points where TSTBC differs from BeCecil:

- **Type system:** Although it has problems, BeCecil has a static type system that separates code inheritance from subtyping. In TSTBC subtyping is tied to the code inheritance. At this point in our study, the static type system for TSTBC has not been implemented. Although the TSTBC method selection mechanism uses inheritance relations, and therefore subtyping, to decide method applicability, at present it only has a dynamic type system.

- **Method update versus storage tables:** BeCecil is using a mechanism called storage tables to implement instance variables. A storage table is a relation that associates keys (tuple of object identities) to a value (single object) [Chambers & Leavens 96]. TSTBC on the other hand uses method update to implement instance variables. Using method update instead of storage tables gives TSTBC more uniformity. Everything is treated as a generic function. At the same time, with method update, we are able to give block of code to be executed instead of an expression. Both storage tables and method replacement will be explained in detail in later sections.
- **Extending Objects:** In BeCecil superclasses (and supertypes) can be added to existing objects in nested scopes. In TSTBC only generic functions can be extended in a nested scope. Even this is accomplished by creating a new generic function object, and extending it with the old object that is desired to be extended. Once this is done, the new generic function object can be extended with new methods. In TSTBC one can inherit from an object that is defined in a surrounding scope, but the parents of an object are set at the time the object is created. This limitation is made because we had static typing in mind. If the inheritance relation of objects are allowed to change, it is impossible to make sure statically that generic functions have the unique most-specific method to apply. The concept of a unique most-specific method is explained in later sections.

The rest of this paper is organized as follows. In section 2, we explain the basic features of TSTBC, and give some examples to show how to program using TSTBC. In section 3, we talk about the dynamic semantics and important semantic issues that was raised during this study. Section 4 will compare TSTBC to BeCecil in detail, discuss the future work and conclusions.

2 Basic Features of TSTBC

In this section we introduce and explain TSTBC. The core concrete syntax and some examples are presented to show how object-oriented features might be programmed. Since the core language is very small, it is not intended for direct use. Some syntactic sugars will be presented in later part of this section that provide ease in programming.

The basic features of TSTBC are very closely related to the those of BeCecil, therefore some of the examples and explanations are directly taken from the 1996 paper of Chambers and Leavens [Chambers & Leavens 1996].

2.1 Core Syntax

The abstract syntax of the core version of TSTBC appears in Figure 2-1 below. TSTBC has a call-by-value parameter passing mechanism with the indirect model of objects [Friedman et al. 92]. Therefore, all the actual parameters are evaluated at the time of the call, and their values that are object identities are passed.

2.2 Objects, Instances and Inheritance

An object in TSTBC can be created by an object declaration, which creates an object and binds it to the name given. An object in TSTBC can assume several roles.

3. It can inherit code from other objects, this will allow it to behave as an instance of the other prototypes.
3. Other objects can inherit from it, in which case the same object is acting as a prototype to these other objects.
3. Finally, it can act as a generic function by grouping methods together.

In another words, objects can be instances, classes, and generic functions at the same time. Each object has a unique identity that is bound to the object's name when the object is created. For example, the following declaration creates an object and binds `Point` to its object identifier. It also specifies that the `Point` objects inherits from a top level object called "`any`"

```
object Point inherits any
```

Object names, such as `Point`, are constant references to objects. While the state of the object may change, the name always denotes the same object.

In addition to its identity, each object has an inheritance relation and a location that can be used to store a set of methods. The inheritance relation of an object is fixed when the object is created. If we create the object using the object declaration given above, then the object only inherits from a top level object called "`any`" that all objects inherit from. On the other hand an object may be declared to inherit from other more specific objects, as in the following.

```
object Line inherits Point
```

Since TSTBC supports multiple inheritance, we could have a list of object identifiers on the right hand side of the `inherits` keyword separated by commas. The following example declares an object `ColorPoint` that inherits from `Point` and an object named `Color`.

P	::=		Program
		B	execute the given block
B	::=		Block
		RDS E	the recursive declaration sequence provides context for E
RDS	::=		Recursive-Declaration-Sequence
		D*	mutual recursion is allowed within D*
D	::=		Declaration
		object I **inherits** CN*	object: allocates a new object named I, I inherits from CN*
	|	**has** CN GF	extension: CN to include the methods of GF
	|	**hide** RDS **in** D* **end**	hide: the declarations of RDS are only visible to the D*
CN	::=		Class-Name
		I	identifier denoting a known object
M	::=		Method
		method(F***)** **{**B **}**	method: like λ, a block abstraction
	|	**method(**F***)**:T **{**B **}**	method: like λ, a block abstraction with return type
GF	::=		Generic-Function attributes
		M	method
	|	GF_1 **&** GF_2	combination: GF_2 favored over GF_1 if specializers clash
	|	I	identifier
F	::=		Formal argument
		I **:** T	the specializer for argument I is T (type-expression)
E	::=		Expression
		N	Numeral
	|	I	indexed identifier: evaluates to the object denoted by I
	|	E_0 **(**AA***)**	application: evaluates operator then operands left-to-right
	|	**list [**E***]**	list: list of Expressions
	|	**update** E(E*) :=**{**B**}**	update: method update
	|	**{**B**}**	block: evaluates the block B and returns its value
	|	E_1 **;** E_2	sequence: evaluates E_1 then E_2, returns the value of E_2
T	::=		Type expression
		TN	type name: a user-defined type
	|	**(**T***)** **->** T_{n+1}	invocable: with arguments T*, returns T_{n+1}
	|	T_1 **&** T_2	conjunction: glb of types
	|	T_1 **|** T_2	disjunction: lub of types
TN	::=		Type-Name
		CN	Class-Name
AA	::=		Actual-Argument
		E	undirected: applicable if inherits from specializer
	|	E **:**CNS	directed: appl. if also all CNS inherits from specializer
CNS	::=		List of Class-Names
		CN	class-name
	|	CN **&** CNS	clacc-name followed by list of class-names

Figure 2-1: Core syntax of TSTBC. As usual, an asterisk (*) denotes zero or more repetitions of the preceding nonterminal.

```
object Color inherits any
object ColorPoint inherits Point, Color
```

In TSTBC subtyping and inheritance are parallel. That is, there is no distinction between inheritance and subtyping.

Following BeCecil's example, TSTBC does not distinguish the inheritance and "instance-of" relationships. In this respect TSTBC is a prototype-based language. Since an instance of a class is just an object that inherits from the class, instances of classes are created just like subclasses. For example, one could declare an instance, `MyPoint`, of `Point` as follows.

```
object MyPoint inherits Point
```

As mentioned above, all the objects in TSTBC inherits from a top level object called `any`. Instead of writing "**inherits** `any`" with each such declaration we created a syntactic sugar, so that a declaration of the form,

```
object something
```

is translated into the following.

```
object something inherits any
```

Other syntactic sugars are explained in section 2.11.

2.3 Generic Functions and Methods

As mentioned earlier, objects in TSTBC can also act as generic functions. A generic function in TSTBC is a collection of multimethods. Each method in a generic function can have a different number of arguments and may also specialize on different classes. To declare a generic function, one would need to create an object to act as a generic function first. As an example, the generic function `Equal` could be created by the following object declaration.

```
object Equal
```

Assuming that the generic functions `x` and `y` are implemented for `Point`, we could then use the **has** declaration to extend the Equal object with a method. Below, we have a method that specializes on two `Point` objects, and extends the generic function `Equal`.

```
has Equal method(p1:Point, p2:Point){
   and(Equal(x(p1), x(p2)), Equal(y(p1), y(p2)))
}
```

A **has** declaration is similar to a `method` declaration in Cecil, and to `defmethod` in CLOS. Each **has** declaration is followed by a generic-function attribute. A generic function attribute can be a method declaration as in the above example, an identifier which denotes another generic function, or a combinations of generic function attributes, which is written in the form *GF1* **&** *GF2*. Here *GF1* is first generic function, *GF2* is the second generic function. If the specializers of the methods in *GF1* and *GF2* clash, the methods in *GF2* are preferred over those of *GF1*. The two methods clash if both have exactly the same tuple of specializers. In Figure 2-2, the first method of `eq1` and the first method of `eq2` are clashing. Thus the `eq3` generic function holds second method of `eq1` and both methods of `eq2`.

```
object eq1
has eq1 method (i:int, j:int){i}
has eq1 method (b:bool, c:bool) {eq(b,c)}

object eq2
has eq2 method (i:int, j:int){eq(i,j)}
has eq2 method (i:int, j:int, k:int){and(eq(i,j), eq(j,k))}
object eq3
has eq3 eq1 & eq2
```

Figure 2-2: Combination of generic functions

Each method declaration has the form of a keyword **`method`**, a list of formals, an optional type attribute which specifies the return type of the method, and finally a block of code which is enclosed in between the curly braces. The formals of the method have the form $I : T$, where I is the name of the formal, and T is its type or specializer. The list of types for each formal gives a list of specializers for that method. Since method dispatch depends on all the arguments, one needs to know the specializer of each argument. This is one of the main differences from a single dispatch language in which each method would have a single distinguished "receiver" object.

Method bodies in general contain a block, which consists of a recursive declaration sequence and an expression, which is evaluated and its value returned as the result of the method. In the case of the above `Equal` method, the block that forms the body has no declarations, but only an expression, which is evaluated when the method is called, and its value returned.

The expression in the body of the above method calls the generic function `Equal` twice, but each time with integer arguments (the two `x` and `y` coordinates of the points); thus these calls to `Equal` invoke other methods of the same generic function. Dispatch to methods is dynamic, and based on the ancestry of all the actual argument objects and the subtype relation closed with the generic function. A method is *applicable* to a tuple of actuals if each actual inherits from the corresponding specializer, according to the reflexive transitive closure of the inheritance relation that is obtained by the union of the inheritance relations enclosed in the generic function and the actual argument. When a generic function is invoked, the generic function must have a unique, most-specific method that is applicable to the actuals; this method is then executed with actual parameters. For example, the `Equal` method is applicable to two arguments that both inherit from `Point`, because the specializers of the method are both `Point`.

```
object mkPoint
has mkPoint method(x:int, y:int){
  object NewPoint inherits Point
  initialize(NewPoint, x, y)
  NewPoint}

x(mkPoint(1,2))
```

Assuming that `x` is defined for `Point` object, above code demonstrates that, inheritance relation that is inside the actual argument is needed to find an applicable method. The generic function `x` works on `Point` object, but does not have the information that `NewPoint` which is returned by `mkPoint` inherits from `Point`, since `x` is creates in the surrounding scope. The `NewPoint` carries its inheritance relation with it, so that applicability can be determined.

Deciding applicability of a method is done in several steps:

1. Find all the methods that are stored with the generic function being applied.
2. Take all methods from the above list that has the same number of arguments as the actuals.
3. Take all methods from that list such that each actual inherits from the corresponding formal. All of these methods are applicable to the given actuals.
4. From the list of applicable methods, find the unique, most-specific method. This is the method such that all its specializers inherit from all the other applicable methods' corresponding specializers.

At step 3, we state that, each actual inherits from the corresponding formal. In the case of directed actuals, which have the form *X:T*, both the actual itself, and all the specializer that follows it, must inherit from the formal. Directed actuals can thus be used to call methods of a super class (parent object). Using the previously defined `Point` and `ColorPoint` objects, consider the following.

```
CP inherits ColorPoint
x(CP:Point)
```

If the generic function `x` had methods for both `Point` and `ColorPoint`, then the above example would execute the one that specializes on `Point` rather than `ColorPoint`.

If at the end of the third step, the list of methods is empty, then a *"method not found"* error is given. On the other hand if the list of methods has more than one method at the end of the fourth step, a "*method ambiguous*" error is returned. The example in Figure 2-3 would produce an error "*method ambiguous*", since the two `init` methods are not more specific than one another

```
object init
has init method (p:Point, i:any, j:int) {
   update x(p):= {i};
   update y(p):= {j};
p }
has init method (p:Point, i:int, j:any) {
   update x(p):= {i};
   update y(p):= {j};
p }
object myPoint inherits Point
x(init(myPoint, 1,2))
```

Figure 2-3: An example that will generate "method ambiguous" error.

These errors are considered *type errors* in TSTBC [Chambers & Leavens 95]. Since there is no static type system implemented for TSTBC, these errors are caught dynamically. Designing a type system to prevent such type errors statically in TSTBC is future work.

2.4 Programming Instance Variables

Instance variables of an object are also handled as methods that specialize on that object:

```
object x
has x method(p:Point){0}
object y
has y method(p:Point){0}
```

The above example show how to program the variables`x` and `y` of the `Point` object. In this case whenever `x` and `y` are called with a `Point` argument, they will execute the code that is placed as the body of the method. Therefore they both will return 0 when called. Since TSTBC is a prototype-based language, any object that inherits from `Point` will also inherit the values for `x` and `y`. In the case of the `MyPoint` example above, if `x` is dispatched with `MyPoint` as its argument, it will return the value 0. In general, however it is desirable to change the value of instance variables. In TSTBC this is accomplished by using an **`update`** expression. An **`update`** expression has the form **`update`** $E_0(E^*)$**:=** **{**B**}**, E_0 is an expression that evaluates to an object which represents a generic function. E^* is list of expressions that each evaluate to an expressible value.The body of the **`update`** expression is a block that is enclosed in curly brackets. The block is formed into a closure and saved within the generic function. This block is evaluated when the generic function is dispatched with actual arguments that matches the list of expression E^* exactly. The example below shows how the **`update`** expression is used. After executing following update expressions, `x` generic function that is dispatched using `APoint` object as its argument, returns 1.

```
object APoint inherits Point
update x(APoint) :={1}
update y(APoint) :={1}
x(APoint)
```

An update expression finds the unique most-specific method that is applicable to E^* and updates that method to execute the body of the update expression whenever the generic function is dispatched using the exact arguments in E^*. Any other object that inherits from the objects in the expression list are not affected by the update expression. They will still execute using the default method. In the following example, `BPoint` inherits from `APoint`, but it is not affected from the update of `APoint`, when `x` is applied to the `BPoint`, it still returns the default value 0.

```
object BPoint inherits APoint
x(BPoint)
```

2.5 Programming Variables

Variables are programmed the same way that instance variables are programmed in TSTBC, except that they do not specialize on any object. Their method definition uses an empty list of formals. The following example shows how one can define a variable and change its value. First one needs to define an object to act as a variable, and extend it into a generic function using a **`has`** declaration which is followed by a **`method`** definition with zero formals. This **`method`** definition allows one to initialize the variable. The initial value of the variable can be changed anytime by method update. The following block of code will return the value 561.

```
object newVar
has newVar method(){0}
update newVar():= {555};
plus(newVar(),6)
```

2.6 Dynamic Creation of Objects

Since a method body is designed to be a block (recursive declaration sequence, plus expressions), object creation and initialization may be encapsulated in generic functions. This allows the programmer to create new objects dynamically. In TSTBC, it is often convenient to program both an initializer (these are called constructors in C++) and a separate primitive constructor to create and initialize the new object. The following is an example.

```
object initialize
has initialize method(p:Point,i:int,j:int):Point{
   update x(p) := {i};
   update y(p) := {j};
   p
}
object mkPoint
has mkPoint method(x:int, y:int) {
   object res inherits Point
   initialize(res, x, y)
}
```

Figure 2-4: Dynamic Creation of Objects

As noted in Figure 2-4, the body of a method contains a block which is a recursive declaration sequence and an expression. Since the object `res` is created inside this nested recursive declaration sequence, every time the method is executed a new object will be created, initialized and returned to the caller. Thus block structure allows one to create objects dynamically, even though objects are only created by object declarations.

2.7 Information Hiding: hide declarations

The information hiding mechanism in TSTBC is the **`hide`** declaration. A **`hide`** declaration consists of a recursive declaration sequence and a list of declarations. This provides means for encapsulation because the declarations in the recursive declaration sequence of the hide declaration are only visible in the declaration sequence that follows the keyword **`in`**.

As an example of using a **`hide`** declaration, consider code in Figure 2-5 that implements an `interval` object. In this example, it is assumed that functions `min` and `max` are already defined for integer objects. An `interval` object is represented by its `lower` and `upper` boundaries. Since the bounds are defined inside the hide declaration. they can not be accessed anywhere in the program except the declaration sequence between the keywords **`in`** and **`end`**. The generic functions `lower` and `upper` are only visible to the generic functions that are in the public part of the hide declaration, and to each other. Any other generic function can only access them through the public functions. This is similar to the class declarations in C++; declarations that are in between the **hide** and **in** keywords represent the private part of a class, and declarations between the keywords **in** and **end** represent the public part of a class.

```
object interval
hide
   object lower
   has lower method(i:interval){0}
   object upper
   has upper method(i:interval){0}
in
   object initialize
   has initialize method (i:interval, lowb:int, upb:int) {
         update lower(i) := {min(lowb, upb)};
         update upper(i) := {max(lowb, upb)};
         i}
   object assign
   has assign method(l:interval, r:interval){
         update lower(l) := {lower(r)};
         update upper(l) := {upper(r)};
         l}
   object getLower
   has getLower method(i:interval){lower(i)}
   object getUpper
   has getUpper method(i:interval){upper(i)}
end
object oldinterval inherits interval
object newinterval inherits interval
getLower(initialize(oldinterval, 1,5));
getUpper(assign(newinterval, oldinterval))
```

Figure 2-5: Usage of the hide declaration in TSTBC

As it can be seen from the above example, new intervals can be created using the public function `initialize`, one interval can be copied to the another by using the `assign` public function, and any client code can get the boundary values for a specific interval object by using the functions `getLower` and `getUpper`, but they do not have the ability to do anything else. This gives programmers the ability to specify an interface to their implementation, and hide the implementation details inside the **`hide`** declaration.

