BeCecil, A Core Object-Oriented Language with Block Structure and Multimethods: Semantics and Typing

Craig Chambers and Gary T. Leavens
Technical Report #UW-CSE-96-12-02
December 1996

This report, minus the appendices, will appear in the proceedings of the *The Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL 4)*, Paris, France, January 1997.

Keywords: Multimethods, generic functions, object-oriented programming languages, encapsulation, information hiding, static typechecking, block structure, subtyping, inheritance, BeCecil 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, type systems; 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, type structure.

BeCecil, a Core Object-Oriented Language with Block Structure and Multimethods: Semantics and Typing

Craig Chambers

Department of Computer Science and Engineering
University of Washington
Box 352350 Seattle, WA 98195-2350 USA
(206) 685-2094; fax: (206) 543-2969
chambers@cs.washington.edu

Gary T. Leavens

Department of Computer Science
Iowa State University
229 Atanasoff Hall, Ames, IA 50011-1040 USA
(515) 294-1580; fax: (515) 294-0258
leavens@cs.iastate.edu

January 6, 1997

Abstract

We present and analyze the semantics and static type system for BeCecil, a theoretical (core) language with multimethods. BeCecil is a simple and orthogonal version of object-oriented languages like Cecil, CLOS, and Dylan. BeCecil has a new, simple mechanism for information hiding, which allows subclassing and yet can preserve representation invariants. BeCecil is also block-structured; within a block, one can extend a generic function with new multimethods, which may come from other generic functions. The inheritance relationships of objects may be extended in any block, and are statically scoped. The type system separates classes from types, and inheritance from subtyping. Subtype relationships are also extensible and statically scoped. These features combine to make BeCecil unusually expressive, while still allowing static typechecking.

1 Introduction

Object-oriented (OO) languages with multimethods [Bobrow *et al.* 86, Moon 86, Chambers 92] offer increased expressiveness over languages with only singly-dispatched methods, for the following reasons:

- dispatching on all arguments is more flexible and symmetric than dispatching on only the first argument;
- multiple dispatching generalizes and unifies global procedures, singly-dispatched methods, and (statically) overloaded procedures;
- multiple dispatching resolves complications relating to co- versus contravariant redefinition and "binary methods" [Bruce *et al.* 95, Castagna 95].

Multimethods also lead to a style of language that permits objects to be extended at several points in a program. By making such extensions in a nested scope, one can customize the interface of classes for particular client applications without polluting the interface as seen by other clients. This style can also support more role-based or subject-oriented programming models [Shilling & Sweeney 89, Wirfs-Brock & Johnson 90, Reenskaug & Anderson 92, Harrison & Ossher 93, Ossher *et al.* 95, Van Hilst & Notkin 96].

However, in comparison with languages with single-dispatched methods, languages with multimethods have received little theoretical attention. Work on the λ&-calculus [Ghelli 91, Castagna *et al.* 92, Castagna *et al.* 95, Castagna 95b] and other languages [Rouaix 90, Agrawal *et al.* 91, Mugridge *et al.* 91, Chambers & Leavens 94, Chambers & Leavens 95] has given some basic ideas for type systems and type checking algorithms. Although CLOS [Steele 90, Paepcke 93] and Dylan [Shalit 97, Feinberg 97] are multimethod languages with module systems, their module systems, which have received little theoretical attention, either permit encapsulation violations or limit extensibility. Thus the design of (statically typed) module systems that both allow interesting extensions (e.g., subclasses) and prevent encapsulation violations and clashes between independently-developed extensions remains an important problem [Cook 90].

We have been developing and experimenting with Cecil, an expressive and practical object-oriented language based on multimethods [Chambers 92, Chambers 95]. We have gained practical experience with programming in Cecil by writing an 80,000-line optimizing compiler, Vortex [Dean *et al.* 96]. We recently presented a static type system, efficient typechecking algorithm, and module system for Cecil [Chambers & Leavens 95]. As part of our efforts to formalize the module system and generalize the underlying object model, we are designing a core language, BeCecil,[*] that includes extensible generic functions, inheritance, and static type checking. BeCecil is intended to directly support the following standard and novel programming idioms.

- BeCecil supports a **prototype-based object model** that unifies classes and instances in a single `object` construct, supports (multiple) inheritance between objects, and supports (mutable) state and object identity.
- BeCecil supports **multimethods**, collected in (first-class) generic function objects. Each multimethod case of a generic function can be a nested, lexically-scoped closure; traditional lexically-scoped functions are modeled by generic function objects with one case. Unlike CLOS or Dylan, multimethods are not artificially linearized in terms of specificity, which permits ambiguous definitions (that need to be prevented by the type system). Instance variables are modeled as special kinds of multimethod implementations, integrating them into the regular message dispatching mechanism. Special message-sending features model `super`-send-like operations to invoke overridden multimethod cases.
- BeCecil supports a **static type system** that separates types from objects and subtyping from code inheritance (objects conform to the types they implement) and guarantees that all dynamically-dispatched messages will find a single most-specific matching multimethod case at run-time. The type system does not force dummy implementations of abstract operations.
- BeCecil supports **extensible, customizable objects**, in that generic functions, superclasses, and supertypes[†] can be added to existing objects or types by external clients of those types. New generic functions can be constructed that build upon all the cases of existing generic functions, in a sense producing a customized version of the original generic function. (BeCecil's extensibility is in marked contrast to the monolithic nature of traditional class declarations in existing object-oriented languages, and is motivated in part by BeCecil's inclusion of multimethods.)
- BeCecil supports **scoping and encapsulation** of all declarations, including object, type, inheritance, subtyping, and multimethod declarations, limiting their static visibility and dynamic effect to a particular region of program text. In addition to traditional uses of encapsulation of hidden state of objects, this feature enables clients to make extensions to existing objects in a nested scope, thereby hiding the extensions from other unrelated clients. (Supporting true scoping of previously-global

[*] **B**lock-structured, **e**xtensible **Cecil**.

[†] New types to which an existing object conforms, and new supertypes of an existing type.

notions such as the inheritance and subtyping graphs and the set of multimethods in a generic function was a challenge.)

- BeCecil will support a notion of **separate typechecking**, where modules (named collections of declarations) can be statically typechecked in isolation from clients, unrelated modules, and even any extending modules. (Modular typechecking is difficult in the face of multimethods, where ambiguities can arise between independent additions of multimethod cases to a common generic function.)

BeCecil strives to make extension of existing code as easy and useful as possible without modification and while preserving static typechecking. This goal leads to multimethods, accessing instance variables solely through messages, separating types from classes, allowing clients to extend existing objects externally (along with controlling the scope of the extensions), and type system support for separate typechecking of separately-developed collections of declarations.

BeCecil currently supports all the desired features listed above except named collections of declarations and separate typechecking; adding modules and modular typechecking is current work. A proof that our type system is sound with respect to the dynamic semantics is also current work.

Although BeCecil is intended to be a core subset of Cecil, it does not model all of Cecil directly. In particular it does not model non-local returns, predicate classes [Chambers 93], and parameterized types.

The rest of this paper is organized as follows. The dynamic semantics is described in three sections. Section 2 gives an overview of language minus its typing aspects, including its syntax and various examples. Appendix A* formally defines the dynamic semantics of the language and Appendix B discusses various design issues related to the dynamic semantics. Similarly, BeCecil's type system is also described in three sections. Section 3 gives an overview of the type system, Appendix C formally defines the type system, and Appendix D discusses various design issues related to the type system. Finally, Section 4 discusses related work in some detail, and Section 5 offers more discussion and conclusions.

* The appendix sections are available from the URL ftp://ftp.cs.iastate.edu/pub/techreports/TR96-17/appendix.ps.Z [Chambers & Leavens 96].

2 Syntax, Overview, Sugars, and Examples for the Untyped Subset of BeCecil

In this section we present the untyped subset of BeCecil. We first present the syntax of this subset, show how to program standard OO mechanisms, present some syntactic sugars, discuss some small examples, and conclude with some informal descriptions of subtle points in the semantics and how BeCecil supports abstract data type (ADT) patterns. We hope that this material will give readers a feel for the language before the detailed dynamic semantics are presented in Appendix A [Chambers & Leavens 96].

2.1 Syntax

The abstract syntax of BeCecil appears in Figure 2-1 below, except for the syntax of the type system, which is explained in Section 3. BeCecil is a call-by-value language (based on an indirect [Friedman *et al.* 92] or

P	::=		*Program*
		RDS **;** *B*	*prelude, followed by a block*
RDS	::=		*Recursive-Declaration-Sequence*
		*D**	*mutual recursion is allowed within D**
B	::=		*Block*
		RDS E	*the recursive declaration sequence provides context for E*
D	::=		*Declaration*
		object *I*	*object: allocates a new object named I*
	|	CN_1 **inherits** CN_2	*inheritance: CN_1 directly inherits from CN_2*
	|	*CN* **has** *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 that names an object (a class)*
GF	::=		*Generic-Function attributes*
		I	*identifier: the methods and tables of I*
	|	**method(***F****)** **{***B***}**	*method: like λ, a block abstraction*
	|	**storage(***F****)** **:=** *E*	*storage table: new table initialized to value of E*
	|	**acceptor(***F****)** **:=** *I* **{***B***}**	*acceptor: can process an assignment with value I*
	|	GF_1 **&** GF_2	*combination: GF_2 favored over GF_1 if clash*
F	::=		*Formal argument*
		I **@** *CN*	*the specializer for this argument is the class CN*
E	::=		*Expression*
		I	*class name: evaluates to the object denoted by CN*
	|	E_0 **(***AA****)**	*application: evaluates operator then operands left-to-right*
	|	E_0 **(***AA****)** **:=** E_{n+1}	*assignment: with key AA* and value E_{n+1}*
	|	E_1 **;** E_2	*sequence: evaluates E_1, then E_2*
AA	::=		*Actual-Argument*
		E	*undirected: applicabile if inherits from specializer*
	|	*E* **@***CN**	*directed: applicable if also a CN_i inherits from specializer*

Figure 2-1: Abstract syntax of BeCecil (minus typing aspects). As usual, an asterisk (*) denotes zero or more repetitions of the preceding nonterminal. In concrete examples we freely use parentheses as well as separators such as commas between repeated phrases.

shared objects) that is, roughly, a superset of the λ&-calculus [Ghelli 91, Castagna *et al.* 92, Castagna *et al.* 95].

2.2 Brief Overview

As a start towards understanding the semantics of BeCecil, we show how to program some standard OO mechanisms in BeCecil, and then describe how to program some other basic mechanisms.

2.2.1 How to Program Standard OO Mechanisms in BeCecil

An object in BeCecil can be created only by an object declaration. Objects in a program can act as classes or instances in other OO languages. Each object has a unique identity. For example, the following could be used to declare a class `Point_rep`. (We use the suffix `_rep`, which stands for "representation," to distinguish classes from types in examples.)

```
object Point_rep
```

An object may be declared to inherit from some other object by an inheritance declaration. One can use more than one inheritance declaration with the same object named on the left-hand side to achieve multiple inheritance. For example the following declares that `Point_rep` inherits from an object named `any`, and declares another object, `CP_rep` (for colored points), that inherits from `Point_rep` and an object named `Color_rep`. The object named `any` is used in our examples as an object from which (almost) all objects inherit.

```
Point_rep inherits any
object CP_rep
CP_rep inherits Point_rep
CP_rep inherits Color_rep
```

We chose to make BeCecil simpler by not distinguishing the inheritance and "instance-of" relationships. In this respect BeCecil resembles prototype-based languages [Ungar & Smith 87, Chambers92, Chambers 95, Blascheck 94, Abadi & Cardelli 96]. 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_rep` as follows.

```
object myPoint
myPoint inherits Point_rep
```

Objects in BeCecil can also act as generic functions. A generic function in BeCecil is, roughly, a collection of multimethods, as in CLOS [Steele 90, Paepcke 93], Cecil, or Dylan [Shalit 97, Feinberg *et al.* 97]. (Unlike CLOS or Dylan, however, the methods in a BeCecil generic function need not all take the same number of arguments.) For example to declare the generic function `equal`, one would write the following. In our examples, generic functions inherit from an object named `GenericFun_rep` (which itself inherits from `any`).

```
object equal
equal inherits GenericFun_rep
```

Assuming that the generic functions `x` and `y` are implemented for points, the following extension (`has`) declaration can be used to extend the generic function `equal` with a method specialized for two `Point_rep` arguments.

```
equal has method(p1@Point_rep, p2@Point_rep) {
   and(equal(x(p1), x(p2)), equal(y(p1), y(p2)))
}
```

An extension declaration is similar to a `method` declaration in Cecil, and to `defmethod` in CLOS. The formals have the form *I*`@`*CN*, where *I* is the name of the formal and *CN* is the name of the formal's

specializer object. Note that there is a formal for each argument, unlike single-dispatch languages, such as Smalltalk [Goldberg & Robson 83] or C++ [Stroustrup 91], where there is a distinguished "receiver" (written `self` or `this`) that is not mentioned in the list of formals. The body of the method is written between the curly braces. 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. A method is *applicable* to a tuple of actuals if each actual inherits from the corresponding specializer. When a generic function is invoked, the generic function must have a unique, most-specific method that is applicable to the actuals; that method is then called. For example, the method above is applicable to two arguments that both inherit from `Point_rep`, because the specializers of the method are both `Point_rep`.

Instance variables of an object are modeled by *storage tables*. A storage table can be thought of as a table, or relation, that associates *keys* to *values*. Keys consist of a tuple of object identities, and a value is a single object. In BeCecil, a generic function can contain both storage tables and methods; reading a storage table looks like applying a generic function. In such an application, the key is formed from the identities of the arguments of the invocation. If no value is currently stored for the given key, the table's *default value* is returned. For example, consider the following declarations. These declare two objects that act as instance variables, `x` and `y`. Each instance variable is modeled by a generic function with a storage table attribute. Each storage table declared can be thought of as an association between points and integers (objects that inherit from `Point_rep` and objects that inherit from `int_rep`). The identities of points are the keys to these tables, and the associated integers are the values. The expression following `:=` in each storage table declaration, `0` in this example, is the default value.

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

Since `myPoint` inherits from `Point_rep`, all methods and storage tables that have formal arguments that are specialized on `Point_rep` are applicable to `myPoint`. For example, the expression `x(myPoint)` returns the `x` coordinate for `myPoint`, which at this point is the default value 0.

Storage tables can be modified using an assignment expression; in an assignment, the arguments to the left of the `:=` form the key, and the expression to the right of the `:=` gives the value to be associated with that key. Thus one can set the `x` coordinate of `myPoint` to 3 by using the following assignment expression.

```
x(myPoint) := 3
```

BeCecil also has acceptors, which are like methods that can process an assignment. Having acceptors allows any storage table to be replaced by an acceptor and a method, and also permits the equivalent of "write-only" fields. When used in an assignment, the key is bound to the formals, and the value to the right of the `:=` is bound to the identifier that follows the `:=` in the acceptors declaration. For example, the following declares a generic function, `xy`, with an acceptor that assigns the value given to both the `x` and `y` coordinates of the point that forms the key.

```
object xy
xy inherits GenericFun_rep
xy has acceptor(p@Point_rep) := v {
   x(p) := v;
   y(p) := v
}
```

Thus, the following expression would return 28.

```
xy(myPoint) := 14;
plus(x(myPoint), y(myPoint))
```

Dispatch in assignments is also dynamic, but only considers the specializers of the generic function's acceptors and storage tables. A generic function's methods are not considered when dispatching an assignment. Similarly, when dispatching an application, only the applicability of its methods and storage tables matters.

The pattern of object creation and initialization can be easily encapsulated in a generic function, which allows one to create new objects dynamically. In BeCecil, it is often convenient to program both an initializer (as in Modula-3 [Harbison 92, Nelson 91]; these are called constructors in C++) and a separate primitive constructor (similar to a class method in Smalltalk), so that the initializer can be inherited.

```
object initialize
initialize inherits GenericFun_rep
initialize has method(p@Point_rep, i@int_rep, j@int_rep) {
   x(p) := i;
   y(p) := j;
   p
}
object mkPoint
mkPoint inherits GenericFun_rep
mkPoint has method(x@int_rep, y@int_rep) {
   object res
   res inherits Point_rep
   initialize(res, x, y)
}
```

Note that, because the declaration of `res` in `mkPoint` is inside a nested recursive declaration sequence (the declarations of the block that forms the body of that method), it creates a new object each time that method is called. Thus block structure allows one to create objects dynamically, even though objects are only created by object declarations.

To achieve information hiding, we chose to add just the capability to hide a recursive declaration sequence, as this gives one the ability to hide generic functions. This capability is found in BeCecil's hide declaration. A hide declaration is similar to a `local` declaration in Standard ML [Milner *et al.* 90] in that the declarations in its recursive declaration sequence are only visible in the declaration sequence that follows the keyword `in`. (However, unlike Standard ML, the declarations are all mutually recursive by default.) As an example, consider the implementation of a gray-scale model of colors, in which the interface presented to clients allows them to create a color based on gray intensities given as floating-point numbers between 0.0 and 1.0, but for which the hardware uses intensities specified by integers between 0 and 255. Since the hardware may change, and since we wish to enforce the invariant that every instance has all its intensities given by integers between 0 and 255, we hide the storage table for instances of this class. Notice how the code, given below, resembles, say, a C++ class declaration, with a private part that comes before the public part. (Comments start with two hyphens, and continue to the end of that line, as in Ada [Ada 83] and Cecil.

We assume that declarations for generic functions `intensity` and `paint`, several generic functions to manipulate integers and floating point numbers, and a class `Region_rep` are given elsewhere.)

```
object Grayscale_rep
Grayscale_rep inherits Color_rep
hide
   -- private declarations
   object scale
   scale inherits GenericFun_rep
   scale has storage(c@Grayscale_rep) := 0
in
   -- public declarations
   initialize has method(c@Grayscale_rep, intensity@float_rep) {
      scale(c) := truncate(multiply(min(max(0.0, intensity), 1.0), 255.0));
      c
   }
   intensity has method(c@Grayscale_rep) {
      divide(mkFloat(scale(c)), 255.0)
   }
   paint has method(c@Grayscale_rep, r@Region_rep) { ... }
end
intensity has acceptor(c@Grayscale_rep) := f {
   initialize(c, f)
}
```

It is worth thinking about how the above code maintains the invariant that, for every object *o* that inherits from `Grayscale_rep`, the value of `scale`(*o*) is between 0 and 255. In stating this invariant, by `scale`(*o*), we mean the result of the generic function `scale` defined in the hidden declarations. (Clients may define their own generic function `scale`, with a method specialized on `Grayscale_rep`, but because of static scoping, the generic function `scale` referred to within the hide declaration is always the one defined in the private part of the hide declaration.) The storage table `scale` has a default value of 0, which satisfies the invariant. The only way that the storage table can associate the identity of *o* with some other value is by the use of the `initialize` method, because no other method in the hide declaration assigns to that storage table, no method within the hide declaration returns the `scale` generic function, and no code outside the hide declaration can access the `scale` generic function. Since `initialize` preserves the invariant, the invariant always holds.

Notice that the argument for the preservation of the invariant given above does not depend on *o* being created by a special generic function, such as `mkGrayscale` given below. The generic function `mkGrayscale` is something that any client can write, as it is written outside of the hide declaration.

```
object mkGrayscale
mkGrayscale inherits GenericFun_rep
mkGrayscale has method(intensity@float_rep) {
   object res
   res inherits Grayscale_rep
   initialize(res, intensity)
}
```

However, a client can, outside the hide declaration, write a declaration of a subclass that does not have the implicit behavioral properties of an instance of `Grayscale_rep` that was created by `mkGrayscale`. This can be done by overriding all the public methods and specializing them on the new subclass. The prevention of such imposters is discussed in Section 2.6 below.

2.2.2 Multimethods and Inheritance of Methods

Because BeCecil uses multiple-dispatch instead of single-dispatch, there is no trouble in programming binary methods [Bruce *et al.* 95] such as `equal`, as we have seen above. Moreover, 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 [Chambers 92, Castagna 95].

To invoke an overridden method inherited from a superclass in BeCecil, one can use the directed form of actual arguments. In this form, one writes an expression followed by an at-sign (`@`), and a list of class names. The method selected must be such that both the object that is the value of the expression and at least one of the named classes inherit from the corresponding formal parameter's specializer. For example, the following is how one would write a class of gray-scale points, `GrayPoint_rep`, as a subclass of `Point_rep` and `Grayscale_rep`. Notice how the `equal` and `initialize` methods use directed actual arguments to call the methods specialized on `Point_rep` and `Grayscale_rep` as part of their work. This is similar to the use of `super` in Smalltalk, or `Point_rep::` and `Grayscale_rep::` in C++.

```
object GrayPoint_rep -- grayscale points
GrayPoint_rep inherits Point_rep
GrayPoint_rep inherits Grayscale_rep
initalize has method(gsp@GrayPoint_rep, i@int_rep, j@int_rep,
                     intensity@float_rep) {
   initialize(gsp@Grayscale_rep, intensity);     -- calls initialize for Grayscale_rep
   initialize(gsp@Point_rep, i, j);              -- calls initialize for Point_rep
   gsp
}
object mkGSP
mkGSP inherits GenericFun_rep
mkGSP has method(i@int_rep, j@int_rep, intensity@float_rep) {
   object res
   res inherits GrayPoint_rep
   initalize(res, i, j, intensity)
}
equal has method(gsp1@GrayPoint_rep, gsp2@GrayPoint_rep) {
   and(equal(intensity(gsp1), intensity(gsp2)),
       equal(gsp1@Point_rep, gsp2@Point_rep))     -- calls equal for Point_rep
}
```

2.2.3 Programming Variables

Storage tables can also be used to program variables in BeCecil. The only trick is to use a tuple of zero arguments for the key. The table then associates the zero-tuple to some object, which can be changed by an assignment expression. For example, the following block returns 227.

```
object my_var
my_var has storage() := 0
my_var() := 226;
plus(my_var(), 1)
```

2.2.4 Programming First-Class Procedures

Because generic functions are objects, they can be used as first-class procedures. For example, consider the following generic function that can act as a while-loop. This generic function `while` has a method that takes two generic function arguments: a condition (`c`), and a statement (`s`). These generic functions are expected to have zero-argument methods (thunks) that can be invoked to perform parts of the loop. The implementation of the method below relies on another higher-order generic function, `ifTrue`, which takes

a boolean and a generic function as its arguments, and only calls the generic function with zero arguments if the boolean is true. The generic function, `loop`, passed as the second argument to `ifTrue` is declared inside the method for `while`. The usual static closures are made for methods, and thus `c` and `s` within `loop` refer to the actual arguments passed to `while`.

```
object while
while inherits GenericFun_rep
while has method(c@GenericFun_rep, s@GenericFun_rep) {
   object loop
   loop inherits GenericFun_rep
   loop has method() {s(); while(c, s)}
   ifTrue(c(), loop)
}
```

2.3 Some Useful Sugars

As can be seen from the above examples, BeCecil is a bit tedious to program in, because it is a core language. So it is helpful to define a few syntactic sugars to make the programming of more interesting examples easier to write and to read. These sugars also give one some idea of the expressive power of BeCecil, and can be used to compare BeCecil to other core OO languages, such as the Abadi and Cardelli calculus [Abadi & Cardelli 95] or Castagna's λ_object [Castagna 95b].