2.8 Multimethods and Inheritance of Methods

As mentioned in introduction, TSTBC uses multiple-dispatch instead of single-dispatch, therefore programming binary methods [Bruce *et al.* 95] is not a problem as it is in single dispatch languages. `The Equal` method is an example of a binary method. In the example given earlier, the `Equal` method is defined for `Point` object by using another `Equal` method that is defined for integer ("`int`") objects. The example in Figure 2-6 also extends the `Equal` generic function with a method that is defined on `ColorPoint` objects.

As it can be seen from the example in Figure 2-6, the interaction of multimethods and inheritance is powerful, and avoids the need to write an exponential number of methods to deal with all the cases of binary methods [Chambers 92, Castagna 95].

To invoke an overridden method inherited from a superclass in TSTBC, one can use the directed form of actual arguments. In this form, the actual arguments are written as an expression followed by a colon (:), and a list of class names that are separated by an ampersand (&). When directed actuals are used, the method selected must be such that both the object that is the value of the expression and all of the named classes inherit from the corresponding formal parameter's specializer. In the above example directed actuals are used in the definition of the `initialize` method that is specialized on two `ColorPoint` objects. This method dispatches two other `initialize` methods, one to initialize `x` and `y` instance variables of the `Point` object and the other to initialize the color value of a `ColorScale object,` therefore it uses the directed form of actuals to call the appropriate `initialize` method for each one. The `Equal` method also uses the directed form of actuals to dispatch a method that belongs to a super class. Directed actuals are similar to the use of `super` in Smalltalk or `Point::` and `ColorScale::` in C++.

2.9 Programming First-Class Procedures

Because generic functions are objects, they can be used as first-class procedures. The example presented in Figure 2-7 demonstrates how a generic function can be used as a first class procedure. The square brackets ([]) represents a parameterless procedure. It is a syntactic sugar and explained in Section 2.11. The object `void` is defined in the standard prelude and it would return itself, as its value when called with no arguments. The standard prelude is explained in section 3.

In Figure 2-7, first, the generic function `ifTrue` is defined. It takes a boolean (a pre-defined `bool` object), and a generic function that does not have any arguments. `IfTrue` calls the generic function that is passed to it, only if the value of boolean is true, otherwise it returns a `void` object. The generic function `while` which acts as a while-loop, is defined by using `ifTrue`. The method of `while` generic function accepts two generic function as its arguments. Both of these functions have zero arguments themselves. The first generic function argument is a condition, and the second one is a statement that is executed when the condition is true. The rest of the above example just demonstrates how this while-loop might be used. After executing `while(equalC2, updateC)`, object `c` will have a value of 4.

2.10 Type Expressions

In TSTBC, the type expressions in Figure 2-1 are defined inductively. User-defined type names which are essentially object names, are the basis for this induction. The second kind of type expression describes the generic function object, the invocable type, which consists of a list of argument types (T^*), an arrow (**->**), and a return type (T_{n+1}). The other two types are the conjunction and disjunction of types, and they are specified as T_1 **&** T_2, and T_1 **|** T_2 respectively. The conjunction is the least specific type that subtypes both *T1* and *T2*. The disjunction is the most specific type that is a supertype of both *T1* and *T2*.

```
oject initialize
has initialize method (p:Point, i:int, j:int):Point {
  update x(p):= {i};
  update y(p):= {j};
  p }
object ColorScale
object getColor
hide
  object setColor
  has setColor method( c:ColorPoint) {0}
in
  has initialize method(cp:ColorScale, color:int) {
    update setColor(cp):= {color};
    cp
    }
  has getColor method (cp:ColorScale) {
    setColor(cp)
    }
end
object ColorPoint inherits Point, ColorScale
has initialize method (cp:ColorPoint, i:int, j:int, color:int)
{
  initialize(cp:ColorScale, color);
  initialize(cp:Point, i, j);
  cp
  }
object mkColorPoint
has mkColorPoint method(color:int, i:int, j:int){
  object retCP inherits ColorPoint
  initialize(retCP, i, j, color)
  }
object mkPoint
has mkPoint method(i:int, j:int){
  object retP inherits Point
  initialize(retP, i, j)
  }
has Equal method(cp1:ColorPoint, cp2:ColorPoint) {
  and( Equal(getColor(cp1), getColor(cp2)),
       Equal(cp1:Point, cp2:Point))
  }
Equal(mkColorPoint(1,1,2),mkPoint(1,2)),
Equal(mkColorPoint(1,1,2),mkColorPoint(1,1,2))
```

Figure 2-6: Interaction of multimethods and inheritance

```
object ifTrue
has ifTrue method(b:bool, a:()->any){if(b, [a()], void)}

object while
has while method(c:()->boolean, s:()->any){
   object loop
   has loop method(){s(); while(c, s)}
   ifTrue(c(), loop)
}
object c
has c method(){2}
object updateC
has updateC method() {
   update c():= {plus(3,1)}
   }
object equalC2
has equalC2 method() { eq(c(),2) }
while(equalC2, updateC)
```

Figure 2-7: Programming first class procedures

As mentioned before, subtyping is tied to the inheritance in TSTBC. Therefore for one user-defined type name, S, is to subtype another, T, the object S should inherit from T. Also the other way around, if S inherits from T, S must pass subtyping checks.

Also subtype checking for invocable types are not done at this point, because to be able to check subtype relation between the two generic functions, one needs to use the "contravariant" subtyping rule [Cardelli 88]. This rule states that; if (T1)->T2 is a subtype of (S1)-> S2 then S1 should be subtype of T1 and T2 should be subtype of S2. Castagna states that "functional types are contravariant on the domains and covariant on the codomains" [Castagna 96]. To be able to use contravariant rule, one needs to know the result type of a given function, but since stating the result type is optional in a **`method`** declaration, without a type checker it is not possible to infer the result type of a function, so we were not able to implement the subtype relations for invocable types.

The subtype relations for Conjunctive and Disjunctive types are defined as follows:

3. T subtypes (T1 & T2), if T subtypes T1 and T subtypes T2

3. (T1 & T2) subtypes T, if either T1 subtypes T or T2 subtypes T

3. T subtypes (T1 | T2), if either T subtypes T1 or T subtypes T2

3. (T1 | T2) subtypes T, if T1 subtypes T and T2 subtypes T

The subtyping relations defined above are used to decide the applicability and specificity of methods in a generic function object. For example in Figure 2-8, the first of three test methods is using the conjunction of types `Color` and `Point`, therefore it must be called with a type that subtypes the both types. Assuming that `ColorPoint` is defined as

```
object ColorPoint inherits Color, Point
```

Then `ColorPoint` subtypes both `Color` and `Point` and the first method of the test function is applicable to the `ColorPoint` object.

```
object test
has test method(cp:Color&Point){0}
has test method(cp:Point){1}
has test method(cp:Color){2}
list[
test(ColorPoint),                  --returnes 0
test(Point),                       --returnes 1
test(Color)]                       --returnes 2
```

Figure 2-8: Example for conjunction of type expressions

In Figure 2-9, `ColorPoint`, `Point` or `Color` can be used in to call the same method, since the method is defined using the disjunctive type.

```
has test method(cp:Color|Point){0}
list[
test(ColorPoint),                  --returns 0
test(Color),                       --returns 0
test(Point)]                       --returns 0
```

Figure 2-9: Example for disjunction of type expressions

The use of the **`list`** expression is introduced for the first time in Figures 2-8 and 2-9. The list expression causes each expression inside the square brackets to be executed and the values of each expression is returned in a list. As an example the expression in Figure 2-8 returns the list of values 0, 1 and 2 ([0, 1, 2]). Also the use of characters “--” indicate the beginning of a comment, and rest of that line is treated as a comment.

2.11 Syntactic Sugars

The version of TSTBC that has been presented until this point is the core language. Programming in the core language directly is very tedious. It is also somewhat difficult to program some of the concepts just using the core language. Therefore, syntactic sugars are an important part of TSTBC. They make programming in TSTBC easier. One can read and write examples with ease and see the expressive power of the language.

In the sugars below, we write $\lceil X \rceil$ to indicate the desugaring of a phrase X. This should be understood that if X is a TSTBC phrase, then $\lceil X \rceil$ is just X with its subphrases recursively desugared. TSTBC has two kinds of sugars: declaration, and expression sugars.

2.11.1 Declaration Sugars

All the declaration sugars, except the first two, produce a list of declarations. This list is just inserted at the point where the declaration sugar appears, it does not create a recursive declaration sequence.

The first form of syntactic sugars is already mentioned in section 2.2

⌈**object** *I*⌉ ≡
    object *I* **inherits** any

Objects that are to be used as generic functions are declared using the keyword **gf**. Although this does not shorten the original **object** declaration, it provides more information by saying that this object is intended to be used as an generic function.

⌈**gf** *I*⌉ ≡
    object *I* **inherits** any

The following sugar allows one to declare a function with a single method more easily. Although there is nothing to prevent somebody from extending this generic function with additional methods, this sugar is intended to be used with the generic functions that has only one method attached to it. The two types of **fun** declarations corresponds to the two types of **method** declarations, one with a return type, and one without a return type.

⌈**fun** *I*(*F**)**:***T* **{***B***}**⌉ ≡
    ⌈**gf** *I*⌉
    ⌈**has** *I* **method(***F****):***T* **{***B***}**⌉

⌈**fun** *I*(*F**) **{***B***}**⌉ ≡
    ⌈**gf** *I*⌉
    ⌈**has** *I* **method(***F****)** **{***B***}**⌉

The following sugars declare objects that are to be used as variables and as fields (instance variables). A variable is declared by using a generic function that has a zero-argument method, while a field is declared by a generic function that has a one-argument method. Again the two types of **var** and **field** declarations correspond to the method declaration with or without the result type.

⌈**var** *I***:***T* **:=** *E*⌉ ≡
    ⌈**gf** *I*⌉
    ⌈**has** *I* **method():***T* **{***E***}**⌉

⌈**var** *I* **:=** *E*⌉ ≡
    ⌈**gf** *I*⌉
    ⌈**has** *I* **method(){***E***}**⌉

⌈**field** *I* **of** *CN* **:=** *E*⌉ ≡
    ⌈**gf** *I*⌉
    ⌈**has** *I* **with method(***x***:***CN***)** **{***E* **}**⌉ *where x is a fresh identifier.*

⌈**field** *I***:***T* **of** *CN* **:=** *E*⌉ ≡
    ⌈**gf** *I*⌉
    ⌈**has** *I* **with method(***x***:***CN***):***T* **{***E* **}**⌉ *where x is a fresh identifier.*

2.11.2 Expression Sugars

The first local expression sugar is the following, which creates an “instance” of a class named *CN*.

$\lceil$ **new** *CN* $\rceil \equiv$

  { object *I* **inherits** *CN*

  I

  } *where I is a fresh identifier*

The following expression sugar can be used to make anonymous concrete objects with a given generic function attribute.

$\lceil$ **anon** *GF* $\rceil \equiv$

  { $\lceil$ **gf** *I* $\rceil$

  $\lceil$ **has** *I* *GF* $\rceil$

  I

  } *where I is a fresh identifier.*

The above sugar for anonymous generic functions lets one to desugar simultaneous and sequential eagerly-evaluated let-expressions. (These work for all objects that inherit from `any`.)

$\lceil$ **let** I_1**:**T_1 **=** E_1**,** ..., I_n**:**T_n **=** E_n **in** E_0 $\rceil \equiv$

  $\lceil$ **{ gf** *I*

  has *I* **method(**I_1**:**T_1**,** ...**,** I_n**:**T_n**)** **{**E_0**})**

  I **(**E_1**,** ...**,** E_n**)**

  } $\rceil$ *where I is a fresh identifier.*

$\lceil$ **let** I_1**:**T_1 **=** E_1**;** ...**;** I_n**:**T_n **=** E_n **in** E_0 $\rceil \equiv$

  $\lceil$ **let** I_1**:**T_1 **=** E_1 **in** ... **let** I_n**:**T_n **=** E_n **in** E_0 $\rceil$

For writing control structures, it is helpful to have a sugar for making parameterless procedures (thunks). We adapt part of the syntax for Smalltalk "blocks" for this.

$\lceil$ **[** *B* **]** $\rceil \equiv$

  $\lceil$ **anon method() {** *B* **}** $\rceil$

As an example that uses syntactic sugars consider the Figure 2-10. This is an extended version of the interval example that was presented in Figure 2-5. First is the definitions of `min` and `max` functions which use built in `greater` and `if` functions. Note that, the `lower` and `upper` generic functions are defined as fields of the `interval` object. The object `initialize` is defined as a generic function since it can be extended to be used in the initialization of different objects. The rest of the generic functions are defined as **fun**, because they only have one method and they are not intended to be extended.

```
fun min(a:int, b:int){if(greater(a,b), b, a)}
fun max(a:int, b:int){if(greater(a,b), a, b)}
gf initialize
object interval
hide
   field lower of interval := 0
   field upper of interval := 0
in
   has initialize method (i_o:interval, lowb:int, upb:int) {
      update lower(i_o) := {min(lowb, upb)};
      update upper(i_o) := {max(lowb, upb)};
      i_o}
   fun assign(l:interval, r:interval){
      update lower(l) := {lower(r)};
      update upper(l) := {upper(r)};
      l}
   fun tail_i(l:interval){
      let lower_plus_1:int = plus(lower(l),1)
      in if(greater(lower_plus_1, upper(l)),
               [void],
               [let res:interval = clone(l)
                  in update lower(res) := {lower_plus_1};
                     res ])
   }
   fun head_i(l:interval){lower(l)}
   fun getLower(i_o:interval){lower(i_o)}
   fun getUpper(i_o:interval){upper(i_o)}
end
gf clone
has clone method(r:interval) {assign(new interval, r) }
object oldinterval inherits interval
object newinterval inherits interval
var newer := clone(newinterval)
list[
head_i(newer()),
head_i(initialize(oldinterval, 1, 5)),
let  returnInterval:interval = (tail_i(oldinterval));
     copyInterval:interval = returnInterval
in list [getLower(returnInterval),getUpper(returnInterval),
          getUpper(copyInterval)],
getUpper(assign(newinterval, oldinterval))
]
```

Figure 2-10: Demonstration of syntactic sugars

3 Dynamic Semantics, and Important Semantic Issues

In this section, the dynamic semantics of TSTBC are presented, and some important semantic issues that were raised during implementation are discussed. To express the dynamic semantics of TSTBC, we have used Haskell programming language. It is convenient to use Haskell because this implementation enables direct execution of the denotational semantics [McDonald & Allison 89].

The dynamic semantics of TSTBC mainly deals with following domains: objects, inheritance relations, generic functions and closures. The domains that are used in the implementation of dynamic semantics are presented in Figure 3-1 as Haskell type definitions. In Haskell square brackets (*[.]*) represents a list, types inside a pair of parentheses (separated by commas) represents tuples, triples, etc.(*(.,.,.)*). The arrows (->) represent functions, and the arrow is read as "to". As an example "`double::Int->Int`" is a function from `Int` to `Int`. In Haskell, functions accepts their arguments one at a time, which is called currying. The keyword *type* simply creates type synonyms, while the keyword *data* creates discriminated union of types where the vertical bar (|) is used to indicate alternatives. The symbol "::" represents the type of a function in Haskell (function definition). [Thompson 96].

3.1 Domains

An object in TSTBC is represented as a location (*ObjectId = Location*). When an **`object`** declaration is processed, it allocates a location for that object. This location holds a pair that consists of another location which is also allocated at the time of the object creation, and an inheritance relation (*Inherits*). The location inside the pair holds the methods that objects may have. If object is not acting as a generic function, this location just holds an empty list. If it is a generic function, this location stores a list of methods (*Cases*) that belongs to the generic function. The inheritance relation is defined as a finite relation between object identities, and is maintained within a scope. Whenever an **`object`** declaration is processed, the inheritance relation is updated to include ancestry of the new object. The updated inheritance relation is included inside the newly created object. Therefore objects have the ability to remember their ancestry even if they are used outside of the scope they were created in.

The *Environment* is a finite function from object identifiers to denotable values. Like all the finite functions in this interpreter, it is represented as a list of pairs. The Environment provides a link between the object identifiers and the other domains that keep the object as an identity rather than an identifier such as inheritance relation and store.