In the sugars below, we write $\lceil X \rceil$ to indicate the desugaring of a phrase *X*. (It should be understood that if *X* is a BeCecil phrase, then $\lceil X \rceil$ is just *X* with its subphrases recursively desugared.) We first discuss declaration sugars, then formal argument and expression sugars.

2.3.5 Declaration Sugars

The declaration sugars in this section generally produce a sequence of declarations, which are to be inserted at the point where the sugar occurs. That is, the desugaring does not produce a nested declaration sequence, but simply several declarations at the same level as the sugar appears. For example, consider the following sugared declarations.

```
object o1
o1 inherits y, z, any
object q
```

The middle declaration above is a sugared declaration form. The desugaring rule for it is the following.

$$\lceil CN_0 \ \textbf{inherits} \ CN_1,\ldots,CN_n \rceil \equiv$$
$$CN_0 \ \textbf{inherits} \ CN_1$$
$$\ldots$$
$$CN_0 \ \textbf{inherits} \ CN_n$$

Therefore the desugaring of the above example produces the following sequence of declarations.

```
object o1
o1 inherits y
o1 inherits z
o1 inherits any
object q
```

Like the first desugaring rule given above, the following declaration sugar also allows one to more succinctly declare direct inheritance relationships.

$$\lceil \textbf{object} \ I \ \textbf{inherits} \ CN_1,\ldots,CN_n \rceil \equiv$$
$$\textbf{object} \ I$$
$$\lceil I \ \textbf{inherits} \ CN_1,\ldots,CN_n \rceil$$

In practical examples, it is convenient to have ways to declare objects that are to be used as generic functions. Hence the following sugars.

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

The following sugar allows one to declare a function with a single method more easily.

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

The following sugars allows one to conveniently declare objects that are to be used as variables and as fields (instance variables). A variable is modeled by a generic function with a zero-argument storage table. A field is modeled by a generic function with a one-argument storage table. The argument of such a storage table is the "record," and the value of that record's field is returned or updated by invoking or assigning to the storage table.

⌈**var** *I* **:=** *E*⌉ ≡
    ⌈**gf** *I*⌉
    I **has storage() :=** ⌈*E*⌉
⌈**field** *I* **of** *CN* **:=** *E*⌉ ≡
    ⌈**gf** *I*⌉
    I **has storage(***x***@***CN***) :=** ⌈*E*⌉, where *x* is a fresh identifier.

2.3.6 Formal Argument Sugar

If the object `any` is used as a class from which (almost) all others inherit, as in Cecil and our examples, then a formal parameter specialized on `any` is effectively not specialized at all. Thus the following sugar for formal arguments allows one to omit the specializer if it is `any`.

⌈*I*⌉ ≡
    ⌈*I* `@ any`⌉

2.3.7 Expression Sugars

For real programming, it is quite convenient to be able to use blocks as expressions. Such a sugar makes it easy to have further sugars that made anonymous concrete objects with a given generic function value (like `lambda` in Scheme [Clinger & Rees 91]), various flavors of `let`-expressions, and even Smalltalk-like thunks, as we shall see. However, unlike the sugars described above, the sugar for block expressions, of the form { *B* }, is not a local expression sugar, as it involves adding new declarations to the block or declaration list in which the block expression appears. Because such a sugar is somewhat complex to explain, we do not describe it formally, but instead give an example to clarify how this could be done. Consider the following.

```
...-- some declarations
plus({var x := 3
      times(x(),x())},
     {var y := 4
      fun f(z@int_rep) {y() := plus(y(),5); y()}
      f(y())
      }
    )
```

This would desugar as follows, by adding a new method (named by a fresh identifier) for each nested block, and a call to that method in the place where the nested block occurred in the original expression.

```
...-- some declarations
fun block1() {
   var x := 3
   times(x(),x())
}
fun block2() {
   var y := 4
   fun f(z@int_rep) { y() := plus(y(),5); y() }
   f(y())
}
plus(block1(), block2())
```

By iterating this process one can eliminate all such block expressions and still preserve scoping. Using this, it is even possible to regard sequence expressions as a syntactic sugar [Abadi & Cardelli 95]. However, because these are not local sugars (they do not operate on expressions in place), we have chosen to include sequence expressions directly in the syntax of BeCecil. In some sense, it would be simpler to add block expressions as a primitive to BeCecil, enabling all of these other sugars, but that complicates the proof of type soundness by making the syntax of expressions and blocks (and hence declarations) mutually recursive. The present desugared syntax does not allow blocks within expressions, and thus somewhat simplifies the proof. In what follows, we regard block expressions, of the form `{` *B* `}`, as syntactic sugars.

Assuming the sugar for blocks, the first local expression sugar is the following, which creates an "instance" of a class named *CN*.

⌈**new** *CN*⌉ ≡
⌈**{ object** *I* **inherits** *CN* **}**⌉^{I}, where *I* is a fresh identifier (not free in *CN*).

One can use the following expression sugar to make anonymous concrete objects with a given generic function attribute (like `lambda` in Scheme).

⌈**anon** *GF*⌉ ≡
⌈**{ object** *I* **inherits** GenericFun_rep *I* **has** *GF* **}**⌉^{I}, where *I* is a fresh identifier.

With anonymous generic functions, one can desugar simultaneous and sequential eagerly-evaluated **let**-expressions. (These work for all objects that inherit from `any`.)

⌈**let** I_1 **=** E_1**,** …**,** I_n **=** E_n **in** E_0⌉ ≡
⌈**(anon method(**I_1**,**…**,**I_n**){**E_0**})(**E_1**,**…**,**E_n**)**⌉
⌈**let** I_1 **=** E_1**;** …**;** I_n **=** E_n **in** E_0⌉ ≡
⌈**let** I_1 **=** E_1 **in** … **let** I_n **=** E_n **in** E_0⌉

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.

⌈**[** *B* **]**⌉ ≡
⌈**anon method() {** *B* **}**⌉

2.4 Some Small Examples

In this section we show some small examples of datatypes programmed in BeCecil.

2.4.8 The Untyped Standard Prelude

To make the examples more interesting, the following are assumed as the prelude for example programs throughout this section, and throughout Appendix A and Appendix B. The declarations in this "untyped standard prelude" include that of an object `nothing`, which can also act as a generic function with no arguments. This object is used as the return value for procedures that do not wish to return anything. We use the trick of making this a generic function, so that it can be passed to higher-order functions, such as `if`. The standard prelude also declares some objects to be used as ancestors, and integer and floating point literals, such as 1, 2, 3, and 4.7. In various examples we will also assume some generic functions with methods that operate on numbers, such as `plus`, `less`, `leq`, etc.

```
-- the "untyped standard prelude"
object nothing                          -- used for methods that do not want to return anything
nothing has method() { nothing }

object any inherits nothing             -- superclass of all other classes
object GenericFun_rep inherits any

object number_rep inherits any
object int_rep inherits number_rep
object float_rep inherits number_rep
object 1 inherits int_rep               -- numeric literals are considered to be identifiers
object 2 inherits int_rep
object 3 inherits int_rep
   ...
object 4.7 inherits float_rep
   ...
```

2.4.9 Boolean

The Booleans can be coded much as in Smalltalk [Goldberg & Robson 83]. In coding this example we adopt the position of only hiding storage tables for objects. Thus one cannot guarantee that there are only two Boolean objects in a program, but allowing all classes to be subclassed is standard for OO languages; for example in Smalltalk one can make new subclasses of Boolean. (On the other hand, Java [Gosling *et al*. 96] and Dylan have mechanisms that can prevent subclassing.) This point of view has the advantage of allowing clients to specialize on `true` and `false`. Having `true` and `false` also be thunks that return themselves is useful when one does not need short-circuit evaluation; see the `equal` method for `interval_rep` in Section 2.4.14.

```
object boolean_rep inherits any
gf if     -- implemented by subclasses
gf not    -- also implemented by subclasses
gf equal  -- implemented below
fun ifTrue(b@boolean_rep, c@GenericFun_rep) { if(b, c, nothing) }
fun and(b@boolean_rep, c@GenericFun_rep) { if(b, c, false) }
fun or(b@boolean_rep, c@GenericFun_rep) { if(b, true, c) }
object true inherits boolean_rep
true has method() { true } -- useful when a thunk that returns true is desired
if has method(b@true, c@GenericFun_rep, ignored@GenericFun_rep) { c() }
not has method(b@true) { false }
object false inherits boolean_rep
```

```
false has method() { false } -- useful when a thunk that returns false is desired
if has method(b@false, ignored@GenericFun_rep, a@GenericFun_rep) { a() }
not has method(b@false) { true }
equal has method(x@boolean_rep, y@boolean_rep) { false }
equal has method(x@true, y@true) { true }
equal has method(x@false, y@false) { true }
```

2.4.10 collection

The following is an abstract class for collections. It gives an implementation of `length` that can be inherited by concrete objects that implement `do`.

```
object collection_rep inherits any
gf isEmpty    -- to be implemented by subclasses
gf do         -- to be implemented by subclasses
fun length(c@collection_rep) {
   var res := 0
   do(c, anon method(x) {res() := plus(res(), 1); nothing});
   res()
}
```

2.4.11 list

The following, and the next several examples below, give an implementation of lists, *a la* Cook [Cook 90]. That is, we use two abstract classes, `list_rep` and `nonempty_rep`, a concrete object, `nil`, and a concrete class `cons_rep`. The abstract class `list_rep` defines the protocol for accessing the elements of a list, and gives a general implementation of `do` that can be inherited by concrete objects.

```
object list_rep inherits collection_rep
gf head       -- to be implemented by subclasses
gf tail       -- to be implemented by subclasses
do has method(c@list_rep, b@GenericFun_rep) {
   if(isEmpty(c), nothing, [b(head(c)); do(tail(c), b)])
}
```

2.4.12 nil

Since BeCecil does not distinguish between classes and objects, the empty list can be implemented directly as a concrete object that inherits from `list_rep`.

```
object nil inherits list_rep
isEmpty has method(n@nil) { true }
tail has method(n@nil) { nil }      -- or one could use the trick below...
head has method(n@nil) { head(n) } -- loop forever!
```

2.4.13 nonempty lists and cons

The following gives the code for `nonempty_rep` and `cons_rep`. We use two classes so that one can inherit the `isEmpty` method from `nonempty_rep` more easily (in other nonempty list representations that do not want to inherit from `cons_rep`). The "fields" of `cons_rep` are hidden. The hiding of the fields makes `cons_rep` like a class with private instance variables, with private meaning the same as in C++. (It is unclear how to support something like the "protected" notion of C++.) The object `default_list_elem` is used for the default value of list elements. We use an initialization method to allow subclasses of `cons_rep` to initialize the hidden fields.

```
object nonempty_rep inherits list_rep
isEmpty has method(l@nonempty_rep) { false }
```

```
object default_list_elem inherits any

object cons_rep inherits nonempty_rep
gf initialize -- generic initialization function
hide
   field hd of cons_rep := default_list_elem
   field tl of cons_rep := nil
in
   initialize has method(c@cons_rep, x, l@list_rep) {
      hd(c) := x;
      tl(c) := l;
      c
   }
   tail has method(l@cons_rep) { tl(l) }
   head has method(l@cons_rep) { hd(l) }
   tail has acceptor(l@cons_rep) := new_tail { tl(l) := new_tail }
   head has acceptor(l@cons_rep) := new_head { hd(l) := new_head }
end
fun cons(x, l@list_rep) { initialize(new cons_rep, x, l) }
```

The implementation of the `equal` generic function for lists is broken into three cases, using multimethod dispatch.

```
equal has method(x@list_rep, y@list_rep) { false }
equal has method(x@nil, y@nil) { true }
equal has method(x@nonempty_rep, y@nonempty_rep) {
   and(equal(hd(x), hd(y)), [equal(tl(x), tl(y))])
}
```

2.4.14 interval

The ability to add new implementations of a type without changing existing code is a hallmark of OO programming. Cook considered another example to show this, namely adding intervals of integers as another subclass of `nonempty_rep`. Notice how in our example below, the `equal` method is specialized to take advantage of knowledge of the representations when both lists are intervals, which results in a faster method than the default for nonempty lists given above. Another thing to notice about this example is the programming of the `assign` and `clone` methods; these are used in the next example below. Finally, notice that the coding of initialize maintains the invariant for all objects, *o*, that inherit from `interval_rep`, `lower`(*o*) ≤ `upper`(*o*).

```
object interval_rep inherits nonempty_rep
hide
   field lower of interval_rep := 0
   field upper of interval_rep := 0
in
   initialize has method(i@interval_rep, lb@int_rep, ub@int_rep) {
      lower(i) := min(lb, ub);
      upper(i) := max(lb, ub);
      i
   }
   gf assign -- like C++ assignment operator
   assign has method(l@interval_rep, r@interval_rep) {
      lower(l) := lower(r);
      upper(l) := upper(r);
      l
   }
   tail has method(l@interval_rep) {
```

```
        let lower_plus_1 = plus(lower(l), 1)
        in if(greater(lower_plus_1, upper(l)),
              [nil],
              [var res := clone(l)
               lower(res()) := lower_plus_1;
               res()
              ]);
    }
    head has method(l@interval_rep) { lower(l) }
    equal has method(x@interval_rep, y@interval_rep) {
       and(equal(lower(x), lower(y)), equal(upper(x), upper(y)))
    }
end
fun mkInterval(lb@int_rep, ub@int_rep) {
    initialize(new interval_rep, lb, ub)
}
gf clone
clone has method(r@interval_rep) { assign(new interval_rep, r) }
```

2.4.15 gray_interval

The following example, which builds on the `interval` and `Grayscale_rep` examples above, shows how to use directed actuals to inherit methods. Directed actuals are used all the methods except `mkGrayInterval` and `clone`. The directed actuals prevent recursion, much as `super` would in Smalltalk, or `interval_rep::equal` would in C++. By overriding `assign` and `clone`, the `tail` method is inherited without further rewriting.

```
object gray_interval_rep inherits interval_rep, Grayscale_rep
initialize has method(gi@gray_interval_rep, lb@int_rep, ub@int_rep,
                      intensity@float_rep) {
    initialize(gi@Grayscale_rep, intensity);
    initialize(gi@interval_rep, lb, ub)
}
assign has method(l@gray_interval_rep, r@gray_interval_rep) {
    initialize(l@Grayscale_rep, intensity(r));
    assign(l@interval_rep, r@interval_rep)
}
fun mkGrayInterval(lb@int_rep, ub@int_rep, intensity@float_rep) {
    initialize(new gray_interval_rep, lb, ub, intensity)
}
clone has method(gi@gray_interval_rep) {
    assign(new gray_interval_rep, gi)
}
equal has method(x@gray_interval_rep, y@gray_interval_rep) {
    and(equal(intensity(x), intensity(y)),
        equal(x@interval_rep, y@interval_rep))
}
```

2.4.16 array1

The following example shows how a storage table with two arguments can be used as a one-dimensional array. The class `array1_rep` implements one-dimensional arrays of floats. The function `while` used in this example was described in Section 2.2.4. Notice that the use of both an acceptor and a method for the generic function sub allows one to use a pleasing notation for array subscripting, while still allowing bounds checking. For example, one can write: `sub(myArray,3):=47.0`.

```
object array1_rep inherits collection_rep
```

```
hide
   object a_stor inherits any
   a_stor has storage(a@array1_rep, i@int_rep) := 0.0
   field max_i of array1_rep := 9
in
   initialize has method(a@array1_rep, n@int_rep) {
      max_i(a) := n;
      do(a, anon method(i){a_stor(a,i) := 0.0});
      a
   }
   isEmpty has method(a@array1_rep) {less(max_i(a), 0)}
   do has method(a@array1_rep, b@GenericFun_rep) {
      foreach(a, anon method(i){b(a_stor(a,i))})
   }
   fun foreach(a@array1_rep, b@GenericFun_rep) {
      var i := 0
      while([leq(i,max_i)], [b(i); i() := plus(i(), 1)])
   }
   gf sub -- short for "subscript"
   sub has method(a@array1_rep, i@int_rep) {
      ifTrue(and(leq(0,i), [leq(i, max_i(a))]),
             [a_stor(a, i)])
   }
   sub has acceptor(a@array1_rep, i@int_rep) := e {
      ifTrue(and(leq(0,i), [leq(i, max_i(a))]),
             [a_stor(a, i) := e; a])
   }
end
fun mkArray(n@int_rep) { initialize(new array1_rep, n) }
```

2.5 Block Structure, Extensions, and Customization in Nested Scopes

BeCecil is designed to enable objects to be extensible. In this section we discuss how methods and inheritance relationships can be extended, and how these extensions are effected by nested scoping. We also describe how, in a nested recursive declaration sequence, one can customize that recursive declaration sequence's view of generic functions.

The concept of "scope" is a bit tricky with hide declarations in BeCecil, hence we avoid that term except when making general statements about "nested scopes". There are two more precise concepts that we use instead. The first is the usual notion of the *visibility* of a declaration; this is the area of the program text in which the declaration has effect. We also speak of area of visibility of a name, meaning the area in which the declaration that introduced the name is visible. The second is the concept of an *contour* of an object declaration; this is the area of the program text in which one may use extension (`has`) declarations that add methods or storage tables to the object's generic function value. We abuse terminology and refer to the contour of a name, meaning the contour of the name's object declaration. Contours correspond to recursive declaration sequences in the syntax, because only within the recursive declaration sequence where an object is declared can it be extended.

The reason that an object can only be extended within the recursive declaration sequence in which it is declared is that otherwise an object might be extended in two different ways in different nested contours. Since the hidden declarations of a hide declaration form a recursive declaration sequence, this means that privacy is based on generic functions, not individual methods, in BeCecil. That is, one cannot extend the

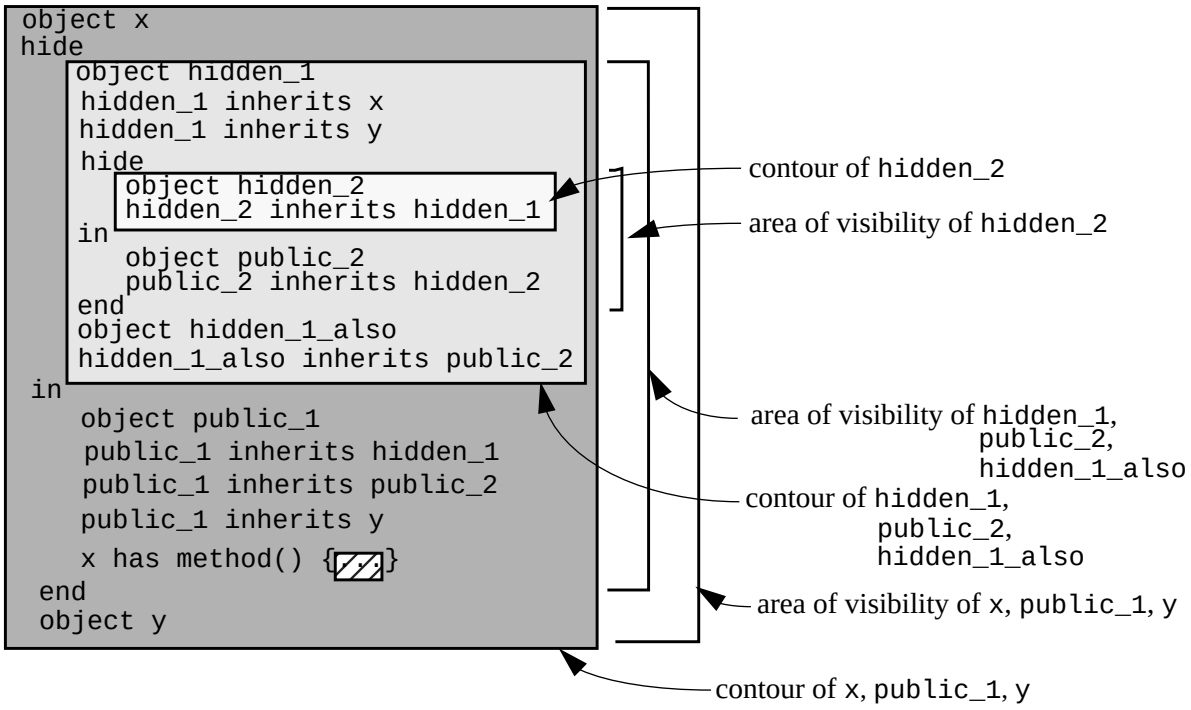

Figure 2-2: Illustration of the contours and areas of visibility of some declarations.

same generic function in both parts of a hide declaration. In this respect, BeCecil follows CLOS and Dylan, which also base privacy on generic functions.