The type *Cases* represents the list of methods. Processing a **`has`** declaration updates the cases of an object. Each method is represented as a triple of a *TypeAttributeVec,* an *InstanceTable* and a *ClosureBlock*. The *TypeAttributeVec* is list of *TypeAttribute*s which are the specializers of each method. The *ClosureBlock* holds a closure that consists of the names of the formals, a *Function* that will execute the body of the method, plus the *Context* that is an *Environment* and an *Inherits* pair at the time of method declaration. The *Function* that holds the method body and the *Context* are the default values for a given method; that is, if the method were to be updated using an **`update`** expression, then a new expression closure (*ClosureExp*) would be created. This expression closure has three parts; the expressible values that are used in the update (*ExpressibleValueVec*), a new *Function* which holds the block to be executed, and the new *Context* at the time of update. This expression closure is saved inside the InstanceTable of the method. When the object is called as a

```
data StorableValue            = SVObject Object
                              | SVCases Cases

data ExpressibleValue         =EVInt DInt
                              | EVBool DBool
                              | EVObject ObjectId
                              | EVList ExpressibleValueVec
type ExpressibleValueVec      = [ExpressibleValue]

data DenotableValue           =DVInt DInt
                              | DVBool DBool
                              | DVObject ObjectId
                              | DVList ExpressibleValueVec

type Object                   = (Location, Inherits)          --Location is a pointer to Cases
type Cases                    = Powerset MethodD
type MethodD                  = (TypeAttributeVec, InstanceTable, ClosureBlock)
type InstanceTable            = Powerset ClosureExp
data TypeAttribute            = Id ObjectId
                              | Arrow InvocableType
                              | AndType TypeAttribute TypeAttribute
                              | OrType TypeAttribute TypeAttribute
type TypeAttributeVec         = [TypeAttribute]
type InvocableType            =(TypeAttributeVec, TypeAttribute)
data ClosureBlock             = ClosureB (Formals, Function, Context)
data ClosureExp               = ClosureE (ExpressibleValueVec, Function, Context)

type Environment              = FiniteFun Identifier DenotableValue
type Inherits                 = FiniteRel ObjectId ObjectId
type InheritsIden             = FiniteRel Identifier Identifier
type Elaborated               = (ObjIdEnv, InheritsIden, HasEnv)
type ObjIdEnv                 = FiniteFun Identifier ObjectId
type HasEnv                   = [(Identifier, Generic_Function_Attribute, Context)]

type Context                  = (Environment,Inherits)
qectype ObjectIdVec           = [ObjectId]
type Formals                  = [Identifier]
type Body                     = Block
type Location                 = Integer
type DInt                     = Integer
type Function                 = Environment->Inherits->Store-> (ExpressibleValue,Store)
type DBool                    = Bool
type Identifier               = String
```

Figure 3-1: : Domains for the Dynamic Semantics of TSTBC given as Haskell type definitions. FiniteFun and FineteRel are defined as a list of pairs.

generic function with a set of actuals, the list of expression closures inside the InstanceTable is checked to see if there is an exact matching *ExpressibleValueVec* to the actuals. If there is, a function that is in the same expression closure is executed with a context again in the same closure, otherwise the default function that is inside the *ClosureBlock* is executed with the default *Context.*

3.2 Overview of the Dynamic Semantics

In the following subsections, we will go through the parts of the grammar and describe the dynamic semantics by using the Haskell implementation. The Haskell implementation of the dynamic semantics follows the denotational semantics very closely. As an example we will look at dynamic semantics of a program in both Schmidt's notation [Schmidt 94] and the Haskell implementation to see the resemblance. After that, we will only use the Haskell implementation to explain the rest of the semantics. In Schmidt's notation the double square brackets ([[.]]) is used for the meaning of the term inside the brackets. In the Haskell implementation, this is represented as a function with a name of the form *meaningX,* where the suffix *X* specifies what kind of term the function is applied to. The first line in every function defines the type of the function. Also the Haskell implementation uses the abstract syntax that is defined in the file *AbstrSyn.hs* file which is included in the Appendix.

One important distinction between the Haskell implementation and denotational semantics is that all the values in Haskell include the undefined value $\perp$. For example, a boolean can be True, False or Undefined. Therefore Haskell implementation of booleans corresponds to the denotational representation of Bo $\underline{\varphi}$. 1

3.2.1 Programs

The evaluation of a program takes an initial *Context* and initial *Store*, and produces an expressible value.

Since program is defined as a block in the syntax, the denotational semantics for a program is given as follows. Here b represents a block, and s represents a store

[[.]] : Program$_\perp$ -> Context$_\perp$ -> Store$_\perp$ -> ExpressibleValue$_\perp$
[[b]] context s= *fst* ([[b]] context s)

The Haskell translation of the same semantics is also shown below,

meaningP :: Program -> Context -> Store -> ExpressibleValue
meaningP ((Prog b)) context s = fst(meaningB((b)) context s)

The above code segment says that, the meaning of a program is the *fst* function applied to the meaning of the block. The function *fst* takes a pair and returns the first element in the pair. If one does not have a standard prelude to start with, the initial store and initial context would be empty. However, since TSTBC has predefined objects like `int` and `bool`, and some generic functions such as `if`, `plus`, `minus`, etc., our initial store and initial context includes the definition of those objects. Figure 3-2 shows the initial environment and the initial inheritance relation that m ake up the initial context. The names *la, li, lb, etc.* are the object identities for the objects *any, int, bool, etc*., and they are defined as *Integers* in the file *Locations.hs*. Since initialization of the store is a rather lengthy code segment, it is presented in Appendix (file *TSTBCInitial.hs*).

```
iinitial_env = [ ("any", DVObject la),("void", DVObject lvoid),("true", DVBool True),
                 ("not", DVObject lnot),("and", DVObject land),("or", DVObject lor),
                 ("plus", DVObject lplus),("minus", DVObject lminus),
                 ("eq", DVObject leq),("head", DVObject lhead),
                 ("tail", DVObject ltail),("cons", DVObject lcons),
                 ("append", DVObject lappend),("bool", DVObject lb),("int", DVObject li),
                 ("list", DVObject ll),("less", DVObject lless),
                 ("greater", DVObject lgreater),("if", DVObject lif)]

initial_inh_rel = [ (la,la),(lb,la), (li,la), (ll,la),(lvoid,la),(lnot,la),(land,la),(lor,la),(lplus,la),
                 (lminus,la),(leq,la),(lhead,la),(ltail,la),(lcons,la),(lappend,la),((lless,la),
                 (lgreater,la),(lif,la)]
```

Figure 3-2: Definitions of initial inheritance relation and initial environment for TSTBC

3.2.2 Blocks

The evaluation of a block also takes a *Context* and a *Store* and returns a pair which includes an expressible value plus a new store. Since a block consists of a recursive declaration sequence and an expression, the meaning function for a block first evaluates the recursive declaration sequence, and then takes the new environment and the new inheritance relation that are created by the recursive declaration sequence, and unions them with the original environment and inheritance relations that were in the context and uses this union to evaluate the expression. While evaluating the expression, it also uses the new store, because the recursive declaration sequence modifies the store. (More explanation follows.)

```
meaningB :: Block -> Context -> Store -> (ExpressibleValue, Store)
meaningB (( BL rds e )) context s = meaningE ((e))  (((fst context ) ‘unionMinus‘ env1),
                                                     ((snd context) ‘union‘ inh1)) s1
                       where ((env1, inh1), s1) = meaningRDS((rds)) context s
```

In the above code *snd*, is a function that takes a pair and returns the second element of the pair. *Union* takes two lists and returns a list with all the elements of the two input lists without duplicates. On the other hand, *unionMinus* takes two lists of key, value pairs and returns a list that the keys on the second list shadow the keys on the first. That is, if the same key has a value in both the first and the second list, the final list only keeps the value on the second list. Below is a haskell implementation for *unionMinus*.

```
unionMinus :: Eq a =>FiniteFun a b -> (FiniteFun a b) -> FiniteFun a b
f1 ‘unionMinus‘ f2 = f2 ++ [(x,y) | (x,y) <- f1, not (x ‘elem‘ dom_f2)]
                         where dom_f2 = (dom f2)
```

The Haskell functions are normally applied as a prefix operator to the operands. The back quotes, ‘, provide a mechanism to turn a prefix function into an infix form.

3.2.3 Recursive Declaration Sequence

The evaluation of a recursive declaration sequence takes a *Context* and a *Store* and returns a new *Context* and a new *Store*. The semantics of the recursive declaration sequence is tricky. The main trick is that the *Context* and the *Store*, which are used to evaluate a single declaration, are also the *Context* and the *Store* that are generated by all the declarations in the recursive declaration sequence. For this purpose, we have created a domain *Elaborated*. While processing each declaration individually, we just put all the information that is encountered into an *Elaborated*. The *Elaborated* has three parts: the *ObjectIdEnv* which keeps track of newly created objects (*Identifiers*) and their objects identities, the *InheritsIden*, which keeps track of the inheritance relations that is being created in this scope, and a *hasEnv*, which keeps track of the methods that are being added to the generic functions. The *InheritsIden* keeps track of inheritance relations using object identifiers, since it may not have access to their object identities at the time the declaration is processed. The interpreter treats the recursive declaration sequence as a plain declaration sequence at first. It goes through all the declarations in the declaration sequence and gathers the information that is needed to process the recursive declaration sequence into an *Elaborated*. Once all the information is inside the *Elaborated*, it can be processed into a *Context* and a new *Store*.

The code section that is presented in Figure 3-3 is the implementation of the semantics of a recursive declaration sequence. We refer to the line numbers in further explanations below. In this code segment, first, the recursive declaration sequence is treated as a sequence of declarations and all of them are processed to create the *Elaborated* (line 3). On line 5, the new Environment is being created by using the information inside the *ObjectIdEnv (obenv),* and on line 6, the new environment is unioned with the environment from surrounding scope to be used with all the declarations. The new environment would be the part of the *Context* being returned at the end. The next step is to create the new inheritance relation (lines 11 to 19). To establish a new inheritance relation, one needs to go through the *InheritsIden (inhnames)* and for each pair of identifiers in the set, their corresponding object identities are retrieved from the surrounding inheritance relation shadowed by *ObjectIdEnv (obenv)*. The code segment between lines 7 and 10 retrieves the objects from the surrounding scope while the code segment between lines 11 and 19 creates a new inheritance relation. On line 20, an inheritance relation including both the current and surrounding scope, is put together to be included inside the new objects. The code segment between lines 21 to 28, updates the inheritance relations of the newly created objects on the store, while the code segment between lines 29 to 36 updates all the generic functions with their methods. The generic function attributes that are created using `has` declarations are saved inside the *hasEnv (hasenv)* with the context they are created in. At this point generic function attributes are taken from the *hasenv*, evaluated using *meaningGF* function in a context, which is the current context shadowed by the context that generic function attributes are defined in. Since the current context does not include the definitions that are inside the hide declarations, it is important to shadow the current environment with the environment the generic function attribute is defined in, otherwise the methods of the generic function will not have access to the declarations inside of a hide declaration. In code segment below, the initialize method would not be able to see the `scale` method in the hide declaration.

```
object initialize
hide
   object scale
   has scale method() {0}
```

```
meaningRDS :: Recursive_Declaration_Sequence -> Context -> Store -> (Context, Store)
meaningRDS (( DL ds )) context s =
let (elabt1,s1)= meaningDS((ds)) (newEnv', newInh') s
      (obenv, inhnames, hasenv) = elabt1
      newEnv = [(i, DVObject l) | (i,l) <- obenv]
      newEnv'=((fst context) ‘unionMinus‘ newEnv)
      objectsabove = justObjects (fst context)
      justObjects [] = []
      justObjects ((i,(DVObject l)):xs) = (i,l):(justObjects xs)
      justObjects (_:xs) = (justObjects xs)
      newInh =
               map (\(i,j) -> let i1 = case (lookup i obenv) of
                                                Nothing -> error "Object not allocated"
                                                Just i1 -> i1
                                    j1 = case (lookup j (objectsabove ‘unionMinus‘ obenv)) of
                                                Nothing -> error "Cannot inherit from an object that is not defined'
                                                Just j1 -> j1
                              in (i1,j1))
                     inhnames
      newInh' = ((snd context) ‘union‘ newInh)
      updateinher s [] = s
      updateinher s (n:ns) =
               let   (Just (SVObject (metLoc,_))) = lookupStore(Just objLoc, s)
                     objLoc = case (lookup n obenv) of
                                                Nothing -> error "No location to update"
                                                (Just ll) -> ll
               in updateinher (updateStore (objLoc,SVObject(metLoc,newInh'),s)) ns
      s'' = updateinher s1 (dom obenv)
      updatemethods s [] = s
      updatemethods s (n:ns) =
               let (Just (SVObject (l,inher))) = lookupStore((lookup n obenv), s)
               in updatemethods (updateStore(l , SVCases (foldr (++) []
                                                  [(meaningGF((m))((newEnv' ‘unionMinus‘ (fst c)),
                                                                   (newInh' ‘union‘ (snd c))) s)
                                                         | (n1,m,c) <- hasenv, n1==n]) ,
                                                s)) ns
      call_updatemethods n store ident =
               case  n of
                     (0) -> store
                     (1) -> updatemethods store ident
                     _ ->call_updatemethods (n-1)(updatemethods store ident) ident
      s' = call_updatemethods (length obenv) s'' (dom obenv)
in ((newEnv, newInh), s')
```

Figure 3-3: Evaluation of the recursive declaration sequense

```
in
   has initialize method(intensity:int) {
   update scale():={intensity};
   c}
end
```

The appropriate location is updated to include these methods that are returned by the evaluation of the generic function attributes. The code that is between lines 37 to 42, makes sure that all the methods are included with all the objects, by going through the *obenv* as many times as it has objects in it. This makes sure that a generic function that is defined in terms of the other generic functions will be updated properly, since the iteration is done as many times as we have newly defined objects (*length obenv*, line 42). The following example demonstrates why it is important to iterate over *updatemethods*. In the first iteration, the methods for `gf2` and `gf3` will be empty, therefore `gf1` will have an empty set of methods, but the methods for `gf1` and `gf2` will be updated. The next iteration should see the new methods for gf1 and gf2, and update the gf1 accordingly.

```
object gf1
has gf1 gf2 & gf3
object gf2
object gf3
has gf2 method() { ... }
has gf3 method() { ... }
```

At the end, the new *Context* and the new *Store* that includes all information declared in the recursive declaration sequence are returned.

3.2.4 Declaration Sequences

The evaluation of a declaration sequence takes a *Context* and a *Store* and produces a pair of an *Elaborated* plus a new *Store*. In the declaration sequence, each declaration creates a new *Elaborated* and a new *Store*, since some declarations have side-effects. The meaning of a declaration sequence puts together all the *Elaborated*s that are created by each individual declaration.

```
meaningDS :: [Declaration] -> Context -> Store -> (Elaborated, Store)
meaningDS(( [] )) context s =  (([],[],[]), s)
meaningDS(( d:ds )) context s = (( oenv, (inh1  ‘union‘ inh2), (hasenv1 ‘union‘ hasenv2)),s’)
                                    where oenv = case (oenv1 ‘unionDot‘ oenv2) of
                                                      Nothing -> error "type error"
                                                      (Just oenvNew) -> oenvNew
                                             ((oenv1, inh1, hasenv1),s’’) = meaningD((d)) context s
                                             ((oenv2, inh2, hasenv2),s’)  = meaningDS((ds)) context s’’
```

Figure 3-4: Evaluation of a list of declarations

In Figure 3-4, *d* is a single declaration (the first declaration in the list) and *ds* is the rest of the list. Therefore the meaning function defined for a declaration sequence is recursive. The function *unionDot* makes sure that the same object is not defined more than once. The definition of *unionDot* is given below.

```
unionDot :: Eq a =>FiniteFun a b -> FiniteFun a b -> Maybe (FiniteFun a b)
f1 `unionDot` f2 = if (disjoint f1 f2)  then (Just (f2 ++ f1))
                                        else Nothing
```

3.2.5 Declarations

The meaning of a single declaration takes a *Context* plus a *Store* and produces an *Elaborated* plus a new *Store*. It is important to return a new store, because the **`object`** declarations modify the *Store*. The type definition for the *meaningD* function is:

```
meaningD :: Declaration -> Context -> Store -> (Elaborated, Store)
```

The **`object`** declaration creates a new elaborated. First the *Store* for the methods that the object might have is allocated, then a new *Store* that includes the location for the methods, and empty list of inheritance relation is created for the object itself. The methods and the inheritance relation for the object are updated in the recursive declaration sequence. Now, a pair that contains the object identifier plus the object identity is created to make the *ObjectIdEnv*. Secondly, a list of pairs containing the object identifier plus the identifier of each object that this object inherits from is included in to the *InheritsIden*. As a result, an *Elaborated* with a new *ObjectIdEnv*, a new *InheritsIden* and an empty *hasEnv* is returned.

```
meaningD (( ObjectI i cns )) context s =(([(i,id)], map (\j ->(i,j)) cns, []), s')
                                    where    (lm,sm) = allocateStore(s,SVCases [])
                                             (id,s') = allocateStore(sm,SVObject (lm,[]))
```

AllocateStore is a function that takes a *Store* and a *StorableValue* which is either an object or a set of methods (*Cases*) that belongs to an object and returns a modified *Store* plus a location.