As an example and further explanation of these concepts, consider Figure 2-2. In this figure there are two hide declarations within an outer recursive declaration sequence. The visibility of hidden declarations in a hide is not surprising. For example, `hidden_2` is visible within the entire hide declaration in which it is declared, including the public (non-hidden) part. The public declarations of a hide declaration are visible within all of the surrounding recursive declaration sequence. Thus, a declaration need only be put in the public part of a hide declaration if it needs to see the hidden declarations. To simplify things a bit, neither set of declarations in a hide declaration may shadow the other. Since recursive declaration sequences allow mutual recursion, `public_1` is visible within the outermost recursive declaration sequence; for example, it is visible where the declaration of object `x` resides. Since in this example no names are redeclared, there are no holes in the visibility of any of the declarations in Figure 2-2.

In Figure 2-2, one can extend `hidden_2` only within the recursive declaration sequence in which it is declared. The contour of this recursive declaration sequence is the small box (▭) that surrounds its declaration. Although `hidden_2` is visible in the public part of that hide declaration, it cannot be extended there, because that is not part of the same recursive declaration sequence. The block that forms the body of the method for `x` at the bottom of the outermost hide declaration in the figure (▨) also is not part of the recursive declaration sequence in which `x` is declared, and thus `x` cannot be extended there.

However, BeCecil does provide mechanisms, based on the λ&-calculus, for customizing generic functions within a nested contour. The identifier and combination generic function attributes allow one to use the generic function value of another object (perhaps declared in a surrounding contour), when customizing a nested contour's view of a generic function. For example, suppose one wants to debug the `equal` method by calling it with several different inputs, and that one wants to print a message before and after it is called with arguments that inherit from `interval_rep`. Then one could write the following, which defines a

generic function `equal2`, which has all the methods of `equal`, except that the method specialized on two `interval_rep` arguments is replaced by the debugging version. (Note that recursive calls from within the methods of `equal` are not sent to `equal2`, and would not be even if the name were `equal`, because with static scoping the name `equal` in such methods refers to the generic function defined in the outer contour.)

```
fun debug_equal() {
   gf equal2
   equal2 has equal & method(l1@interval_rep, l2@interval_rep) {
                        print("calling equal with args ", l1, l2);
                        var res := equal(l1, l2);
                        print("the call returned ", res());
                        res();
                      }
   ... equal2(...) ...
}
```

The block structure of BeCecil also effects inheritance. As as start towards explaining this interaction, it is important to first look at the effect of an inheritance declaration, disregarding block structure for the moment. The only dynamic effect of an inheritance declaration is on the applicability and relative specificity of methods, storage tables, and acceptors for objects that can see it. To ease this kind of discussion, let us call something that is either a method, storage table, or acceptor a *case*. Specificity of cases is based on the pointwise extension of an inheritance relation to the specializers of cases. That is, a case c_1 is as specific as c_2 if they have the same number of formals, and if the specializers of c_1 each inherit (directly or indirectly) from the corresponding specializer of c_2. We say that c_1 *overrides* c_2, or c_1 *is more specific than* c_2, if c_1 is as specific as c_2 but their specializers are different. Let us also call a case that is either a method or a storage table an *invocable*. Recall that, among a generic function's invocables, the most specific one that is applicable to the actual arguments is used. For example, in the following block, the last method is called, because it is more specific than the others.

```
equal has method(n1@number_rep, n2@int_rep) {...}
equal has method(n1@int_rep, n2@number_rep) {...}
equal has method(n1@int_rep, n2@int_rep) {...} -- this one is called
equal(7,7)
```

A generic function must use some inheritance relation to decide the applicability and specificity of its cases. What inheritance relation should it use? If a programming language only allows inheritance relations to be defined at the top-level, then there is no problem, but in BeCecil, objects and their inheritance relations may be defined in several places. We investigated several alternatives (see Section B.8 for a discussion of them), but to reason statically about a program, and in particular to permit static type checking, the inheritance relation used cannot vary dynamically. Thus the inheritance relation used is the one that is defined in the recursive declaration sequence (and surrounding contours) where the generic function is defined. For this reason, each object contains the this inheritance relation.

The inheritance relation recorded within an object also serves another purpose, in that it defines the object's ancestry. The problem solved by this is that dynamically created objects do not appear in the inheritance relations of generic functions that are not declared in recursive declaration sequences where the object's declaration is visible. For example, consider the following code. The inheritance relation that is recorded in the object `push` says that `Stack_rep` inherits from `any`, but says nothing about objects created within the body of `f`. Of course, it cannot, since the body of `f` is a nested contour. But then how is the call to `push` in the body of `f` supposed to work? It would seem that the newly created object does not, according to the inheritance relation recorded for `push`, inherit from `Stack_rep`. This is true, but the object `myStack`

does know that it inherits from `Stack_rep` (and from `any`). So the application works if some ancestor of `myStack` is known to be `Stack_rep`.

```
object Stack_rep inherits any
gf push
push has method(s@Stack_rep, j@int_rep) {...}
fun f() {
   object myStack inherits Stack_rep
   push(myStack, 27)
}
f()
```

Now consider a variation on the above example where the generic function and one of the arguments come from different contours. In the following, the inheritance relationship between `Stack_rep` and `collection_rep` is only known within the function `nested`. Inside `nested`, the generic function `size` is declared and assigned to the variable `g`. The generic function `size` has an inheritance relation that knows that `Stack_rep` inherits from `collection_rep`. The variable `x` is assigned an object that inherits from `Stack_rep`, but which does not have `collection_rep` as an ancestor. The application at the end works, however, because the object stored in `x` has an ancestor that is known to inherit from the specializer of the generic function stored in `g` (according to its inheritance relation).

```
object collection_rep inherits any
object Stack_rep inherits any
var g := anon method(x@collection_rep) {...}
var x := new Stack_rep
fun nested() {
   Stack_rep inherits collection_rep
   gf size
   size has method(s@collection_rep) {...}
   g() := size
}
fun nested_also() { x() := new Stack_rep }
nested();
nested_also();
g()(x()) -- works
```

So the general rule is that a case is applicable at some formal argument position if some ancestor of the actual argument is known to inherit from the formal's specializer, according to the inheritance relation recorded in the generic function. This is sensible, because it allows one to understand a generic function (like `size` in the above example) in the context in which it was declared. It then behaves as it was understood no matter where it is used. In this sense the semantics for inheritance is static.

2.6 Procedural and ADT Patterns

In BeCecil, one can program not only OO patterns, but also procedural and ADT patterns. Consider first argument dispatch. BeCecil has dynamic multiple dispatch as its basic application mechanism; that is, the particular invocable in a generic function that is called is based on the run-time ancestry of all the actual arguments. However, if one has an object, such as BeCecil's `any`, that is the ancestor of (almost) all others, then by specializing on `any`, one is effectively not specializing on that argument position. Thus in BeCecil, one can program as in a single-dispatch language by always specializing on just the first argument of a case, or one can program in a procedural style by never specializing on any arguments. The following are thus examples in singly-dispatched OO and procedural styles (respectively).

```
single_oo_method has method(first@myClass, snd@any, thd@any) {...}
proc_method has method(first@any, second@any, thd@any) {...}
```

As Cook points out [Cook 90], one characteristic of ADT languages, such as Ada 83 [Ada 83], CLU [Liskov *et al.* 81], or Standard ML, is that they completely control the implementation of a particular type of object. That is, they are able to guarantee behavioral properties of objects of a given type, because they control the ability to create and modify objects of that type. However, usually objects in such languages are not extensible (i.e., there is no concept of inheritance). The information hiding used in our previous (OO) examples always involved hiding generic functions (especially for storage tables). This allows one to change data structures at will, and also to enforce representation invariants. It thus gives information hiding, but does not prevent impersonation [Morris 73], since a client can always create a subclass of a given class and override all the methods that it would otherwise inherit. To see this, consider the following code, which declares an object that inherits from `Grayscale_rep` (of Section 2.2.1). If the specifications of the methods `initialize`, `intensity`, and `paint` include a requirement that they do not loop forever, then their specifications are not satisfied when their first argument is `bad`, even though `bad` inherits from `Grayscale_rep`. Thus, behavioral properties cannot be guaranteed to hold for all objects that inherit from `Grayscale_rep`.

```
object bad inherits Grayscale_rep
initialize has method(b@bad, f@float_rep) { initialize(b,f) } -- loop forever!
intensity has method(b@bad) { intensity(b) }                  -- loop forever!
paint has method(b@bad, r@Region_rep) { paint(b,r) }          -- loop forever!
```

Languages like Ada 83 or CLU are able to guarantee such behavioral properties because they are not extensible; that is, in such a language one can implement a type and prohibit a client from creating instances, except by using some defined interface. BeCecil also allows this style of programming. If the name of a class is hidden, then clients cannot create instances directly. Clients also cannot override methods for subclasses of the class, because clients will not be able to declare subclasses at all. This style of coding trades extensibility for guaranteed control over instances. For example, the following shows how, by hiding the name `GS2_rep`, one can make a class that is like `Grayscale_rep`, but prevents imposters. Note that the acceptor for the generic function `intensity` has been moved inside the hide declaration, since otherwise it could not specialize on `GS2_rep`.

```
hide
   -- private declarations
   object GS2_rep
   field scale of GS2_rep := 0  -- sugared version of the previous declaration of scale
in
   -- public declarations
   GS2_rep inherits Color_rep
   initialize has method(c@GS2_rep, intensity@float_rep) {
      scale(c) := truncate(multiply(min(max(0.0, intensity), 1.0), 255));
      c
   }
   gf mkGS2
   mkGS2 has method(intensity@float_rep) {
      initialize(new GS2_rep, intensity)
   }
   intensity has method(c@GS2_rep) {
      divide(mkFloat(scale(c)), 255.0)
   }
   intensity has acceptor(c@GS2_rep) := v { initialize(c, v) }
   paint has method(c@GS2_rep, r@Region_rep) { ... }
end
```

The trade-off between extensibility and such control over subclasses and creation of instances is perhaps not absolute. One can imagine a language in which subclasses could not override methods of their

superclasses, or a more complex language than BeCecil in which the code for a subclass's methods would have to provably satisfy the specification of the superclass. But this trade-off does seem fundamental in languages where subclasses have unlimited ability to change inherited behavior and behavioral specifications are not an enforced part of the program text.

In summary, BeCecil allows one to program in an ADT style that gives every bit as much control over instances of a class as one would have in a language like Ada 83 or CLU. When doing so, one gives up extensibility, but that is no worse than in such ADT languages. Indeed, BeCecil is more flexible, because within a hide declaration, one can use subclassing on the hidden class, even though clients cannot do so. Finally, in BeCecil one can program in either an ADT style, or an OO style, and both styles can be used in the same program.