When a **`has`** declaration is processed, the *Elaborated* that is returned includes an empty *ObjectIdEnv*, and *InheritsIden* while its *hasEnv* holds a triple that consists of an object identifier (*cn*), the generic function attribute (*gf*), and the context that is passed to it. This triple allows us to process the generic function attributes in the recursive declaration sequence.

```
meaningD (( Has cn gf )) context s = (([],[],[(cn,gf, context)]), s)
```

The last declaration (**`hide`**) processes its recursive declaration sequence and using the new context generated from this plus the original context, processes its declaration sequence.

```
meaningD (( Hide rds ds )) context s =meaningDS((ds)) (((fst context ) `unionMinus` env1),
                                                     ((snd context)  `union`    inh1)) s1
                                    where ((env1, inh1), s1) = meaningRDS((rds)) context s
```

3.2.6 Generic Function Attributes

The evaluation of a generic function attribute takes a *Context* and a *Store* and produces a set of methods (*Cases*). The type of the *meaningGF* function is:

```
meaningGF:: Generic_Function_Attribute -> Context -> Store -> Cases
```

The first form of a generic function attribute is simply an identifier, this allows TSTBC to add one generic functions' methods to another.

meaningGF((GFIdent i)) context s = case (lookup i (fst context)) of
(Just (DVObject l)) -> (getDcases (Just l) s)
_ -> error "type error"

As as example if one were to define an `eq` generic function in the surrounding scope that has methods for `int` and `bool` objects, we can easily define a new generic function `equal` to include `eq`, and extend the `equal` object to work on the other objects like `list` or `Point`.

```
gf equal
has equal eq
has equal method (p1:Point, p2:Point) { ... }
```

This is an important property, since the generic function `eq` cannot be extended in the nested scope.

The second form of the generic function attribute processes a definition like the last line in above example. It just processes the method declaration and returns a list that holds a single method.

meaningGF((GFMethod m)) context s = ([dmethod])
where dmethod = meaningM((m)) context

The third form is similar to the first form in a way that new generic function is updated with the methods of both *gf1* and *gf2*. If the specializers of *gf1* and *gf2* clash for any method, the methods of *gf2* will overwrite the methods of *gf1*.

meaningGF((GFComb gf1 gf2)) context s =((method1) ‘unionMinusTriple‘ method2)
where method1 = meaningGF((gf1)) context s
method2 = meaningGF((gf2)) context s

3.2.7 Methods

When a method definition is evaluated in a given *Context*, it produces a single method (*Method*D). For the purposes of dynamic semantics, the two forms of method declaration are collapsed into one. If we were to implement the static semantics, the optional return type would have been used to check if the return type implemented is matching the type declared. In dynamic semantics, we do not have a way to infer the result type of a method therefore the return type is ignored. The *meaningM* function creates a method; a *TypeAttributeVec* is formed by using the types of the formal parameters, an empty list as an *InstanceTable*, and a *ClosureBlock* that holds the formal names, method body and the *Contex*t.

meaningM :: Method_Def -> Context ->MethodD
meaningM((Method_t fs t b)) context = meaningM((Method fs b)) context
meaningM((Method fs b)) context =
(t_list, [], ClosureB (i_list, (\env -> \inh -> \s->(meaningB((b)) (env,inh) s)), context))
where t_list = map (\(FA i t) -> meaningT((t)) context) fs
i_list= map (\(FA i t) -> i) fs

3.2.8 Expressions

To evaluate expressions, a *Context* and a *Store* are passed to the *meaningE* function. This returns an *ExpressibleValue* plus a new *Store*. In TSTBC, some expressions have side-effects, therefore returning a modified store is an important part of the expression evaluation. The *meaningE* function has the type:

meaningE :: Expression -> Context -> Store -> (ExpressibleValue, Store)

Evaluating the numerical expressions and identifier expression does not produce any side affects and they are not very interesting. A numerical expression just returns the numeric value, while an identifier expression looks for the identifier in the *Environment*, converts its denotable value to an expressible value and returns the expressible value. If identifier is not found in the environment an "*Unbound identifier*" error is returned.

```
meaningE (( Num n )) context s = (EVInt n, s)
meaningE (( Varref i )) context s = case (lookup i (fst context)) of
                                        Nothing -> error ("unbound identifier " ++ i)
                                        (Just (DVInt j)) -> (EVInt j, s)
                                        (Just (DVBool j)) -> (EVBool j, s)
                                        (Just (DVObject l)) -> (EVObject l, s)
                                        (Just (DVList es)) -> (EVList es, s)
```

To evaluate a **`list`** expression, all the expressions in the list are evaluated from left to right. Considering that one expression might change the store, next expression is evaluated in the newly returned store from previous expression. Then the list is formed from these results.

```
meaningE (( List es)) context s = let   (exValList, s') = exValLHelp es s
                                        exValLHelp [] s = ([], s)
                                        exValLHelp (e:es) s = ((ex:exlist), store)
                                                where (ex, s'') = meaningE ((e)) context s
                                                      (exlist, store) = exValLHelp es s''
                                  in (EVList exValList, s')
```

The most interesting expressions are the generic function applications, and the **`update`** expression.

The code segment in Figure 3-5, evaluates a generic function application. As mentioned in section 2, to apply a generic function, it is important to find the unique most-specific applicable method within the list of methods belonging to the generic function. In function application, first, the expression that is being applied to the actuals is evaluated (line 2). As a second step, the actual arguments are evaluated to obtain their values (line 3). Since TSTBC uses the call-by-value with indirect method of parameter passing, it is important to evaluate the actual arguments at this point. If an actual argument evaluates to a value (like a boolean or an integer), then this value is passed to the method. If it evaluates to an object, the object identity is passed, providing a reference to that object (the indirect part of the parameter passing mechanism). After evaluating the actuals, the methods of the generic function are retrieved (lines 5-7). It is also checked that the expression that is applied is in fact a generic function (an object, not just a value). On line 9, all the methods that are applicable to the list of given actuals are retrieved. Applicability holds if the method being checked has the same number of formals as the number of actuals, and if all the actuals inherit from the corresponding formals, in the inheritance relation, which is passes to the generic function application. If the function *findAppCases* returns an empty list of methods, an error, specifying that no applicable method has found, is returned (line 10-11). If not, from the list of applicable methods, the unique most-specific method must be found. The most specific method is the one such that all its specializers inherit from every other methods' corresponding specializer in the list of applicable methods. The function *findMostSpecific* finds and returns the list of most specific methods (line 13). The list should have exactly one method, if not, either there is not a most specific method

```
meaningE (( Apply gf acts )) context s =
  let     (gfVal, s') = meaningE((gf)) context s
          (actsExpList, s'',cns) =  evalActs acts context s'
          actsDenotList = actsExpList 'seq'  map expToDenot actsExpList
          (dcases,l) = case gfVal of
                    (EVObject l) -> ((getDcases (Just l) s''), l)
                    _ -> error "type error: not a generic function"
          inheritGF = getInherit l s''
          appCases = findAppCases cns dcases (length acts) inheritGF s''
  in case length(appCases) of
          0 -> error "Method not found"
          _ ->let     inh = reflexiveTransitiveClosure (snd context)
                    mostSpecific = findMostSpecific appCases appCases inh
              in case (mostSpecific) of
                    [(typeAttVec, tableOfInstances,ClosureB (formals,f,context2))] ->
                              let   appInstance = getRowOfTable tableOfInstances actsExpList
                                    Just bindFormalsToActuals = (bind formals actsDenotList)
                              in case appInstance of
                                    [] -> f ((fst context2) 'unionMinus' bindFormalsToActuals)
                                          (snd context2)
                                          s''
                                    [ClosureE (exValList, f, context3)]->
                                          f ((fst context3) 'unionMinus' bindFormalsToActuals)
                                            (snd context3)
                                          s''
                                    _ -> error "more then one matching row! Not possible"
                    _ -> error "Method ambiguous"
```

Figure 3-5: Evaluation of the generic function application

(methods in generic functions are not related) or there are more than one most specific methods. In either case, the generic function is ambiguous.

If the unique most-specific method is found, it is time to see if the method has been updated with specific actual arguments. These are stored in the domain *InstanceTable*. At this point the *InstanceTable* of the method is checked, if the exact matching arguments are found the closure that was inside instanceTable that corresponds to the actual arguments is executed. If exact matching actuals are not found inside the *InstanceTable* then the default closure inside the method (*ClosureBlock*) is executed by binding the actuals to the formals of the method.

An important point that was encountered during the implementation of function applications is that, the methods of a generic function should be retrieved from the store, after the evaluation of the actual arguments are done. Otherwise nested function applications do not work properly. To demonstrate this, consider the following methods,

```
has equal method (p1:Point, p2:Point): boolean {
    and (equal(x(p1), x(p2)), equal(y(p1), y(p2))) }
```

```
has initialize method (p:Point, i:int, j:int):Point {
      update x(p):= i;
      update y(p):= j;
      p }
```

If the methods of generic function are retrieved, before the actuals are evaluated, then `x(initialize(MyPoint,1,2),3,4))` would return the default value of `x`, while `initialize(MyPoint,1,2);x(MyPoint)`, returns 1. If the actuals are evaluated first, and the methods of the generic function are retrieved, nested function applications work properly, and both `x(initialize(myPoint,1,2),3,4))` and `initialize(MyPoint,1,2);x(myPoint)` return 1, as we might expect.

Another interesting expression is the **`update`** expression. At first, it follows the same algorithm that function application follows, since one is supposed to find the unique most-specific method that is applicable to the expression list passed to the **`update`** expression. Once the unique most-specific method is found, the *InstanceTable* of that method is updated, the updated method is put back into the old method list and this updated list is saved back to the *Store*.

```
1 meaningE (( Update gf expList en)) context s =
2   let     (gfVal, s') = meaningE((gf)) context s
3           (actsExpList, s'',cns) = evalActs (map (\e->Undir e)expList) context s'
4           (dcases,lgf) = case gfVal of
5                   (EVObject l)->((getDcases (Just l) s''), l)
6                   _ -> error "type error: not a generic function"
7           inheritGF = getInherit lgf s''
8           appCases = findAppCases cns dcases (length expList) inheritGF s''
9   in case length(appCases) of
10          0 -> error "Method not found"
11          _ ->let   inh = reflexiveTransitiveClosure (snd context)
12                  mostSpecific = findMostSpecific appCases appCases inh
13              in case mostSpecific of
14                  [(typeAttVec, tableOfInstances, closure)] ->
15                          let  newInstances=updateRowOfTable en tableOfInstances actsExpList context
16                               newDcases =
17                                    dcases `unionMinusTriple` [(typeAttVec, newInstances, closure)]
18                               lm = case lookupStore((Just lgf),s'') of
19                                          (Just (SVObject (l,i))) -> l
20                          in (EVObject lgf, updateStore(lm, SVCases newDcases , s'))
21                  _ -> error "Method ambiguous"
```

Figure 3-6: Evaluation of a method update

In the code segment that is presented in Figure 3-6, lines 1 to 14 are approximately the same as the code for application, but once the most specific method is found, the *tableofInstances* of the method is updated by using the function *updateRowOfTable (line 15),* then the updated method is put back into the old methods using *unionMinusTriple* (same as *unionMinus* except defined over triples instead of pairs). Lastly the store is updated to reflect the changes that were made to the method.

```
unionMinusTriple :: Eq a => [(a,b,c)] -> [(a,b,c)] -> [(a,b,c)]
f1 `unionMinusTriple` f2 = f2 ++ [(x,y,z) | (x,y,z) <- f1, not (x `elem` domTriple_f2)]
                                   where domTriple_f2 = (domTriple f2)
```

Last two expressions are the block expression (**{***B***}**), and sequence of expressions (*E1***;** *E2*). The evaluation of a block expression just calls the meaning function for blocks. The sequence of expressions evaluated by first evaluating *E1*, and discarding its expressible value, and evaluating *E2* in a new store that is modified by *E1*.

```
meaningE (( BlockExpr b )) context s = (meaningB((b)) context s)
meaningE (( Semi e1 e2 )) context s = meaningE((e2)) context s'
   where (exVal,s') = meaningE((e1)) context s
```

All the helping functions that are used above such as *findApCases, findMostSpecific,* etc., are kept in a file named *TSTBCHelperFunctions.hs* which is included in Appendix.

The code is organized into several files; the domains that are presented at the beginning of this section are stored in a file named *TSTBCDomains.hs*, all the meaning functions and some helper functions which uses meaning functions, are in a file *TSTBCSemantics.lhs*. Files *TSTBCTest.ls,* and *TSTBCFilenames.hs* are used in testing, while files *TSTBCInitial.hs* and *Locations.hs* are used to create the initial *Store* and initial *Contex*t.

4 Conclusions

In this section, we discuss the similarities and differences between TSTBC and BeCecil further, talk about the direction of the future work, and discuss the contributions of the current work.

4.1 Comparisons to BeCecil

As mentioned in the introduction, BeCecil is language that TSTBC is build upon. TSTBC is simpler in many aspects. The main differences are in the type system, treatment of assignment and object extendibility.

Let us first talk about the similarities between the two languages.

Both languages have a prototype-based object model. In prototype-based languages, there is no distinction between the classes and instance of relationship. Lieberman states that, "a prototype represents the default behavior for a concept, and new objects can use part of the knowledge stored in the prototype by saying how the new object differs from the prototype" [Liberman 86]. Object based languages are both simpler and more flexible than traditional class-based languages [Abadi & Cardelli 96]. In section 2, we have showed how to program instance variables by using the prototype based model.

Also both languages have multimethods and multiple inheritance. With multiple inheritance, an object can have more than one parent, enabling it to inherit different behaviors from different objects and combine them into a single interface. With multimethods, methods are collected under generic functions. Every method in a generic function may have different number of specializers and also specialize on different objects. When a generic function is dispatched, which method to apply is decided dynamically depending on the actual arguments and the specializers of the methods inside the generic function.

Another common property of BeCecil and TSTBC is that they both have scoping and encapsulation. All the declarations are visible in the particular section of the code. Scoping can be achieved by hide declarations, blocks inside the methods, and by block expressions. The inheritance relation is only effective for a given scope. At the same time, inheritance relation of the current scope is included in every object that is created in that scope enabling each object to remember its ancestry.

The basic features of TSTBC have been explained in detail in section 2, therefore we will concentrate mostly on the differences in this section.

The main difference between TSTBC and BeCecil is in their treatment of subtyping and inheritance. BeCecil has a static type system that separates code inheritance form the subtyping. The types are declared using **`type`** keyword, so objects are not usable as types. Instead the conformance declaration establishes the connection between the types and objects. Also the subtyping relation between types are declared using another declaration form. Therefore code inheritance has nothing to do with the subtyping relations in BeCecil, and the subtyping relation is declared, not inferred. Although this gives some flexibility to the programmer, this approach of separating inheritance and subtyping is quite complicated, and causes problems with the static typechecking.

Following the examples of many common object oriented languages, TSTBC links code inheritance with subtyping. If an object inherits from another then the first object is also subtype of the second object. The reason for combining subtyping and inheritance was to eliminate some of the problems the BeCecil type system had, and to simplify BeCecil for implementation purposes. Also TSTBC does not have a static type system at this point, although the dynamic semantics are designed while keeping the static semantics in mind. The main constraints that are imposed on the languages for this reason are fixing the inheritance relation of an object at its creation time, and limiting extensibility of a generic function to the scope that is created in.

To see the simplification that is provided by combination of subtyping and inheritance, consider the following example in TSTBC. This example gives us two objects `x` and `y`, and `x` inherits from `y`, we can use both objects as objects, generic functions, or types at the same time and `x` is inferred to be a subtype of `y`.

```
object y
object x inherits y
```

Now we write the same example in BeCecil. As it can be seen from the example in Figure 4-1, it can be somewhat complicated to write the very small example in BeCecil.

```
object x_obj
object y_obj
type x_ty
type y_ty
x_obj conforms x_ty
y_ogj conforms y_ty
x_ty subtypes y_ty
```

Figure 4-1: Defining subtyping relation in BeCecil

Another main difference is that, BeCecil uses storage tables to implement the assignment and instance variables. Storage tables are designed as relations that associate keys to values. Keys are tuple of object identities, and a value is a single object. In BeCecil a generic function can contain both storage tables and methods. Although accessing a storage table looks like a generic function application in reality is a totally different structure that is being processed. BeCecil forms a key from the object identities of the arguments that are used in the application of the storage table, and searches the table for this key. If the key is found, it returns the value associated with the key, if it is not found the default value that is stored in the table is returned. The storage tables are created by using **`storage`** keyword and they are assigned a default value at the time of the creation. The example below shows how a storage table is created.

```
object x
x inherits GenericFun_rep
x has storage(p@Point) := 0
```

To be able to use the storage table we will need an object that has a type `Point`. Assuming that `MyPoint` has a type `Point` we can access the table by using `x(MyPoint)`; we can also update the table with an expression `x(MyPoint):= 3`. In TSTBC and BeCecil, there is no difference between the objects and the generic functions.

In TSTBC, we eliminated storage tables, and instead we implemented method update. This eliminates one extra construct from the language, it also makes it more uniform by treating everything in the language as a generic function and generic functions only contain methods. Since a storage table application and a storage table update look like a generic function application anyway, treating them as generic functions only makes sense. At the same time it provides ease of implementation in that one only deals with generic functions, instead of different structures. If we were to code the example above in TSTBC, we would write;

```
object x
has x method(p:Point){0}
```

To be able to access this variable we would write `x(MyPoint)`, assuming `MyPoint` inherits from `Point`. To be able to update the value of `MyPoint` in the method we would write;

```
update x(MyPoint):={3}
```

The usage of update expression in TSTBC has been explained in earlier sections, but as we can see from the above example, syntactically there is not much difference between the storage tables, and the method update mechanism.

In a storage table, one can only save a value (evaluation of an expression) therefore only a single object to return at the time of retrieval. On the other hand, by using method update mechanism we can specify a block to execute at the time of dispatch. This gives TSTBC more expressive power to execute declarations and expressions that may do different things each time a method is invoked with particular arguments.

In BeCecil, the superclasses and the supertypes of objects can be extended, meaning one can add new superclasses and supertypes to existing objects. In TSTBC we also limit extensibility of objects. New objects can inherit from the surrounding scope, but the inheritance relation, in other words the parents of an object, cannot be changed once the object is created. Also, one cannot add new methods to generic functions that are defined in a surrounding scope. This is a limitation over BeCecil. However it might be required in any case for static typechecking.To be able to extend an object or a generic function gives a programmer greater flexibility. By using extendible objects in nested scopes, one can limit the interface that the client code can see. Only way to extend a generic function in TSTBC would be creating a new object and assigning the old generic function to the new one, and extending the new generic function. The reason for this limitation was mainly to ensure type soundness. If the inheritance relations of the objects changesin an embedded scope it is hard to reason about the generic functions in the surrounding scope, because we have no way of knowing how the inheritance relation might change.

Another main difference between TSTBC and BeCecil is that BeCecil is completely a theoretical language, neither the dynamic semantics nor the static type system has been implemented. In the current work, we have implemented an interpreter for the dynamic semantics of TSTBC.

4.2 Future Work and Conclusions

As has been mentioned several times, the immediate future work would be implementation of the type system for TSTBC. Also, we need to define static semantics formally and prove type soundness. Without formal definitions and soundness proof one cannot be sure that type errors will not happen at run time.

Another advancement of the language would be adding modules to the language to provide better encapsulations mechanisms. Although, TSTBC has an encapsulation mechanism, modules would provide better ways to design and organize the programs. Also one common mechanism that is available in many languages, protected variables, are missing in TSTBC. That would be another way to improve the language.

In this paper, we have demonstrated the expressibility of TSTBC, by explaining how one could program the standard object oriented language features, and first-class procedures using TSTBC. By combining different features of the language, one can simulate the single dispatch languages, or procedural languages using TSTBC. If one only specializes a generic function on a single argument (first, last or his favorite), TSTBC behaves as a single dispatch language. If one does not specialize on any of the arguments, TSTBC acts as a procedural language. This is possible, because all the objects in TSTBC inherit, and therefore subtype, the top level object "`any`", so specializing on `any` virtually means not specializing on that argument.

The new mechanisms of TSTBC also enable it to handle some problems a single dispatch language is not able to, such as the binary methods problem [Bruce et al. 95]. The multiple dispatch mechanism of the generic functions together with inheritance and subtyping lets us handle binary method problems without writing exponential number of methods.

References

[Abadi & Cardelli 96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Arnold & Gosling 98] Ken Arnold, James Gosling. *The Java Programming Language*. Addison Wesley, 1998.

[Bruce *et al.* 95] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theory and Practice of Object Systems*, **1**(3):221-242, 1995.

[Cardelli 88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, **76**(2/3):138-164, February/March, 1988. An earlier version appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pp. 51-66, Springer-Verlag, 1984.

[Castagna 96] Giuseppe Castagna. *Object-Oriented Programming A Unified Foundation,* Birkhäuser, Boston, 1996.

[Chambers 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In O. Lehrmann-Madsen, editor, *ECOOP '92 Conference Proceedings, Utrecht, the Netherlands, June/July, 1992*, volume 615 of *Lecture Notes in Computer Science*, pp. 33-56. Springer-Verlag, Berlin, 1992.

[Chambers 95] Craig Chambers. The Cecil Language: Specification and Rationale: Version 2.0. Department of Computer Science and Engineering, University ofWashington, December, 1995. http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html

[Chambers & Leavens 95] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. *ACM Transactions on Programming Languages and Systems*, **17**(6):805-843. November, 1995.

[Chambers & Leavens 96] Craig Chambers and Gary T. Leavens. BeCecil, A Core Object-Oriented Language with Block Structure and Multimethods: Semantics and Typing. Department of Computer Science and Engineering, University of Washington, UW-CSE-96-12-02, December 1996. Also Department of Computer Science, Iowa State University, TR #96-17, December 1996. ftp://ftp.cs.iastate.edu/pub/techreports/TR96-17/TR.ps.Z; the appendix sections only are in ftp://ftp.cs.iastate.edu/pub/techreports/TR96-17/appendix.ps.Z.

[Feinberg *et al*. 97] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book.* Addison-Wesley Longman, Reading, Mass., 1997.

[Friedman *et al*. 92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. McGraw-Hill, New York, NY, 1992.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1983.

[Liberman 1986] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In Norman Meyrowitz (editor), *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21, number 11 of *ACM SIGPLAN Notices,* pp. 214-223. ACM, New York, November 1986.

[McDonald & Allison 89] C. McDonald and L. Allison. Denotational Semantics of a Command Interpreter and their Implementation in Standard ML. *The Computer Journal,* **32**(5):422-431. October, 1989.

[Paepcke 93] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[Thompson 96] Simon Thompson. *Haskell, The Craft of Functional Programming*. Addison-Wesley, 1996.

[Schmidt 94] David A. Schmidt. *The Structure of Typed Programming Languages.* The MIT Press 1994.

[Shalit 97] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, Reading, Mass., 1997.

[Snyder 86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In Norman Meyrowitz (editor), *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21, number 11 of *ACM SIGPLAN Notices*, pp. 38-45. ACM, New York, November, 1986.

[Steele 90] Guy L. Steele Jr. *Common Lisp: The Language (second edition).* Digital Press, Bedford, MA, 1990.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley, Reading, Mass., 1991.

Appendix

The following files contain the implementation of the interpreter for TSTBC. Files are organized into several groups. Only the files that contain direct information about the language are presented here. The files that contain standard helper functions such as LexerTools.lhs and ParserFunctions.hs are not included because they are taken from the standard library of the course Computer Science 641, Semantic Models for Programming Languages.

- The first group of files contain the dynamic semantics
  - AbstSyn.hs
    - Contains the abstract syntax for TSTBC.
  - TSTBCDomains.hs
    - Contains domains for dynamic semantics.
  - TSTBCSemantics.lhs
    - Contains the meaning functions for each term in the grammar.
  - TSTBCHelperFunctions.hs
    - Contains helper functions that are used in TSTBCSemantics.lhs.
- The second group of files contain the files for parsing the concrete syntax, and desugaring syntactic sugars.
  - TSTBCParser.lhs
    - Parses the concrete syntax.
  - Syntax_Expand.lhs
    - Desugars the syntactic sugars.
  - FreeVars.lhs
    - Finds free and bound variables in a segment of code.
- The last group of codes provides files for testing the implementation.
  - TSTBCTest.hs
    - Contains the main functions for testing.
  - TSTBCInitials.hs
    - Initializes the initial context and the store

```
-- AbstrSyn.hs
--------------------------------
-- Abstract Syntax for BeCecil
--------------------------------

module AbstrSyn (Program(..), Recursive_Declaration_Sequence(..),
       Block(..), Declaration(..), Generic_Function_Attribute(..),
       Type_Expression(..), Method_Def(..),
       Class_Name, Expression(..), Actual_Argument(..), Identifier,
                 Let_Dec(..) , Formal_Argument(..)
       ) where

data Program= Prog B
                           deriving Eq
type P   = Program

data Recursive_Declaration_Sequence
         = DL [D]
                           deriving Eq
type RDS= Recursive_Declaration_Sequence

data Block= BL RDS E
                           deriving Eq
type B   = Block

data Declaration= ObjectI I [CN]
         | Has CN GF
         | Hide RDS [D]
--Everything below here is syntactic sugar for Declarations
                        | Object I
                        | DecList [D]
                        | DeclareGF I
                        | Fun I [F] B
                        | Fun_t I [F] T B
                        | Var I E
                        | Var_t I T E
                        | Field I CN E
                        | Field_t I T CN E
                           deriving Eq
type D   = Declaration

data Method_Def         = Method [F] B | Method_t [F] T B
                           deriving Eq
type M                  = Method_Def

data Generic_Function_Attribute
         = GFIdent Identifier
                        | GFMethod  M
         | GFComb GF GF
                           deriving Eq
```

```
type GF =Generic_Function_Attribute

data Formal_Argument= FA I T
                           deriving Eq
type F   = Formal_Argument

data Type_Expression= TypeNameRef TN
         | [T] :-> T
         | Conj T T
         | Disj T T
                           deriving Eq
type T   = Type_Expression

type TN  = CN

type Class_Name= Identifier
type CN  = Class_Name

data Expression= Num N
                        | List [E]
                        | Varref I
         | Apply E [AA]
                        | Update E [E] B
         | Semi E E
                        | BlockExpr B
--Everything below here is syntactic sugar for Expression
                        | New CN
                        | Anon GF
                        | Let [LL] E
                        | LetS [LL] E
                        | Thunk B
                           deriving Eq
type E   = Expression


data Let_Dec            =LetD I T E
                           deriving Eq
type LL                 =Let_Dec

data Actual_Argument= Undir E
         | E `At` [CN]
                           deriving Eq
type AA  = Actual_Argument

type Identifier= String
type I   = Identifier
type N                  = Integer
```

```
--TSTBCDomains.hs
----------------------------------------
-- Domains for TSTBC Dynamic Semantics
----------------------------------------

module TSTBCDomains where
import AbstrSyn
import FinFun
import qualified Stores
import List

--Integers

type DInt = Integer
type DBool = Bool


--Locations

type Location           = Stores.SLocation

--Stores

type Store = Stores.SStore StorableValue


emptyStore :: Store
emptyStore = Stores.emptyStore

allocateStore :: (Store, StorableValue) -> (Location, Store)
allocateStore = strict Stores.allocate

lookupStore ::(Maybe Location, Store) -> Maybe StorableValue
lookupStore (Nothing, s) = Nothing
lookupStore (Just l,s) = lookup l s

updateStore ::(Location, StorableValue, Store) -> Store
updateStore (l,nval,[]) = error "update error: location not allocated"
updateStore (l,nval,((ll,oval):s)) = if (l == ll)
                                         then (ll,nval):s
                                        else (ll,oval):updateStore(l,nval,s)

--Storable Value

data StorableValue = SVObject Object
                   | SVCases Cases

--Expressible Value

data ExpressibleValue = EVInt DInt
```

```
                      | EVBool DBool
                      | EVObject ObjectId
                      | EVList ExpressibleValueVec
                         deriving Eq
type ExpressibleValueVec = [ExpressibleValue]

data DenotableValue = DVInt DInt
                    | DVBool DBool
                    | DVObject ObjectId
                    | DVList ExpressibleValueVec
                       deriving Eq

type Object            = (Location, Inherits)
                                           --Location is a pointer to Cases
type Cases             = Powerset MethodD
type MethodD           = (TypeAttributeVec, InstanceTable, ClosureBlock)
type InstanceTable     = Powerset ClosureExp
data TypeAttribute     = Id ObjectId
             | Arrow InvocableType
             | AndType TypeAttribute TypeAttribute
             | OrType TypeAttribute TypeAttribute
                         deriving Eq
type TypeAttributeVec= [TypeAttribute]
type InvocableType= (TypeAttributeVec, TypeAttribute)
data ClosureBlock       = ClosureB (Formals, Function, Context)
data ClosureExp         = ClosureE (ExpressibleValueVec, Function, Context)

type Environment        = FiniteFun Identifier DenotableValue
type Inherits           = FiniteRel ObjectId ObjectId
type InheritsIden       = FiniteRel Identifier Identifier

type Elaborated         =(ObjIdEnv, InheritsIden, HasEnv)
type ObjIdEnv           = FiniteFun Identifier ObjectId
type HasEnv            = [(Identifier, Generic_Function_Attribute, Context)]
type Context             = (Environment,Inherits)

type ObjectId           = Location
type ObjectIdVec        = [ObjectId]
type Formals            = [Identifier]
type Function           = Environment->Inherits->Store->
                                                    (ExpressibleValue,Store)
type Body               = Block
```

```
--TSTBCSemantics.lhs
-----------------------------------------------------
-- Dynamic Semantics for TSTBC
-----------------------------------------------------

> module TSTBCSemantics where
> import TSTBCDomains
> import TSTBCHelperFunctions
> import TSTBCParser
> import TSTBCUnparser
> import Locations
> import AbstrSyn
> import FinFun
> import List



          PROGRAMS

> meaningP_Store :: Program -> Context -> Store -> (ExpressibleValue, Store)
> meaningP_Store (( Prog b )) context s = meaningB((b)) context s


> meaningP :: Program -> Context -> Store -> ExpressibleValue
> meaningP (( Prog b )) context s = fst(meaningB((b)) context s)

          BLOCKS

> meaningB :: Block -> Context -> Store -> (ExpressibleValue, Store)
> meaningB (( BL rds e )) context s =
>        meaningE ((e))  (((fst context) `unionMinus`  env1),
>                         ((snd context) `union` inh1))
>                        s1
>    where ((env1, inh1), s1) = meaningRDS((rds)) context s

          RECURSIVE DECLARATION SEQUENCE

Since the declarations do not have to be sequential, we do not process
declarations in to the environment wright away. They are stored in to
Elaborated.
The recursive declaration sequence is first treated as a list of
declarations and the information in each individual declaration is
processed into the Elaborated.
Here in the recursive declaration sequence, we go through the elaborated and
update the environment, inheritance relation and the store with all the
information accumulated into Elaborated from individual declarations


> meaningRDS :: Recursive_Declaration_Sequence -> Context -> Store ->
>                                              (Context, Store)
```

```
> meaningRDS (( DL ds )) context s =
>   let (elabt1,s1) = meaningDS((ds))  (newEnv', newInh') s
>       (obenv, inhnames, hasenv) = elabt1

>       newEnv = [(i, DVObject l) | (i,l) <- obenv]
>       newEnv'=((fst context) `unionMinus` newEnv)

>       objectsabove = justObjects (fst context)
>       justObjects [] = []
>       justObjects ((i,(DVObject l)):xs) = (i,l):(justObjects xs)
>       justObjects (_:xs) = (justObjects xs)
>       newInh =
>        map (\(i,j) ->
>          let i1 = case (lookup i obenv) of
>                    Nothing -> error "Object not allocated"
>                    Just i1 -> i1
>              j1 = case (lookup j (objectsabove `unionMinus` obenv)) of
>                    Nothing ->
>                 error "Cannot inherit from an object that is not defined"
>                    Just j1 -> j1
>          in i1 `seq` j1 `seq` (i1,j1))
>             inhnames
>       newInh' = ((snd context) `union` newInh)

>       updateinher s [] = s
>       updateinher s (n:ns) =
>            let (Just (SVObject (metLoc,_))) = lookupStore(Just objLoc, s)
>                objLoc = case (lookup n obenv) of
>                           Nothing -> error "No location to update"
>                           (Just ll) -> ll
>            in newInh' `seq` objLoc `seq` metLoc `seq`
>               updateinher
>                      (updateStore (objLoc,
>                                   SVObject(metLoc,newInh'),
>                                   s))
>                      ns
>       s'' = let s2 = updateinher s1 (dom obenv)
>             in s1 `seq` s2 `seq` s2
>
>       updatemethods s [] = s
>       updatemethods s (n:ns) =
>        let (Just (SVObject (l,inher))) = lookupStore((lookup n obenv), s)
>        in s `seq`  inher `seq`
>         updatemethods
>          (updateStore(l ,
>                      SVCases (foldr (++) []
>                      [(meaningGF((m))((newEnv' `unionMinus` (fst c)),
>                                      (newInh' `union` (snd c))) s)
>                                 | (n1,m,c) <- hasenv, n1==n]),
>                       s))
```

```
>          ns

>       call_updatemethods n store ident =
>         case  n of
>           (0) -> store
>           (1) -> updatemethods store ident
>           _ ->call_updatemethods (n-1)(updatemethods store ident) ident
>
>       s’ = call_updatemethods (length obenv) s’’ (dom obenv)
>   in s’ ‘seq‘ obenv ‘seq‘
>             ((newEnv, newInh), s’)
>
```