3 Syntax, Overview, Sugars, and Examples for the BeCecil Type System

In this section we give an overview of the BeCecil type system. We first present the syntax of BeCecil with its type annotations, give a brief overview of the main ideas, then describe some sugars and small examples. Finally we describe some subtle points of the type system. All this should give the reader a feel for the type system before the details are presented in Appendix C [Chambers & Leavens 96].

3.1 Syntax

The syntax of BeCecil with types is given in Figure 3-1. The syntax for generic function attributes has type annotations added to formal parameters and a type annotation for return types. Also there are several new declaration forms, and a syntax for type expressions.

3.2 Brief Overview

A *type error* in BeCecil occurs when a program applies a generic function to a tuple of actual arguments, and the generic function either has no case that is applicable to the actuals, or has more than one applicable case, but not a unique, most-specific one [Chambers & Leavens 95]. If the first kind of error ("message not understood") can occur, the generic function is *incomplete*. If the second kind of error ("message ambiguous") can occur, the generic function is *inconsistent*.

The BeCecil type system is designed to statically prevent such type errors. The philosophy behind the type system is to leave the dynamic semantics of the language unchanged. To this end, the central notions of the type system are found only in the static semantics, and have no run-time effect. To allow additional flexibility, classes are not considered to be types, and inheritance is independent of subtyping. Type information in BeCecil is largely declared, not inferred, although the generic function types of objects are inferred from the information given as annotations to formal arguments and return types of method and storage table attributes. To keep the type system simple, there is no parametric polymorphism; thus the only kind of polymorphism supported is subtype polymorphism.

3.2.1 Declarations for the Type System

There are four new declarations in the typed version of BeCecil. The first of these is the type declaration, which declares names for user-defined types. For example, the following declares a new type named `Point`.

```
type Point
```

Since user-defined type names are intended to describe ADTs, they are considered to have implicit behavioral specifications. Hence two such declarations always produce different types, even if they declare the same name in different contours. To achieve this, each declared type name is given a unique identity within any contour in which it is visible.

Since user-defined type names have implicit behavioral specifications, subtyping among type names is declared, not inferred. For example, the following declares that `Point` is a subtype of `Top` (used as the supertype of all user-defined types in our examples), and that `ColorPoint` is a subtype of `Point` and `Color`.

```
Point subtypes Top
ColorPoint subtypes Point
ColorPoint subtypes Color
```

The subtype relation among types is the extension of this declared direct subtyping relation to a reflexive and transitive relation. BeCecil does not require that this relation be antisymmetric.

P	::=	*RDS* **;** *B*	*Program*
RDS	::=	*D**	*Recursive-Declaration-Sequence*
B	::=	*RDS E*	*Block*
D	::=		*Declaration*
		object *I*	*object: allocates a new object named I*
	|	CN_1 **inherits** CN_2	*inheritance: CN_1 directly inherits from CN_2*
	|	*CN* **has** *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**
	|	**type** *I*	*type: new named type*
	|	*TN* **subtypes** *T*	*subtype: TN is a subtype of T*
	|	*CN* **conforms** *T*	*conformance: CN has the type T*
GF	::=		*Generic-Function attributes*
		I	*identifier: the methods and tables of I*
	|	**method(***F****):** *T* **{***B***}**	*method: like λ, a block abstraction*
	|	**storage(***F****) :=** *E***:***T*	*storage table: new table initialized to value of E*
	|	**acceptor(***F****) :=** *I***:***T* **{***B***}**	*acceptor: can process an assignment with value I*
	|	GF_1 **&** GF_2	*combination: GF_2 favored over GF_1*
F	::=	*I* **@** *CN* **:** *T*	*Formal argument*
T	::=		*Type expression*
		TN	*type name: a user-defined type*
	|	**(***T****) ->** T_{n+1}	*invocable: with arguments T*, returns T_{n+1}*
	|	**(***T****) :=** T_{n+1}	*assignable: with key arguments T* and value T_{n+1}*
	|	**exact{***ET****}**	*exact: all information about an object's generic function*
	|	T_1 **&** T_2	*conjunction: glb of types*
	|	T_1 **|** T_2	*disjunction: lub of types*
TN	::=	*I*	*Type-Name*
ET	::=		*Exact-Type for a generic function*
		(*CT****) ->** T_{n+1}	*exact invocable: with arguments CT*, returns T_{n+1}*
	|	**(***CT****) :=** T_{n+1}	*exact assignable: with key arguments CT* and value T_{n+1}*
CT	::=	*CN* **:** *T*	*Class-and-Type*
CN	::=	*I*	*Class-Name*
E	::=		*Expression*
		I	*class name: evaluates to the object denoted by CN*
	|	E_0 **(***AA****)**	*application: evaluates operator then operands left-to-right*
	|	E_0 **(***AA****) :=** E_{n+1}	*assignment: with key AA* and value E_{n+1}*
	|	E_1 **;** E_2	*sequence: evaluates E_1, then E_2*
AA	::=		*Actual-Argument*
		E	*undirected: applicabile if inherits from specializer*
	|	*E* **@***CN**	*directed: applicable if also a CN_i inherits from specializer*

Figure 3-1: Syntax for BeCecil with type checking aspects added. The nonterminal *I* is an identifier.

The connection between the world of objects and the world of types is established by a conformance declaration. Conformance is a relation between objects and types, which says that the object has the given type. For example the following declaration says that the object `Point_rep` conforms to the type `Point`. This declaration makes `Point_rep` a prototype (or concrete object): it can be used in any place that any other object that conforms to `Point` can be used.

```
Point_rep conforms Point
```

Conformance of formal parameters to types, as well as conformance of the results of methods to types, is asserted with the usual colon (:) notation. For example, the following is the extension declaration for the `mkPoint` method.

```
mkPoint has method(i@int_rep:int, j@int_rep:int): Point { ... }
```

Several conformance declarations may be given for the same object. Thus an object may have several types. For example, the following declarations say that `myColorPoint` conforms both to the type `Point`, and the type `Color`.

```
myColorPoint conforms Point
myColorPoint conforms Color
```

An object may conform to a type either directly or indirectly. Direct conformance relationships are declared in the program text, as above. For example, `myColorPoint` directly conforms to `Point`. Indirect conformance relationships take subtyping into account. The phrase *CN conforms to T* means that either *CN* directly conforms to *T*, or that *CN* directly conforms to some type T_2, and T_2 is a subtype of *T*. For example, `myColorPoint` conforms to `Top`. Note that if CN_1 inherits from *CN*, and *CN* conforms to *T*, this does *not* mean that CN_1 conforms to *T*; inheritance relationships have nothing to do with conformance, and subclasses are not required to be subtypes, nor vice versa.

Not every object has to be declared to conform to a user-defined type. It is perfectly legitimate to give no conformance declarations for an object. Such an object is then *abstract*, in the sense that it cannot be used as an object in expressions.

The type expressions in BeCecil are inductively defined, with user-defined type names as the basis. There are three kinds of types that directly describe the generic function value of an object. The most familiar is the invocable type, which consists of a list of argument types, an arrow (`->`), and a return type. For example, the following declares that the generic function equal can be called with two point arguments, and returns a boolean.

```
equal conforms (Point, Point) -> boolean
```

The assignable type is written with a list of types that describe the key, an assignment symbol (`:=`), and a type that describes the value being assigned. For example, the following declares that the generic function `y` can process an assignment with a single key of type `Point`, and a value of type `int`.

```
y conforms (Point) := int
```

The final kind of type that describes the generic function value of an object is an exact type. This consists of the keyword `exact`, and a set of exact invocable and assignable types, each of which contains information about the class of their formal arguments. For example, the following declares a variable and explicitly gives its exact type.

```
gf myVar
myVar has storage() := 27:int
myVar conforms exact{()->int, ():=int}
```

Exact types contain information about each and every case for an object. Hence an object may only conform to one exact type. The exact types of all objects declared in a program are inferred by BeCecil from the type

annotations given for method, and storage, and acceptor attributes of extension declarations. This helps make the language less error-prone, since conformance declarations to exact types, such as the one above, are redundant. It also helps make the language more extensible, since writing down an exact type for an object means that it cannot be extended with any more cases. However, it is sometimes useful to write down an exact type for a formal argument, since an identifier can only be used as a generic function attribute if it conforms to an exact type.

The type system also has conjunction (greatest-lower-bound) types, written with an ampersand (&), and disjunctive types (least-upper-bound) types, written with a vertical bar (|). The conjunction S&T is the least specific type that subtypes both S and T. This type is different than any user-defined type if S and T are not ordered by the subtyping relation. One use of a conjunctive type is collapsing several conformance declarations into one. For example, the first two declarations below are equivalent to the third.

```
x conforms (Point) -> int
x conforms (Point) := int
x conforms (Point) -> int & (Point) := int
```

The disjunction S|T is the most specific type that is a supertype of both S and T.

While subtyping for user-defined type names is declared, subtyping for generic function types is structural. Exact types have no interesting subtypes (because there is no way that the information could be more exact). However, an exact type can be translated into a supertype that is a conjunction of inexact generic function types. For example, suppose that the type system has inferred that the exact type of x is the following.

```
exact{(Point_rep:Point)->int, (Point_rep:Point):=int}
```

Then the exact type information for x can be forgotten by passing to the following supertype.

```
(Point) -> int & (Point) := int
```

Because exact types for generic functions are inferred, and because they have inexact types as supertypes, it is not normally necessary to declare that generic functions conform to inexact generic function types. However, such declarations do have their uses, in particular, when declaring the types of abstract (deferred) methods or storage tables that are to be implemented for all objects that conform to a given type. For example, the type of the generic function `if` would be declared as follows, enabling the type system to check that it is implemented for all objects that conform to the type `boolean`. (Such conformance declarations are similar to the `signature` declarations in Cecil [Chambers 95, Chambers & Leavens 95].)

```
if conforms (boolean, ()->Top, ()->Top) -> Top
```

As a complete example, we show the typed version of the `Point` example from Section 2.2 (without the use of any syntactic sugars).

```
type Point
Point subtypes Top
object Point_rep
Point_rep inherits any
Point_rep conforms Point -- Point_rep is a prototype
object equal
equal inherits GenericFun_rep
equal has method(p1@Point_rep:Point, p2@Point_rep:Point): boolean {
   and(equal(x(p1), x(p2)), equal(y(p1), y(p2)))
}
object x
x inherits GenericFun_rep
x has storage(p@Point_rep:Point) := 0:int
object y
```

```
y inherits GenericFun_rep
y has storage(p@Point_rep:Point) := 0:int
object initialize
inititalize inherits GenericFun_rep
initialize has method(p@Point_rep:Point,
                      i@int_rep:int, j@int_rep:int): Point {
   x(p) := i;
   y(p) := j;
   p
}
object mkPoint
mkPoint inherits GenericFun_rep
mkPoint has method(x@int_rep:int, y@int_rep:int): Point{
   object res
   res inherits Point_rep
   res conforms Point
   initialize(res, x, y)
}
```

The way that the type system infers generic function types and checks applications only allows one to pass actual arguments to a case if they conform to its declared argument types. Therefore each formal's specializer and declared type independently constrain what kinds of arguments can be passed to it. Thus a formal's specializer need not conform to its declared type. For example, one can write the following, even though `any` does not conform to `int`.

```
object double
double inherits GenericFun_rep
double has method(x@any:int):int { plus(x, x) }
```

3.2.2 Client-side and Implementation-side Checks

Type checking can be divided into two parts [Chambers & Leavens 95]: *client-side* checks and *implementation-side* checks. Client-side checks are principally that each generic function application (and assignment) is type-correct, by comparing the types of the actuals and the type of the generic function. Other such checks are that the type of the body of a method conforms to the declared result type of the method. Such type checks are straightforward, except for applications that use directed actuals. For directed actuals, the generic function must be statically known, so that its exact type and inheritance relation can be used to check that all possible tuples of objects, which match the actual argument types, will find a unique most-specific method.