DECLARATIONS

```
> meaningD :: Declaration -> Context -> Store -> (Elaborated, Store)
> meaningD (( ObjectI i cns )) context s =
>             context ‘seq‘ s ‘seq‘
>             ((id ‘seq‘ [(i,id)], map (\j ->(i,j)) cns, []), s’)
>                 where  (lm,sm) = allocateStore(s,SVCases [])
>                        (id,s’) = allocateStore(sm,SVObject (lm,[]))

> meaningD (( Has cn gf )) context s = (([],[],[(cn,gf,context)]), s)

> meaningD (( Hide rds ds )) context s =
>   meaningDS((ds)) (((fst context) ‘unionMinus‘ env1),
>                    ((snd context)  ‘union‘     inh1))
>                    s1
>     where ((env1, inh1), s1) = meaningRDS((rds)) context s
```

DECLARATION SEQUENCE

```
> meaningDS :: [Declaration] -> Context -> Store -> (Elaborated, Store)
> meaningDS(( [] )) context s =  (([],[],[]), s)
> meaningDS(( d:ds )) context s =
>   oenv ‘seq‘ inh1 ‘seq‘ inh2 ‘seq‘ hasenv1 ‘seq‘ hasenv2 ‘seq‘
>         (( oenv, (inh1  ‘union‘ inh2), (hasenv1 ‘union‘ hasenv2)),s’)
>   where oenv = case (oenv1 ‘unionDot‘ oenv2) of
>                  Nothing -> error “type error1”
>                  (Just oenvNew) -> oenvNew
>         ((oenv1, inh1, hasenv1),s’’) = meaningD((d)) context s
>         ((oenv2, inh2, hasenv2),s’)  = meaningDS((ds)) context s’’
```

GENERIC_FUNCTION_ATTRIBUTE

```
> meaningGF:: Generic_Function_Attribute -> Context -> Store -> Cases
> meaningGF((GFIdent i)) context s =
>  context ‘seq‘ s ‘seq‘
```

```
>  case (lookup i (fst context)) of
>    (Just (DVObject l)) -> (getDcases (Just l) s)
>    _ -> error "type error8"
>
> meaningGF((GFMethod m)) context s = ([dmethod])
>        where dmethod = meaningM((m)) context

> meaningGF((GFComb gf1 gf2)) context s =
>            method1 `seq` method2 `seq`
>            ((method1) `unionMinusTriple` (method2))
>                   where method1 = meaningGF((gf1)) context s
>                         method2 = meaningGF((gf2)) context s
```

METHODS

```
> meaningM :: Method_Def -> Context ->MethodD
> meaningM((Method fs b)) context =
>  context `seq`
>  (t_list, [], ClosureB (i_list, (\env -> \inh -> \s->
>                           (meaningB((b)) (env,inh) s)), context))
>         where t_list = map (\(FA i t) -> meaningT((t)) context) fs
>               i_list=  map (\(FA i t) -> i) fs

> meaningM((Method_t fs t b)) context = meaningM((Method fs b)) context
```

EXPRESSIONS

```
> meaningE :: Expression -> Context -> Store -> (ExpressibleValue, Store)

> meaningE (( Num n )) context s =
>     context `seq` s `seq` (EVInt n, s)

> meaningE (( List es)) context s =
>     let (exValList, s')  = exValLHelp es s
>         exValLHelp [] s = ([], s)
>         exValLHelp (e:es) s  =  ((ex:exlist), store)
>               where (ex, s'') =  meaningE ((e)) context s
>                     (exlist, store) = exValLHelp es s''
>     in  (EVList exValList, s')


> meaningE (( Varref i )) context s = s `seq`
>        case (lookup i (fst context)) of
>               Nothing -> error ("unbound identifier " ++ i)
>               (Just (DVInt j)) -> (EVInt j, s)
>               (Just (DVBool j)) -> (EVBool j, s)
```

```
>               (Just (DVObject l)) -> (EVObject l, s)
>               (Just (DVList es)) -> (EVList es, s)
>               (Just _) -> error "not possible"

> meaningE (( Apply gf acts )) context s =
>  let (gfVal, s') = meaningE((gf)) context s
>      (actsExpList, s'',cns) =  evalActs acts context s'
>      actsDenotList = actsExpList `seq`  map expToDenot actsExpList
>      (dcases,l) = gfVal `seq` s'' `seq`
>                   case gfVal of
>                        (EVObject l) -> ((getDcases (Just l) s''), l)
>                        _ -> error "type error: not a generic function"
>      inheritGF = getInherit l s''
```

Find applicable cases:
If tuple of actuals inherit from correcponding elements of tuple of formals then that method is applicable to the actuals.
For an actual to inherit from a formal, all the specializers of that actual should inherit from that formal.

```
>      appCases = findAppCases cns dcases (length acts) inheritGF s''

>  in context `seq` s'' `seq`
>     case length(appCases) of
>      0 -> error "method not found"
>      _ ->let inh = reflexiveTransitiveClosure (snd context)
>              mostSpecific = findMostSpecific appCases appCases inh
>          in
>           case (mostSpecific) of
>          [(typeAttVec, tableOfInstances,ClosureB (formals,f,context2))] ->
>             let appInstance = getRowOfTable tableOfInstances actsExpList
>                 Just bindFormalsToActuals = (bind formals actsDenotList)
>             in case appInstance of
>               [] ->
>                f ((fst context2) `unionMinus` bindFormalsToActuals)
>                       (snd context2)
>                       s''
>               [ClosureE (exValList, f, context3)]->
>                f ((fst context3) `unionMinus` bindFormalsToActuals)
>                       (snd context3)
>                       s''
>               _ -> error "more then one matching row! Not possible"
>            _ -> error "method ambiguous"
>
>

> meaningE (( Update gf expList block)) context s =
>  let (gfVal, s') = meaningE((gf)) context s
>     (actsExpList, s'',cns) = evalActs (map (\e->Undir e)expList) context s'
>      (dcases,lgf) = case gfVal of
```

```
>                     (EVObject l)->((getDcases (Just l) s''), l)
>                     _ -> error "type error: not a generic function"
>      inheritGF = getInherit lgf s''
>      appCases = findAppCases cns dcases (length expList) inheritGF s''
>  in context `seq` s'' `seq`
>     case length(appCases) of
>      0 -> error "method not found"
>      _ ->let inh = reflexiveTransitiveClosure (snd context)
>              mostSpecific = findMostSpecific appCases appCases inh
>          in
>           case mostSpecific of
>            [(typeAttVec, tableOfInstances, closure)] ->
>              let newInstances=
>                updateRowOfTable block tableOfInstances actsExpList context
>                  newDcases = dcases `unionMinusTriple`
>                                  [(typeAttVec, newInstances, closure)]
>                  lm = case lookupStore((Just lgf),s'') of
>                             (Just (SVObject (l,i))) -> l
>              in (EVObject lgf, updateStore(lm, SVCases newDcases , s'))
>            _ -> error "method ambiguous"

>

> meaningE (( Semi e1 e2 )) context s = meaningE((e2)) context s'
>              where (exVal,s') = meaningE((e1)) context s

> meaningE (( BlockExpr b )) context s = (meaningB((b)) context s)
```

TYPE EXPRESSION

```
> meaningT :: Type_Expression -> Context  ->TypeAttribute
> meaningT ((TypeNameRef tn)) context = (Id id)
>                            where id = getId tn (fst context)
```

Arrow types are changed to any for purposes of method applicability.
If we didn't do this, then we would have to record what type a method
has in the inheritance relation, which seems difficult, since the
result type of a method can be omitted.
This may have an undesirable side-effect of making it impossible to specialize
on an argument position that has an arrow type.

```
> meaningT ((ts :-> t)) context = (Id id )
>                            where id = getId "any" (fst context)
> meaningT ((Conj t1 t2)) context = (AndType (meaningT((t1)) context)
>                                       (meaningT((t2)) context))
> meaningT ((Disj t1 t2)) context = (OrType (meaningT((t1)) context)
>                                      (meaningT((t2)) context))
```

Helper Functions that needs to use functions from TSTBCSemantics

--Evaluate actual parameters.

```
> evalActs [] context s = ([],s,[])

> evalActs ((Undir e):acs) context s =
>  case (meaningE((e)) context s) of
>   ((EVObject l), ss)->(((EVObject l):acVals), store, ([l]:special))
>        where (acVals,store, special) = evalActs acs context ss
>   ((EVInt i), ss)->(((EVInt i):acVals), store, ([li]:special))
>        where (acVals,store, special) = evalActs acs context ss
>   ((EVBool b), ss)->(((EVBool b):acVals), store, ([lb]:special))
>        where (acVals,store, special) = evalActs acs context ss
>   ((EVList es), ss)->(((EVList es):acVals),store,([ll]:special))
>        where (acVals,store, special) = evalActs acs context ss
>   _ -> error "not possible"
>
> evalActs ((e 'At' cns):acs) context s =
>  case (meaningE((e)) context s) of
>   ((EVObject l), ss) ->
>    (((EVObject l):acVals),store,(([l]++(idenToId cns context)):special))
>       where (acVals,store,special) = evalActs acs context ss
>   ((EVInt i), ss) ->
>    (((EVInt i):acVals), store, (([li]++(idenToId cns context)):special))
>        where (acVals,store, special) = evalActs acs context ss
>   ((EVBool b), ss) ->
>    (((EVBool b):acVals), store, (([lb]++(idenToId cns context)):special))
>        where (acVals,store, special) = evalActs acs context ss
>   ((EVList b), ss) ->
>    (((EVList b):acVals), store, (([ll]++(idenToId cns context)):special))
>       where (acVals,store, special) = evalActs acs context ss
>   _ -> error "not possible"
```

--Update InstanceTable of a method

```
> updateRowOfTable :: Block -> InstanceTable -> ExpressibleValueVec ->
>                                           Context -> InstanceTable
>
> updateRowOfTable block [] expValList context =
>       [ClosureE (expValList, (\env -> \inh -> \s->
>                           (meaningB((block)) (env,inh) s)), context)]

> updateRowOfTable block ((ClosureE (ids,f,e)):restTable) expValList context
=
>     if (ids == expValList)
```

```
>       then (ClosureE (ids, (\env -> \inh -> \s->
>                    (meaningB((block)) (env,inh) s)), context) : restTable)
>       else (ClosureE (ids,f,e):
>                       updateRowOfTable block restTable expValList context)
>
> getRowOfTable [] expValList = []
> getRowOfTable (ClosureE (ids,f,e):restTable) expValList =
>       if (ids == expValList)
>           then (ClosureE (ids,f,e):getRowOfTable restTable expValList)
>           else getRowOfTable restTable expValList
```

```
--TSTBCHelperFunctions.lhs
-----------------------------------------------
-- Helper functions for TSTBC dynamic semantics
------------------------------------------------

module TSTBCHelperFunctions where
import TSTBCDomains
import AbstrSyn
import FinFun
import List

-- getDcases: Given location and Store, it retrieves the list of methods
--            that belongs to the object stored at the given location.

getDcases ::Maybe Location -> Store-> Cases
getDcases l s = case lookupStore(l,s) of
                  (Just (SVObject (lm,i))) ->
                        (case lookupStore(Just lm,s) of
                              (Just (SVCases dcases)) -> dcases
                              _ -> error "type errorD1")
                  _ -> error "type errorD2"

-- getId: Given an identifier and environment, it retrieves the the location
--        that the object (named by the identifier) is stored in.
getId :: Identifier -> Environment-> ObjectId
getId i env =case (lookup i env) of
                (Just (DVObject ll)) -> ll
                _-> error ("unbound identifier d " ++ i)

--idenToId: given an Identifier list and Context, it returns the
--          object id's for the identifiers.
idenToId :: [Identifier] -> (Environment,Inherits) -> [ObjectId]
idenToId cns envInhT = map (\i -> getId i (fst envInhT) ) cns


-- getInherit: Given a location(Object id) and a store, it retrives the
--             inheritance relation that object holds.
getInherit :: ObjectId -> Store -> Inherits
getInherit l store = case lookupStore((Just l),store) of
                         (Just (SVObject (l,i))) -> i
                         _ -> error "Not Possible d"


--expToDenot: This function is used while converting evaluated actuals to
--            Denotable values before we bind them to formals.
expToDenot :: ExpressibleValue -> DenotableValue
expToDenot (EVInt n) = (DVInt n)
expToDenot (EVObject l) = (DVObject l)
expToDenot (EVBool b) = (DVBool b)
expToDenot (EVList l) = (DVList l)
```

```
denToExp :: DenotableValue -> ExpressibleValue
denToExp (DVInt n) = (EVInt n)
denToExp (DVObject l) = (EVObject l)
denToExp (DVBool b) = (EVBool b)
denToExp (DVList l) = (EVList l)



--isBetterThan: Given an inheritance relation and two lists of object type
--               attributes
--               it checks if first list is better that the second list.
--              (all the ids in the first list inherits from the corresponding
--               id in the second list).
isBetterThan :: Inherits -> [TypeAttribute] -> [TypeAttribute]  -> Bool
isBetterThan inheritance tA1s tA2 =
     let tApair = zip tA1s tA2
         inh = reflexiveTransitiveClosure inheritance
     in all (\(x,y) -> (betterOne inh x y)) tApair


--betterOne: Given an inheritance relation and two type attribute ,
--          checks if first one is better than (inherits from) the second one.
betterOne :: Inherits -> TypeAttribute -> TypeAttribute -> Bool
betterOne inheritance (Id i1) (Id i2) = (i1,i2) `elem` inheritance
betterOne inheritance t (AndType t1 t2) =
      betterOne inheritance t t1 && betterOne inheritance t t2
betterOne inheritance (AndType t1 t2) t =
      betterOne inheritance t1 t || betterOne inheritance t2 t
betterOne inheritance (OrType t1 t2) t =
      betterOne inheritance t1 t && betterOne inheritance t2 t
betterOne inheritance t (OrType t1 t2) =
      betterOne inheritance t t1 || betterOne inheritance t t2


--Find applicable cases:
--If tuple of actuals inherit from corresponding elements of tuple of formals
--then that method is applicable to the actuals.
--For an actual to inherit from a formal, all the specializers of that actual
--should inherit from the formal.

findAppCases :: [ObjectIdVec] -> Cases -> Int -> Inherits -> Store -> Cases
findAppCases  cns cases actLength inheritGF s =
    --Eliminate methods that does not have same number of arguments as actuals
   let sameLengthCases = [(tas, inst, closure)|(tas, inst, closure) <- cases,
                                              actLength == (length tas)]
   in [(typeAtt, inst, closure )|(typeAtt, inst, closure )<- sameLengthCases,
                            (applicable typeAtt cns inheritGF s) ]
```

```
applicable :: TypeAttributeVec -> [ObjectIdVec] -> Inherits -> Store->DBool
applicable [] [] _ _= True
applicable _ [] _ _= error "Too few arguments"
applicable [] _ _ _= error "Too many arguments"
applicable (t:ts) (c:cns) inheritGF s =
        (checkOne t c inheritGF s)  && (applicable ts cns inheritGF s)


--checkOne: given a list of formals and list of list of specializer ids it
--          checks if all the specializer ids inherit from corresponding
--          formal
checkOne :: TypeAttribute -> ObjectIdVec -> Inherits -> Store -> DBool
checkOne (Id i) cn inheritGF s =
 all (\(y,inheritACT) -> y `elem`
         (reflexiveTransitiveClosure (inheritGF `union` inheritACT)))
     inhList
       where
         inhList = map (\x -> ((x,i), (getInherit x s))) cn
checkOne (AndType  t1 t2) cn inheritGF s =
                   checkOne t1 cn inheritGF s && checkOne t2 cn inheritGF s
checkOne (OrType  t1 t2) cn inheritGF s =
                   checkOne t1 cn inheritGF s || checkOne t2 cn inheritGF s

--Find the most specific method

findMostSpecific :: Cases ->Cases -> Inherits -> Cases
findMostSpecific _ [] inh = []
findMostSpecific appCases ((tA1, inst1, closure1):cases) inh =
          if (all (\(tA2, inst2, closure2) -> (isBetterThan inh tA1 tA2))
                  appCases)
          then ((tA1, inst1, closure1): findMostSpecific appCases cases inh)
          else findMostSpecific appCases cases inh
```

--TSTBCParser.lhs

--This module parses the concreate syntax of TSTBC

```
> module TSTBCParser where
> import ParserFunctions
> import LexerTools
> import AbstrSyn
```

The microsyntax is defined by the following definitions,
which use the function "scanning" from the LexerTools module.

```
> tstbcScanner :: [Char] -> [Token]
> tstbcScanner = scanning tstbcKeywords tstbcSymbols []
>    where tstbcKeywords = ["inherits", "has",  "object",
>                           "hide", "in", "end", "gf", "method",
>                           "update", "with", "only", "fun", "var",
>                           "field", "of", "new", "anon", "let",
>                           "in", "list"
>                          ]
>          tstbcSymbols = ["(", ")", "->", "&", "|", ",", ":=", "{", "}",
>                          ":", ";", "=", "[", "]", "+" ]
```