Implementation side checks are that for each declared object, for each invocable type `(`T*`)->`T_r to which that object conforms, and for each tuple of argument objects that conforms to T*, the object's set of invocables has at least one (completeness) and no more than one (consistency) invocable that is applicable to the actuals; furthermore, the tuple of arguments must conform to the types of this invocable's formals and its result type must be a subtype of the result type of T_r (conformance). A similar check must also be made for each object and each assignable type, `(`T*`):=`T_r, to which each object conforms.

In the following example consider implementation-side checks for the declared objects `bar` and `f`. The object `bar` does not conform to any invocable or assignable type, and thus trivially passes the implementation-side checks. The object `f` conforms to the invocable type, `(Bar)->int`. There is just one tuple of objects, (`bar`), that conforms to the tuple of argument types, and it is such that the single method given for `f` is applicable. Furthermore, this method has the required return type. Hence this example satisfies the implementation-side checks.

```
type Bar
```

```
object bar
bar inherits any
bar conforms Bar
object f
f inherits GenericFun_rep
f has method(x@bar:Bar): int { 3 }
```

It is interesting to see how implementation-side checking effectively checks that each subtype has appropriate methods to satisfy the usual rules for structural subtyping. Suppose the following declarations are added to the above example.

```
type Foo
object foo
foo inherits any
foo conforms Foo
Foo subtypes Bar     -- really?
```

Does the example still pass the implementation-side type checks with these additional declarations? No, because now there is a tuple of objects, (`foo`), that conforms to the tuple of argument types for `f`, but for which there is no applicable method. However, if `foo` were declared to inherit from `bar`, then the method would be applicable, so the checks would be passed. Alternatively, one could fix the example by declaring another method for f, such as the one below.

```
f has method(y@foo:Bar): int { 4 }       -- one way to fix the above example
```

Another interesting aspect of the implementation-side checks is how they enforce a "monotonicity" [Reynolds 80], "regularity" [Goguen & Meseguer 87], or "consistency" [Agrawal *et al.* 91] condition on the types of the methods in a generic function. This condition says that if a generic function conforms to two types, $(T_1,...,T_n)\text{->}T_r$ and $(S_1,...,S_n)\text{->}S_r$, and if for each i, S_i is a subtype of T_i, then S_r must be a subtype of T_r . To see how this is enforced, suppose that, for each i, S_i is a subtype of T_i. Then $(T_1,...,T_n)\text{->}T_r$ is a subtype of $(S_1,...,S_n)\text{->}T_r$ by the usual contravariant subtyping rule for function types [Cardelli 88]. Therefore, if an object conforms to $(T_1,...,T_n)\text{->}T_r$ it must also conform to $(S_1,...,S_n)\text{->}T_r$. Thus methods that can be called with a tuple of arguments of type $(S_1,...,S_n)$ must return a result that conforms to T_r . So any method that is declared to take a tuple of arguments of type $(S_1,...,S_n)$ must have a declared result type, S_r , that is a subtype of T_r .

The practical consequence of this condition is that programmers have to be careful to not make the result types of methods too specific, relative to their declared argument types. For example, consider the following declarations.

```
object negate
negate inherits GenericFun_rep
negate has method(x@int_rep:number): int {...}      -- these do not type check
negate has method(x@float_rep:number): float {...}
```

With these declarations, the code does not pass the implementation-side type checks, because `negate` conforms to the invocable type `(number)->int`, and the last method above does not return an `int` when passed a `number`. The code would pass the checks if the two methods both had a declared return type of `number`. The code would also pass the type checks if both the methods had declared more specific argument types (`int` and `float`).

A generic function can also fail to pass the implementation-side checks if a more specific method uses a more general result type than another method that it specializes, as in the following example, where the second method overrides the first for arguments that inherit from `float_rep`. Again the code would type

check if both methods returned the same type; however, this example cannot be fixed by changing the second method's formal argument type to `float`.

```
object truncate
truncate inherits GenericFun_rep
truncate has method(x@number_rep:number): int {...}     -- these do not type check
truncate has method(x@float_rep:number): number {...}
```

3.3 Syntactic Sugars

The syntactic sugars for the type system add to the sugars of Section 2.3, except where they are noted as replacements.

3.3.3 Typed Declaration Sugars

The first declaration sugar allows one to declare a conformance relationship at the same time that one declares several inheritance relationships.

$\lceil CN_0$ **inherits** $CN_1,...,CN_n$ **conforms** $T \rceil \equiv$
$\quad \lceil CN_0$ **inherits** $CN_1,...,CN_n \rceil$
$\quad CN_0$ **conforms** T

Because of the convention we have been using of naming classes with names of the form `X_rep` and the corresponding type `X`, one often wishes, for example, to declare that an object inherits `X_rep` and conforms to `X` at the same time. This is particularly useful for declaring "instances." The following sugar is similar to one in Cecil that serves a similar purpose.

$\lceil$ **object** I **isa** $TN_1,...,TN_n \rceil \equiv$
$\quad \lceil$ **object** I **inherits** $TN_1\_rep, ..., TN_n\_rep$ **conforms** TN_1 **&** ... **&** $TN_n \rceil$

The following sugars makes it easier to declare subtype relationships.

$\lceil TN_0$ **subtypes** $TN_1,...,TN_n \rceil \equiv$
$\quad TN_0$ **subtypes** TN_1
$\quad$ ...
$\quad TN_0$ **subtypes** TN_n

$\lceil$ **type** I **subtypes** $TN_1,...,TN_n \rceil \equiv$
$\quad$ **type** I
$\quad \lceil I$ **subtypes** $TN_1,...,TN_n \rceil$

The following sugar makes it easier to declare abstract generic functions, by allowing one to declare the name and its type at the same time. This sugar relies on the sugar for declaring generic functions from Section 2.3.

$\lceil$ **gf** I **conforms** $T \rceil \equiv$
$\quad \lceil$ **gf** $I \rceil$
$\quad I$ **conforms** $\lceil T \rceil$

The following sugars are replacements for the sugars of Section 2.3. They add type information to the previous sugars.

$\lceil$ **var** I**:**T **:=** $E \rceil \equiv$
$\quad \lceil$ **gf** $I \rceil$
$\quad I$ **has storage() :=** $\lceil E \rceil$**:**$\lceil T \rceil$

⌈**fun** *I*(*F**):*T* **{***B***}**⌉ ≡
    ⌈**gf** *I*⌉
    I **has method(***F**):*T* **{***B***}**⌉
⌈**field** *I*:T_1 **of** *CN*:T_0 **:=** *E*⌉ ≡
    ⌈**gf** *I*⌉
    I **has storage(***x*@*CN*:⌈T_0⌉**) :=** ⌈*E*⌉:⌈T_1⌉, where *x* is a fresh identifier.

3.3.4 Typed Formal Argument Sugars

As in Cecil, it is convenient to be able to write the specializer and type information for formal arguments in an abbreviated form.

⌈*I* `@:`*TN*⌉ ≡
    I `@` *TN_rep* `:` *TN*

Recall that if the specializer is `any`, it can be omitted.

⌈*I* `:`*T*⌉ ≡
    I `@ any :` ⌈*T*⌉

The following allows one to not mention the specializer `GenericFun_rep` when a formal argument has an invocable type.

⌈*I* `@:(`*T**`)->` T_{n+1}⌉ ≡
    I `@ GenericFun_rep :` **(**⌈*T**⌉**)->**⌈T_{n+1}⌉

For example, we have the following expansion into BeCecil.

```
⌈ifTrue has method(b@:boolean, c@:()->Top): Top {...}⌉ ≡
    ifTrue has method(b@boolean_rep:boolean,
                      c@GenericFun_rep:()->Top): Top {...}
```

3.3.5 Typed Expression Sugars

The following sugar replaces the untyped version of the sugar for **new** given in Section 2.3.7.

⌈ **new** *CN*:*T*⌉ ≡
    ⌈**{ object** *I* **inherits** *CN* **conforms** *T*
      }^{I}⌉ where *I* is a fresh identifier

When creating a new object of a type named *TN* that is to inherit from a class named *TN_rep*, one can use the following sugar.

⌈ **new isa** *TN*⌉ ≡
    ⌈**new** *TN_rep*:*TN*⌉

The expression sugars for the two forms of `let` carry over into the typed version of BeCecil if one adds appropriate type annotations based on the minimal inferred types of the expressions involved. Similarly the expression sugar for Smalltalk-like blocks carries over by adding the return type `Top` to the desugared form given in Section 2.3.7.

3.4 Some Small Examples

In this section we show some small examples of BeCecil code with types. For purposes of giving examples, we assume that the following declarations are used as the standard prelude for all example programs from now on. We continue to assume various other generic functions that operate on numbers as well.

```
-- the "typed standard prelude"
type Top

type void subtypes Top                    -- type of methods that do not want to return anything
object nothing conforms void
nothing has method(): void { nothing }

object any inherits nothing    -- abstract superclass of all other classes
object GenericFun_rep inherits any

type number subtypes Top
type int subtypes number
type float subtypes number
object number_rep inherits any
object int_rep inherits number_rep
object float_rep inherits number_rep
object 1 isa int
object 2 isa int
object 3 isa int
   ...
object 4.7 isa float
   ...
```

3.4.6 Boolean

We add type declarations to the Booleans, from Section 3.4.6, as follows. The code has been changed in places to use the generic function `ifExp` instead of `if`. This is because `if` can return something of any type, and so is only useful as a statement (when one cares about type checking). The generic function `ifExp` works for booleans, but if it is to function as a true if-expression, then it must be extended for each type of result. However, these problems could easily be solved by adding parametric polymorphism to BeCecil's type system.

```
type boolean subtypes Top
object boolean_rep inherits any -- abstract class
-- the following 3 generic functions are to be implemented by conformers
gf if conforms (boolean, ()->Top, ()->Top) -> Top
gf ifExp conforms (boolean, ()->boolean, ()->boolean) -> boolean
gf not conforms (boolean) -> boolean
gf equal conforms (boolean, boolean) -> boolean  -- implemented below
fun ifTrue(b@:boolean, c@:()->Top): Top { if(b, c, nothing) }
fun and(b@:boolean, c@:()->boolean): boolean { ifExp(b, c(), false) }
fun or(b@:boolean, c@:()->boolean): boolean { ifExp(b, true, c()) }

object true isa boolean
true has method(): boolean { true }
if has method(b@true:boolean, c@:()->Top, a@:()->Top): Top { c() }
ifExp has method(b@true:boolean, c@:()->boolean, a@:()->boolean): boolean {
   c()
}
not has method(b@true:boolean): boolean { false }

object false isa boolean
false has method():boolean { false }
if has method(b@false:boolean, c@:()->Top, a@:()->Top): Top { a() }
```

```
ifExp has method(b@false:boolean, c@:()->boolean, a@:()->boolean): boolean {
   a()
}
not has method(b@false:boolean): boolean { true }

equal has method(x@:boolean, y@:boolean): boolean { false }
equal has method(x@true:boolean, y@true:boolean): boolean{ true }
equal has method(x@false:boolean, y@false:boolean): boolean { true }
```