The concrete syntax and parser is the following.
It uses the parser combinators found in the ParserFunctions module.

```
> tstbcParser = program
> program = block              `using`  Prog
> rds = many declaration       `using`  DL
> block:: Parser Token Block
> block = ( rds $~ expression )  `using`  (uncurry BL)
> declaration = ((p_check "object" /$~ p_Id)
>                   $~ (p_check "inherits" /$~ class_name_list))
>                                              `using` (uncurry ObjectI)
>             $| (p_check "object" /$~ p_Id)    `using`  Object
>             $| ((p_check "has" /$~ class_name) $~ generic_function)
>                                              `using` (uncurry Has)
>             $| (p_check "hide" /$~ rds
>                 $~ (p_check "in" /$~ (many declaration))
>                 $~/ p_check "end")            `using` (uncurry Hide)
>             $| (p_check "gf" /$~ p_Id)        `using` DeclareGF
>             $| ((p_check "fun" /$~ p_Id) $~ formals
>                 $~ (p_check "{" /$~ block $~/ p_check "}"))
>                                             `using` (uncurry (uncurry Fun))
>             $| ((p_check "fun" /$~ p_Id) $~ formals
>                 $~ (p_check ":" /$~ type_expression)
>                 $~ (p_check "{" /$~ block $~/ p_check "}"))
```

```
>                             `using` (uncurry (uncurry (uncurry Fun_t)))
>            $| ((p_check "var" /$~ p_Id)
>                $~ (p_check ":=" /$~ expression))
>                           `using` (uncurry Var)
>            $| ((p_check "var" /$~ p_Id)
>                $~ (p_check ":" /$~ type_expression)
>                $~ (p_check ":=" /$~ expression))
>                           `using` (uncurry (uncurry Var_t))
>            $| ((p_check "field" /$~ p_Id)
>                $~ (p_check "of" /$~ class_name)
>                $~ (p_check ":=" /$~ expression))
>                           `using` (uncurry (uncurry Field))
>            $| ((p_check "field" /$~ p_Id)
>                $~ (p_check ":" /$~ type_expression)
>                $~ (p_check "of" /$~ class_name)
>                $~ (p_check ":=" /$~ expression))
>                           `using` (uncurry (uncurry (uncurry Field_t)))

> class_name_list = (some_sep_by (p_check ",") class_name)
>                $| succeed []

> generic_function = (some_sep_by (p_check "&") gf_case)
>                                                `using` (foldl1 GFComb)

> gf_case = p_Id         `using` GFIdent
>        $| method       `using` GFMethod
>        $| (p_check "(" /$~ gf_case $~/ p_check ")")
>
> method  = ((p_check "method" /$~ formals) $~
>            (p_check "{" /$~ block $~/ p_check "}"))
>                                              `using` (uncurry Method)
>        $| ((p_check "method" /$~ formals)
>             $~ (p_check ":" /$~ type_expression) $~
>            (p_check "{" /$~ block $~/ p_check "}"))
>                                    `using` (uncurry (uncurry Method_t))
>
> formals = (p_check "(" /$~ p_check ")") `p_return` []
>        $| (p_check "(" /$~ (some_sep_by (p_check ",") formal)
>              $~/ p_check ")")
> formal = (p_Id $~ (p_check ":" /$~ type_expression)) `using` (uncurry FA)

> expression :: Parser Token Expression
> expression = (some_sep_by (p_check ";") expr)
>                                 `using` (foldl1 Semi)
> expr = (atomic_expr $~ (many actuals ))
>                              `using` (\(gf, als) -> (foldl Apply gf als))
> atomic_expr =
>          p_Number     `using` Num
>       $| (p_check "list" /$~
>            (p_check "[" /$~ expression_list $~/ p_check "]"))
```

```
>                                              `using` List
>       $| p_Id         `using` Varref
>       $| (p_check "{" /$~ block $~/ p_check "}")
>                                              `using` BlockExpr
>       $| (p_check "(" /$~ expression $~/ p_check ")")
>       $| ((p_check "update" /$~ ( atomic_expr $~/ p_check "("))
>           $~ (expression_list $~/ p_check ")")
>           $~ (p_check ":=" /$~
>                 (p_check "{" /$~ block $~/ p_check "}")))
>                              `using` (uncurry (uncurry Update))
>       $| (p_check "new" /$~ class_name)      `using` New
>       $| (p_check "anon" /$~ generic_function)      `using` Anon
>       $| ((p_check "let" /$~ (some_sep_by (p_check ",") let_dec_list ))
>                  $~ (p_check "in" /$~ expression))
>                                              `using` (uncurry Let)
>       $| ((p_check "let" /$~ (some_sep_by (p_check ";") let_dec_list ))
>                  $~ (p_check "in" /$~ expression))
>                                              `using` (uncurry LetS)
>       $| (p_check "[" /$~ block $~/ p_check "]")
>                                              `using` Thunk
> let_dec_list =((p_Id $~/ p_check ":") $~ type_expression
>                $~(p_check "=" /$~ expression))
>                                             `using` (uncurry (uncurry LetD))
> expression_list = (some_sep_by (p_check ",") expression)
>                $| succeed []
> p_Id_list = (some_sep_by (p_check ",") p_Id)
>                $| succeed []
> class_name = p_Id
> type_name = p_Id
```

For type expressions | (disjunction) has lowest precedence, & (conjuction) has higher precedence.

```
> type_expression :: Parser Token Type_Expression
> type_expression =(some_sep_by (p_check "|")type_disjunct)
>                                              `using` foldl1 Disj
> type_disjunct =(some_sep_by (p_check "&") atomic_type)
>                                              `using` foldl1 Conj
> atomic_type =  p_Id `using` TypeNameRef
>              $| ((type_exp_lst $~/ p_check "->") $~ type_expression)
>                                              `using` (uncurry (:->))
>              $| ((p_check "(" /$~ type_expression) $~/ p_check ")")

> type_exp_lst = (p_check "(" /$~ p_check ")") `p_return` []
>         $|(p_check "(" /$~ (some_sep_by (p_check ",") type_expression)
>            $~/ p_check ")")

> actuals = (p_check "(" /$~ p_check ")")      `p_return` []
>         $|(p_check "(" /$~ (some_sep_by (p_check ",") actual)
>            $~/ p_check ")")
```

```
> actual = ((expr $~/ p_check ":") $~ id_lst) `using` (uncurry At)
>       $| expr     `using` Undir

> id_lst = (some_sep_by (p_check "&") p_Id)
```

The read function is an overloaded function that can be used to read in commands from a String. We take the first parse that eats all of the input. If no such parse is available, we fail.

```
> instance Read Program where
>    readsPrec n input = first_parse_by program input

> instance Read Recursive_Declaration_Sequence where
>    readsPrec n input = first_parse_by rds input

> instance Read Block where
>    readsPrec n input = first_parse_by block input

> instance Read Declaration where
>    readsPrec n input = first_parse_by declaration input

> instance Read Generic_Function_Attribute where
>    readsPrec n input = first_parse_by generic_function input

> instance Read Expression where
>    readsPrec n input = first_parse_by expression input

> instance Read Let_Dec where
>    readsPrec n input = first_parse_by let_dec_list input

> instance Read Type_Expression where
>    readsPrec n input = first_parse_by type_expression input
```

The helping function first_parse_by takes the first parse that eats all of the input. If no such parse is available, it fails.

```
> first_parse_by :: Parser Token a -> Parser Char a
> first_parse_by = first_all_consuming tstbcScanner

> first_all_consuming :: (String -> [Token]) -> Parser Token a -> Parser Char
a
> first_all_consuming scanner p input =
>       let parses = p (scanner input)
>       in case filter (\(t,rest) -> rest == []) parses of
>            ((lt,[]):_) -> [(lt,[])]
>            _ -> []
```

```
--Syntax_Expand.lhs
---------------------------------------------------------------------------
--Syntax expands the syntactic sugars from the concreate syntax, into the
--core language.
---------------------------------------------------------------------------

> module Syntax_Expand where
> import AbstrSyn
> import FreeVars
> import List



> class Expandable a where
>  syntax_expand:: a -> a

> instance Expandable Program where
>  syntax_expand (Prog b) = Prog (syntax_expand b)

> instance Expandable Recursive_Declaration_Sequence where
>   syntax_expand (DL ds) = DL (getDecListOut (syntax_expand ds))

> instance Expandable Block where
>  syntax_expand (BL rds e) = BL (syntax_expand rds) (syntax_expand e)

> instance Expandable Declaration where
>  syntax_expand (Object i) = syntax_expand (ObjectI i ["any"])
>  syntax_expand (ObjectI i cns) = ObjectI i cns
>  syntax_expand (Has cn gf) = Has cn (syntax_expand gf)
>  syntax_expand (Hide rds ds) = Hide (syntax_expand rds)
>                                      (getDecListOut (syntax_expand ds))
>  syntax_expand (DeclareGF i) = ObjectI i ["any"]
>  syntax_expand (Fun i fs b) =
>                 DecList [syntax_expand (DeclareGF i),
>                          syntax_expand (Has i (GFMethod (Method fs b)))]
>  syntax_expand (Fun_t i fs t b) =
>                 DecList [syntax_expand (DeclareGF i),
>                        syntax_expand (Has i (GFMethod (Method_t fs t b)))]
>  syntax_expand (Var i e) =
>                 DecList [syntax_expand (DeclareGF i),
>                          syntax_expand (Has i (GFMethod (Method []
>                                                          (BL (DL [])  e))))]
>  syntax_expand (Var_t i t e) =
>                 DecList [syntax_expand (DeclareGF i),
>                          syntax_expand (Has i (GFMethod (Method_t []  t
>                                                          (BL (DL [])  e))))]
>  syntax_expand (Field i cn e) =
>          let newVar = getFreshVar e (TypeNameRef "x") [cn] "i"
```

```
>          in     DecList [syntax_expand (DeclareGF i),
>                          syntax_expand (Has i (GFMethod (Method
>                                              [FA newVar (TypeNameRef cn)]
>                                                 (BL (DL []) e))))]
>  syntax_expand (Field_t i t cn e) =
>          let newVar = getFreshVar e t [cn] "i"
>          in     DecList [syntax_expand (DeclareGF i),
>                          syntax_expand (Has i (GFMethod (Method_t
>                                              [FA newVar (TypeNameRef cn)]
>                                                 t
>                                                 (BL (DL []) e))))]
>

> instance Expandable Method_Def where
>  syntax_expand (Method fs b) = Method (syntax_expand fs) (syntax_expand b)
>  syntax_expand (Method_t fs t b) = Method_t (syntax_expand fs)
>                                              (syntax_expand t)
>                                              (syntax_expand b)

> instance Expandable Generic_Function_Attribute where
>  syntax_expand (GFIdent i) = GFIdent i
>  syntax_expand (GFMethod m) = GFMethod (syntax_expand m)
>  syntax_expand (GFComb gf1 gf2) = GFComb (syntax_expand gf1)
>                                          (syntax_expand gf2)

> instance Expandable Formal_Argument where
>  syntax_expand (FA i t) = FA i (syntax_expand t)

> instance Expandable Type_Expression where
>  syntax_expand (TypeNameRef tn) = TypeNameRef tn
>  syntax_expand (ts :-> t) = (syntax_expand ts) :-> (syntax_expand t)
>  syntax_expand (Conj t1 t2) = Conj (syntax_expand t1) (syntax_expand t2)
>  syntax_expand (Disj t1 t2) = Disj (syntax_expand t1) (syntax_expand t2)

> instance Expandable Expression where
>  syntax_expand (Num n) = Num n
>  syntax_expand (List ls) = List (syntax_expand ls)
>  syntax_expand (Varref i) = Varref i
>  syntax_expand (Apply e acts) = Apply (syntax_expand e)
>                                       (syntax_expand acts)
>  syntax_expand (Update e1 el en) = Update (syntax_expand e1)
>                                       (syntax_expand el)(syntax_expand en)
>  syntax_expand (Semi e1 e2) = Semi (syntax_expand e1) (syntax_expand e2)
>  syntax_expand (BlockExpr b) = BlockExpr (syntax_expand b)
>  syntax_expand (New cn) =
>           let newVar = getFreshVar2 [cn] ("i")
>           in BlockExpr (BL (DL [ObjectI newVar [cn]]) (Varref newVar))
>  syntax_expand (Anon gf) =
>                       let newVar = getFreshVar1 gf ("i")
>                       in BlockExpr (BL
```

```
>                             (DL [(syntax_expand (DeclareGF newVar)),
>                                  (syntax_expand (Has newVar gf))])
>                             (Varref newVar))
>  syntax_expand (Let lds e) =
>    let formals = map (\(LetD i t e1) -> FA  i (syntax_expand t)) lds
>        actuals = map (\(LetD i t e1) -> Undir (syntax_expand e1)) lds
>        gf = GFMethod ( Method formals (BL (DL []) e))
>        newVar = getFreshVar1 gf ("i")
>    in syntax_expand (
>       BlockExpr(BL (DL [(DeclareGF newVar),(Has newVar gf)])
>                    (Apply (Varref newVar) actuals)))
>  syntax_expand (LetS lds e) =
>      syntax_expand (Let [(head lds)] (get_rest (tail lds) e))
> --        (Let [(head lds)] (get_rest (tail lds) e))
>  syntax_expand (Thunk b) =
>       syntax_expand (Anon (GFMethod (Method []  b)))


> instance Expandable Actual_Argument where
>  syntax_expand (Undir e) = Undir (syntax_expand e)
>  syntax_expand (e `At` cns) = (syntax_expand e) `At` cns

> instance Expandable a => Expandable [a] where
>  syntax_expand xs = map syntax_expand xs
```

Helper Functions

```
> get_rest :: [Let_Dec] -> Expression -> Expression
> get_rest lds e
>      |null lds = (syntax_expand e)
>      |otherwise = syntax_expand (Let [(head lds)] (get_rest (tail lds) e ))

> getDecListOut :: [Declaration] -> [Declaration]
> getDecListOut ds = (foldr (++) [] (map getDecListOut_h ds))
> getDecListOut_h :: Declaration -> [Declaration]
> getDecListOut_h (DecList ds) = ds
> getDecListOut_h d = [d]


> getFreshVar expr type_att cnList id =
>  if ( id `elem` (((getFreeList expr) `union` (getFreeList type_att))
>                                                         `union` cnList))
>     then getFreshVar expr type_att cnList (id++"X")
>     else id
> getFreshVar1 gf id =
>  if ( id `elem` (getFreeList gf))
>     then getFreshVar1 gf (id++"X")
>     else id
> getFreshVar2 cnList id =
>  if ( id `elem` cnList)
```

```
>     then getFreshVar2 cnList (id++"X")
>     else id
> getFreeList syntax = free_vars (syntax_expand syntax)
```

```
-- FreeVars.lhs
------------------------------------------------------------
-- Finds free and bound variables from a core language of
-- TSTBC.
------------------------------------------------------------
```

```
> module FreeVars where
> import AbstrSyn
> import List

> class FreeVariables a where
>  free_vars:: a -> [Identifier]

> instance FreeVariables Program where
>  free_vars (Prog p) = (free_vars p)

> instance FreeVariables Recursive_Declaration_Sequence where
>  free_vars (DL ds) = (free_vars ds) `minus` (bound_vars ds)

> instance FreeVariables Block where
>  free_vars (BL rds e) = ((free_vars e) `minus` (bound_vars rds))
>                          `union` (free_vars rds)

> instance FreeVariables Declaration where
>  free_vars (ObjectI i cns) = cns
>  free_vars (Has cn gf) =  [cn] ++ (free_vars gf)
>  free_vars (Hide rds ds) = ((free_vars ds) `minus` (bound_vars rds))
>                              `union` (free_vars rds)
>

> instance FreeVariables Method_Def where
>  free_vars (Method fs b) = ((free_vars b) `minus` (bound_vars fs))
>                            `union` (free_vars fs)
>  free_vars (Method_t fs t b) = (((free_vars b) `minus` (bound_vars fs))
>                            `union` (free_vars fs)) `union` (free_vars t)

> instance FreeVariables Generic_Function_Attribute where
>  free_vars (GFIdent  i) = []
>  free_vars (GFMethod  m) =free_vars m
>  free_vars (GFComb  gf1 gf2) = (free_vars gf1) `union` (free_vars gf2)

> instance FreeVariables Formal_Argument where
```

```
>  free_vars (FA i t) = [i] `union` free_vars t

> instance FreeVariables Type_Expression where
>  free_vars (TypeNameRef tn) = [tn]
>  free_vars (ts :-> t) = (free_vars ts) `union` (free_vars t)
>  free_vars (Conj t1 t2) = (free_vars t1) `union` (free_vars t2)
>  free_vars (Disj t1 t2) = (free_vars t1) `union` (free_vars t2)

> instance FreeVariables Expression where
>  free_vars (Num n) = []
>  free_vars (Varref i) = [i]
>  free_vars (Apply e acts) = (free_vars e) `union` (free_vars acts)
>  free_vars (Update e1 el en) = ((free_vars e1) `union` (free_vars el))
>                                    `union` (free_vars en)
>  free_vars (Semi e1 e2) =  (free_vars e1) `union` (free_vars e2)
>  free_vars (BlockExpr b) = free_vars b
>  free_vars (List ls) = free_vars ls

> instance FreeVariables Actual_Argument where
>  free_vars (Undir e) = free_vars e
>  free_vars (e `At` cns) = (free_vars e) `union` cns

> instance FreeVariables a => FreeVariables [a] where
>  free_vars xs = foldr (++) [] (map free_vars xs)



> class BoundVariables a where
>  bound_vars:: a -> [Identifier]

> instance BoundVariables Program where
>  bound_vars (Prog p) = (bound_vars p)

> instance BoundVariables Recursive_Declaration_Sequence where
>  bound_vars (DL ds) = (bound_vars ds)

> instance BoundVariables Block where
>  bound_vars (BL rds e) = (bound_vars rds) `union` (bound_vars e)
>

> instance BoundVariables Declaration where
>  bound_vars (ObjectI i cns) = [i]
>  bound_vars (Has cn gf) =  bound_vars gf
>  bound_vars (Hide rds ds) = (bound_vars rds) `union` (bound_vars ds)
>

> instance BoundVariables Method_Def where
>  bound_vars (Method fs b) = (bound_vars fs) `union` (bound_vars b)
>  bound_vars (Method_t fs t b) = ((bound_vars fs) `union` (bound_vars b))
>                                 `union` (bound_vars t)
```

```
> instance BoundVariables Generic_Function_Attribute where
>  bound_vars (GFIdent  i) = [i]
>  bound_vars (GFMethod  m) =bound_vars m
>  bound_vars (GFComb  gf1 gf2) = (bound_vars gf1) `union` (bound_vars gf2)

> instance BoundVariables Formal_Argument where
>  bound_vars (FA i t) = [i] ++ bound_vars t

> instance BoundVariables Type_Expression where
>  bound_vars (TypeNameRef tn) = []
>  bound_vars (ts :-> t) = (bound_vars ts) `union` (bound_vars t)
>  bound_vars (Conj t1 t2) = (bound_vars t1) `union` (bound_vars t2)
>  bound_vars (Disj t1 t2) = (bound_vars t1) `union` (bound_vars t2)

> instance BoundVariables Expression where
>  bound_vars (Num n) = []
>  bound_vars (Varref i) = []
>  bound_vars (Apply e acts) = (bound_vars e) `union` (bound_vars acts)
>  bound_vars (Update e1 el en) = ((bound_vars e1) `union` (bound_vars el))
>                                  `union` (bound_vars en)
>  bound_vars (Semi e1 e2) = (bound_vars e1) `union` (bound_vars e2)
>  bound_vars (BlockExpr b) = bound_vars b
>  bound_vars (List ls) = bound_vars ls

> instance BoundVariables Actual_Argument where
>  bound_vars (Undir e) = bound_vars e
>  bound_vars (e `At` cns) = bound_vars e

> instance BoundVariables a => BoundVariables [a] where
>  bound_vars xs = foldr (++) [] (map bound_vars xs)
```

Set functions

```
> minus [] bs = []
> minus (x:xs) bs = if (elem x bs)
>                     then  (minus xs bs)
>                     else  x:(minus xs bs)
```

```
--TSTBCTest.hs
-------------------------------------------------------
--Testing for TSTBC
-------------------------------------------------------

module TSTBCTest where
import TSTBCInitial
import TSTBCSemantics
import TSTBCParser
import TSTBCUnparser
import TSTBCDomains
import AbstrSyn
import Locations
import FinFun
import TSTBCFilenames
import Testing
import Syntax_Expand


evalP :: Program -> String
evalP p =  show (meaningP p (initial_env, initial_inh_rel) initial_store)

repP :: String -> IO ()
repP = putStrLn . evalP . syntax_expand . read

runP :: String -> ExpressibleValue
runP p = let parsedP = (syntax_expand (read p))
         in meaningP parsedP (initial_env, initial_inh_rel) initial_store

-- The following is a read_eval_print loop for programs.

read_eval_print :: IO()
read_eval_print =
  do putStr "? "
     catch (do filename <- getLine
               p1 <- readFile filename
               putStrLn (show (runP p1))
               read_eval_print)
           (\e -> return ())


read_eval_print2 :: IO()
read_eval_print2 =
  do putStr "? "
     catch (do filename <- getLine
               p1 <- readFile filename
               putStrLn  p1
               putStrLn (show (runP p1))
               read_eval_print2)
           (\e -> return ())
```

```
-- The following allows one to read a program in from a file, and run it.
-- Note that Haskell's readFile primitive allows you to get a string
-- for a program from a file

run_from_file :: FilePath -> IO()
run_from_file f =
   do p <- readFile f
      putStrLn p
      repP p

run [] errCount =if errCount == 0
                  then do putStrLn "All tests passed!"
               else do putStrLn ("TOTAL UNEXPECTED RESULTS THIS TEST RUN: "
                                    ++ show errCount)
run ((a,b):abs) errCount =
   do p<- readFile a
      putStrLn ("\nrunning test: " ++ a ++ "\n")
      errs <- run_test (eqTest p (runP p) (b() ))
      run abs (errCount+errs)

go_all = run filenames_with_results 0

go_1  n = run [(filenames_with_results !! n)] 0

---------------------------------------------------------------------------

mainRDS = do
       putStr "Input file: "
       ifile <- getLine
       s <- readFile ifile
       putStr "\n\n********************* Parsing:\n\n"
       putStr (show (syntax_expand (fst (head (first_parse_by rds s)))))
       putStr "\n\n********************* Meaning:\n\n"
       putStr (show (meaningRDS (syntax_expand
                                      (fst (head (first_parse_by rds s))))
            ((initial_env,initial_inh_rel)) initial_store))
       putStr "\n\n********************* "

-------------
mainE = do
       putStr "Input file: "
       ifile <- getLine
       s <- readFile ifile
       putStr "\n\n********************* Parsing:\n\n"
      putStr(show (syntax_expand (fst (head (first_parse_by expression s)))))
       putStr "\n\n********************* Meaning:\n\n"
       putStr(show (meaningE (syntax_expand
                               (fst (head (first_parse_by expression s))))
            ((initial_env, initial_inh_rel)) initial_store))
```

```
       putStr "\n\n********************* "

--------------
mainP = do
       putStr "Input file: "
       ifile <- getLine
       s <- readFile ifile
       putStrLn s
       putStr "\n\n********************* Meaning:\n\n"
       putStr (show (meaningP (syntax_expand(fst (head
                                        (first_parse_by program s))))
            (initial_env, initial_inh_rel) initial_store))
       putStr "\n\n********************* "

--------------
mainPS = do
       putStr "Input file: "
       ifile <- getLine
       s <- readFile ifile
       putStrLn s
       putStr "\n\n********************* Meaning:\n\n"
       putStr (show (meaningP_Store (syntax_expand
                                    (fst (head (first_parse_by program s))))
            (initial_env, initial_inh_rel) initial_store))
       putStr "\n\n********************* "

--------------


mainp = do
        putStr "Input file: "
        ifile <- getLine
        s <- readFile ifile
        putStr "\n\n********************* Parsing:\n\n"
       putStr (show (syntax_expand (fst (head (first_parse_by program s)))))

mainRead = do
           putStr "Input file: "
           ifile <- getLine
           s <- readFile ifile
           putStr "\n\n********************* Parsing:\n\n"
           putStr  (show (fst (head (first_parse_by program s))))

parseIt s = show (syntax_expand (fst (head (first_parse_by program s))))
```

```
--TSTBCInitials.hs
-----------------------------------------------------------------
--Initialization of Environment, Inheritance and Store for TSTBC
-----------------------------------------------------------------

module TSTBCInitial where
import TSTBCSemantics
import TSTBCParser
import TSTBCUnparser
import TSTBCDomains
import AbstrSyn
import Locations
import TSTBCHelperFunctions

emptyEnv = []
emptyInh = []

initial_inh_rel = [(la,la),(lb,la), (li,la), (ll,la),(lvoid,la),(lnot,la),
                   (land,la),(lor,la),(lplus,la),(lminus,la),(leq,la),
                   (lhead,la),(ltail,la),(lcons,la),(lappend,la),
                   (lless,la),(lgreater,la),(lif,la)]

initial_env =
    [("any",      DVObject la),
     ("void",     DVObject lvoid),
     ("true",     DVBool True),
     ("not",      DVObject lnot),
     ("and",      DVObject land),
     ("or",       DVObject lor),
     ("plus",     DVObject lplus),
     ("minus",    DVObject lminus),
     ("eq",       DVObject leq),
     ("head",     DVObject lhead),
     ("tail",     DVObject ltail),
     ("cons",     DVObject lcons),
     ("append",   DVObject lappend),
     ("bool",     DVObject lb),
     ("int",      DVObject li),
     ("list",     DVObject ll),
     ("less",     DVObject lless),
     ("greater",  DVObject lgreater),
     ("if",       DVObject lif)
    ]


initial_store =
    [(la, SVObject (la_cases, initial_inh_rel)),
     (la_cases, SVCases [] ),
     (lvoid, SVObject (lvoid_cases, initial_inh_rel)),
     (lvoid_cases, SVCases [] ),
```

```
(lb, SVObject (lb_cases, initial_inh_rel)),
(lb_cases, SVCases [] ),
(li, SVObject (li_cases, initial_inh_rel)),
(li_cases, SVCases [] ),
(ll, SVObject (ll_cases, initial_inh_rel)),
(ll_cases, SVCases [] ),
(lif, SVObject (lif_cases, initial_inh_rel)),
(lif_cases,
  SVCases [ ([Id lb, Id la, Id la],  [],
         ClosureB
           (["b", "t", "f"],
            (\env -> \inh -> \s ->
                case (lookup "b" env) of
                      (Just (DVBool b)) ->
                       if b then case (lookup "t" env) of
                        Nothing -> error "unbound identifier"
                        (Just (DVObject l)) -> runL l s
                        (Just t) -> (denToExp t, s)
                       else case (lookup "f" env) of
                        Nothing -> error "unbound identifier"
                        (Just (DVObject l)) -> runL l s
                        (Just f) -> (denToExp f, s)
                      _ -> error "type errorI"),
            (emptyEnv,emptyInh)))
            ] ),
(lnot, SVObject (lnot_cases, initial_inh_rel)),
(lnot_cases,
  SVCases [ ([Id lb],  [],
         ClosureB
           (["b"],
            (\env -> \inh -> \s ->
                case (lookup "b" env) of
                      (Just (DVBool b)) -> (EVBool (not b), s)
                      _ -> error "type errorI"),
            (emptyEnv,emptyInh)))
            ] ),
 (land, SVObject (land_cases, initial_inh_rel)),
 (land_cases,
   SVCases [ ([Id lb, Id lb], [],
         ClosureB
           (["a", "b"],
            (\env -> \inh -> \s ->
                    case (lookup "a" env) of
                      (Just (DVBool a)) ->
                            (case (lookup "b" env) of
                              (Just (DVBool b)) -> (EVBool (a && b), s)
                              _ -> error "type errorI1")
                      _ -> error "type errorI1"),
            (emptyEnv,emptyInh)))
            ] ),
```

```
(lor, SVObject (lor_cases, initial_inh_rel)),
(lor_cases,
  SVCases [ ([Id lb, Id lb], [],
        ClosureB
          (["a", "b"],
           (\env -> \inh -> \s ->
                  case (lookup "a" env) of
                    (Just (DVBool a)) ->
                          (case (lookup "b" env) of
                            (Just (DVBool b)) -> (EVBool (a || b) , s)
                            _ -> error "type errorI2")
                    _ -> error "type errorI2"),
           (emptyEnv,emptyInh)))
           ] ),
(lplus, SVObject (lplus_cases, initial_inh_rel)),
(lplus_cases,
  SVCases [ ([Id li, Id li], [],
        ClosureB
          (["a", "b"],
           (\env -> \inh -> \s ->
                  case (lookup "a" env) of
                    (Just (DVInt a)) ->
                          (case (lookup "b" env) of
                            (Just (DVInt b)) -> (EVInt (a + b) , s)
                            _ -> error "type errorI3")
                    _ -> error "type errorI3"),
           (emptyEnv,emptyInh)))
           ] ),
(lminus, SVObject (lminus_cases, initial_inh_rel)),
(lminus_cases,
  SVCases [ ([Id li, Id li],  [],
        ClosureB
          (["a", "b"],
           (\env -> \inh -> \s ->
                  case (lookup "a" env) of
                    (Just (DVInt a)) ->
                          (case (lookup "b" env) of
                            (Just (DVInt b)) -> (EVInt (a - b) , s)
                            _ -> error "type errorI4")
                    _ -> error "type errorI4"),
           (emptyEnv,emptyInh)))
           ] ),
(lgreater, SVObject (lgreater_cases, initial_inh_rel)),
(lgreater_cases,
  SVCases [ ([Id li, Id li],  [],
        ClosureB
          (["a", "b"],
           (\env -> \inh -> \s ->
                  case (lookup "a" env) of
                    (Just (DVInt a)) ->
```

```
                              (case (lookup "b" env) of
                                (Just (DVInt b)) -> if (a > b)
                                                    then (EVBool True, s)
                                                    else (EVBool False, s)
                                _ -> error "type errorI5")
                        _ -> error "type errorI5"),
               (emptyEnv,emptyInh)))
               ] ),
(lless, SVObject (lless_cases, initial_inh_rel)),
(lless_cases,
  SVCases [ ([Id li, Id li],  [],
        ClosureB
          (["a", "b"],
           (\env -> \inh -> \s ->
                  case (lookup "a" env) of
                    (Just (DVInt a)) ->
                          (case (lookup "b" env) of
                            (Just (DVInt b)) -> if (a < b)
                                                then (EVBool True, s)
                                                else (EVBool False, s)
                            _ -> error "type errorI9")
                    _ -> error "type errorI9"),
           (emptyEnv,emptyInh)))
           ] ),
(leq, SVObject (leq_cases, initial_inh_rel)),
(leq_cases,
  SVCases [ ([Id la, Id la],  [],
        ClosureB
          (["a", "b"],
           (\env -> \inh -> \s ->
                  case (lookup "a" env) of
                    (Just (DVInt a)) ->
                         (case (lookup "b" env) of
                           (Just (DVInt b)) -> (EVBool (a == b) , s)
                           _ -> error "type errorI6")
                    (Just (DVBool a)) ->
                         (case (lookup "b" env) of
                           (Just (DVBool b)) -> (EVBool (a == b) , s)
                           _ -> error "type errorI6")
                    (Just (DVList a)) ->
                         (case (lookup "b" env) of
                           (Just (DVList b)) -> (EVBool (a == b) , s)
                           _ -> error "type errorI61")
                    _ -> error "type errorI62"),
           (emptyEnv,emptyInh)))
           ] ),
(lhead, SVObject (lhead_cases, initial_inh_rel)),
(lhead_cases,
  SVCases [ ([Id ll],  [],
        ClosureB
```

```
          (["a"],
           (\env -> \inh -> \s ->
                  case (lookup "a" env) of
                    (Just (DVList exs)) -> (head exs , s)
                    _ -> error "type errorI"),
           (emptyEnv,emptyInh)))
           ] ),
(ltail, SVObject (ltail_cases, initial_inh_rel)),
(ltail_cases,
  SVCases [ ([Id ll],  [],
        ClosureB
          (["a"],
           (\env -> \inh -> \s ->
                  case (lookup "a" env) of
                    (Just (DVList exs)) -> (EVList (tail exs) , s)
                    _ -> error "type errorI"),
           (emptyEnv,emptyInh)))
           ] ),
(lcons, SVObject (lcons_cases, initial_inh_rel)),
(lcons_cases,
  SVCases [ ([Id la, Id ll],  [],
        ClosureB
          (["a", "b"],
           (\env -> \inh -> \s ->
              (case (lookup "b" env) of
                  (Just (DVList exs)) ->
                    (case (lookup "a" env) of
                      (Just (DVObject l)) ->
                                     (EVList ((EVObject l):exs),s)
                      (Just (DVInt i)) ->
                                     (EVList ((EVInt i):exs),s)
                      (Just (DVBool b)) ->
                                     (EVList ((EVBool b):exs),s)
                      (Just (DVList  es)) ->
                                     (EVList ((EVList es):exs),s)
                      _ -> error "type errorI7" )
                  _ -> error "type errorI7")),
           (emptyEnv,emptyInh)))
           ] ),
(lappend, SVObject (lappend_cases, initial_inh_rel)),
(lappend_cases,
  SVCases [ ([Id ll, Id ll],  [],
        ClosureB
          (["a", "b"],
           (\env -> \inh -> \s ->
              (case (lookup "a" env) of
                  (Just (DVList es1)) ->
                    (case (lookup "b" env) of
                      (Just (DVList  es2)) -> (EVList (es1++es2),s)
                      _ -> error "type errorI8" )
```

```
                        _ -> error "type errorI8")),
                  (emptyEnv,emptyInh)))
                  ] )
      ]

runL l s = let cases = (getDcases (Just l) s)
             in case (findApplicable cases s) of
              [] -> (EVObject l, s)
              [([],table,ClosureB(formal,f1,context1))] ->
                 case table of
                    [] -> f1 (fst context1) (snd context1) s
                    [ClosureE (ids,f2,context2)] ->
                              f2 (fst context2) (snd context2) s
              _ -> error "function cannot have more than one () method"

findApplicable [] s = []
findApplicable (case1:cases) s =
     case case1 of
       ([],_,_)-> [case1]
       _ ->findApplicable cases s
```