3.4.7 collection

The following example shows how to add type declarations to the collection type that was given previously. Notice that `collection_rep` does not itself conform to any type, and hence is an abstract class.

```
type collection subtypes Top
type collElem subtypes Top
object collection_rep inherits any -- abstract class
gf isEmpty conforms (collection) -> boolean       -- to be implemented by conformers
gf do conforms (collection, (collElem)->void) -> void  -- this one too
fun length(c@:collection): int {
   var res: int := 0
   do(c, anon method(x:collElem):Top {res() := plus(res(), 1)});
   res()
}
```

3.4.8 list

In the following, notice how `equal` is not declared as here, as it has been declared above. However, unlike the untyped version of this example, its type is noted here.

```
type list subtypes collection
type listElem subtypes collElem
object default_list_elem inherits any conforms listElem
object list_rep inherits collection_rep  -- abstract class
gf head conforms (list) -> listElem      -- to be implemented by conformers
gf tail conforms (list) -> list          -- to be implemented by conformers
equal conforms (list,list) -> boolean    -- to be implemented by conformers
do has method(c@:list, b@:(collElem)->void): void {
   if(isEmpty(c), nothing, [b(head(c)); do(tail(c), b)])
}
```

3.4.9 nil

We do not declare a subtype of `list` for the empty list, as there seems to be no good reason to do so. Notice how the non-sugared form of formal arguments is used in the code below.

```
object nil isa list
isEmpty has method (n@nil:collection):boolean { true }
tail has method(n@nil:list):list { nil }
head has method(n@nil:list):listElem { head(n) } -- loop forever!
```

3.4.10 nonempty lists

The following gives the code for `nonempty_rep`, with type information added.

```
object nonempty_rep inherits list_rep   -- abstract class
isEmpty has method(l@nonempty_rep:collection) { false }
```

3.4.11 cons

As with the empty list, we do not declare a subtype of `list` for lists implemented by `cons_rep`. We declare an initialization generic function to allow initialization of lists. Recall that the object default_list_elem is used as the default list element.

```
gf initialize conforms (list, listElem, list) -> void
object default_list_elem inherits any conforms listElem

object cons_rep inherits nonempty_rep conforms list
hide
   field hd:listElem of cons_rep:list := default_list_elem
   field tl:list of cons_rep:list := nil
in
   initialize has method(c@cons_rep:list, x:listElem, l:list): list {
      hd(c) := x;
      tl(c) := l;
      c
   }
   tail has method(l@cons_rep:list): list {tl(l)}
   head has method(l@cons_rep:list): listElem {hd(l)}
   tail has acceptor(l@cons_rep:list) := new_tail:list {
      tl(l) := new_tail
   }
   head has acceptor(l@cons_rep:list) := new_head:listElem {
      hd(l) := new_head
   }
end
fun cons(x:listElem, l:list): list {
   initialize(new cons_rep:list, x, l)
}
equal has method(x@:list, y@:list): boolean{false}
equal has method(x@nil:list, y@nil:list) {true}
equal has method(x@nonempty_rep:list, y@nonempty_rep:list): boolean {
   and(equal(hd(x), hd(y)), anon method():boolean{equal(tl(x), tl(y))})
}
```

Adding type information to the interval list example of Section 2.4 is left as an exercise for the reader.

3.4.12 link

Since BeCecil does not have parametric polymorphism, using the type list results in a loss of information about the type of the elements in the `list`, since when extracted they are only known to have the type `listElem`. Hence the following type is sometimes useful, as will be seen in the `alist` example below.

```
type link subtypes Top
object link_rep inherits any conforms link
object nil conforms link
hide
   field nxt:link of link_rep:link := nil
in
   initialize has method(l@:link, next@:link): link { nxt(l) := next; l }
   gf next
   next has method(l@:link): link { nxt(l) }
   next has method(l@:link) := v:link { nxt(l) := v }
end
```

3.4.13 alist

The following example, one way to implement association lists, shows some interesting uses of private inheritance and subtyping; in the hidden declarations, `alist_rep` is made to inherit from `link_rep` and to subtype `link`. Notice also that `nil` is also made to conform to the `alist` type.

```
type alist subtypes collection
type key subtypes Top
object default_key conforms key
type value subtypes collElem
object default_value conforms value
nil conforms alist
do has method(c@nil:alist, b@:(collElem)->void): void { nothing }
object alist_rep inherits collection_rep conforms alist
hide
   alist_rep isa link
   field ky:key of alist_rep_rep:alist := default_key
   field vl:value of alist_rep:alist := default_value
in
   initialize has method(al@:alist, k:key, v:value, next@:alist): alist {
      ky(al) := k;
      vl(al) := v;
      initialize(al@link_rep, next);
      al
   }
   isEmpty has method(al@:alist): boolean { false }
   do has method(c@:alist, b@:(collElem)->void): void {
      b(vl(al));
      do(next(al), b)
   }
   gf assoc
   assoc has method(k:key, al@nil:alist, v:value): value { v }
   assoc has method(k:key, al@:alist, v:value): value {
      var res: value := v
      if(equal(k, ky(al)),
         [res() := vl(al)],
         [res() := assoc(k, next(al), v)]);
      res()
   }
end
fun acons(k:key, v:value, next@:alist): alist {
   initialize(new alist_rep:alist, k, v, next)
}
```

3.4.14 Named Subtypes of Generic Function Types

In BeCecil, user-defined type names can be thought of as having implicit behavioral specifications attached. The following example shows how to declare a type that is supposed to represent integer-valued functions that square their arguments.

```
type squarer subtypes (int) -> int
```

One declares that a generic function conforms to this type explicitly, as follows. Note that if an object was declared to conform to `squarer` without having an integer-valued method, it would not pass the implementation-side checks.

```
gf square conforms squarer
square has method(x@:int): int { times(x, x) }
```

Although the type system does not check that `square` satisfies the implicit behavioral specification, it does propagate knowledge about what objects have been asserted to conform to type `squarer`. For example, the function `is_right_triangle`, defined below, requires its `f` argument to conform to squarer.

```
fun is_right_triangle(a@:int, b@:int, c@:int,
                      f@GenericFun_rep:squarer): boolean {
   equal(plus(f(a), f(b)), f(c))
}
```

For example, the following application would type check.

```
is_right_triangle(3,4,5,square)
```

3.5 Some Details

The type system of BeCecil deals with the following features not discussed in [Chambers & Leavens 95]:

- directed actual arguments,
- extensible generic functions,
- first-class generic functions,
- hide declarations, and
- nested contours with local inheritance, object, and extension declarations.

Directed actual arguments are handled by requiring applications with directed actual arguments to be to statically-known generic functions. This is in the spirit of OO languages, which do not allow one to extract arbitrary methods of objects and call them. The information that is used to type check such an application consists of the exact type information for the generic function and the inheritance relation that was closed with the generic function when it was created. This allows the type system to mimic the dynamic semantics.

For the purpose of extending a generic function, cases can be extracted from an object if the object's exact type is known (its identity is not required). This is the main reason for having exact types in the syntax of type expressions.

First-class generic functions are handled by including types for them in the language. The type checking rules are well-known and straightforward.

Hide declarations themselves cause no difficulty for the type system, other than the fact that they introduce nested contours, which do cause problems. We discuss these problems below.

Nested contours in BeCecil require careful attention if they are not to make the type system unsound, and if they are to be useful. To help keep the type system simple and understandable, the type system is designed so that it does not use any information from nested contours, other than their type correctness, when checking a recursive declaration sequence. To put this another way, the type system assumes that the only objects and inheritance relationships that it needs to consider are those declared either in the contour where the objects and generic function attributes it is checking are declared or in surrounding contours, but not those in nested contours. However, this assumption might make the type system unsound, because in a nested contour, one could declare that a new object conforms to an argument type:

- when the new object does not inherit from the set of objects that were assumed to conform to that type ("types acquiring new conformers"), thus causing incompleteness, or
- when the new object inherits from two objects that were assumed to conform to that type and were assumed to not have common descendents ("relating unrelated conformers"), causing inconsistency.

Our names for these problems are given in the parenthetical remarks in each of the above bullets. Note that the above problems, possibly in combination, are the only ways that type unsoundness might arise due to nested contours (assuming that the implementation-side checks for the surrounding contour are carried out correctly, and that client-side checking is done as usual). Thus the main difficulty in designing a sound type system for BeCecil is preventing these problems without rendering nested contours useless.

To illustrate the problem of types acquiring unknown conformers, consider the following (type incorrect) BeCecil block. In the block, the generic function `equal` seems to be completely implemented (if one considers only the top-level declarations), because there is an implementation for the only object that conforms to its argument type. However, the assumption that only objects that inherit from `First_T_rep` conform to the type `T` is falsified in the body of the function `nested`, because `Another_T_rep` is a object that conforms to `T`, and it does not inherit from `First_T_rep`.

```
type T
gf equal conforms (T, T) -> boolean
object First_T_rep inherits any conforms T
equal has method(p1@First_T_rep:T, p2@First_T_rep:T): boolean { true }
fun nested():T {
   object Another_T_rep inherits any conforms T
   Another_T_rep
}
equal(nested(), First_T_rep) -- not understood!
```

To illustrate the problem of relating unrelated conformers, consider the following (type incorrect) block. The top-level recursive declaration sequence is checked, as usual using information in that contour, so it seems that the generic function `equal` is implemented consistently: the argument will either inherit from `Origin` or `Red`, but not both, as far as that contour is concerned. But in the function `nested` at the end, the object `Red_Origin` does inherit from both objects. Thus the call to `equal`, although seemingly fine, results in a message ambiguous error.

```
type Comparable
gf equal conforms (Comparable, Comparable) -> boolean
type ImPoint subtypes Comparable -- points that are not necessarily mutable
object Origin inherits any conforms ImPoint
equal has method(p1@Origin:Comparable, p2@Origin:Comparable):boolean {
   true
}
type Color subtypes Comparable
object Red inherits any conforms Color
equal has method(p1@Red:Comparable, p2@Red:Comparable):boolean { true }
var myImPoint:ImPoint := Origin
fun nested(): void {
   object Red_Origin inherits Red, Origin conforms ImPoint & Color
   myImPoint() := Red_Origin;
   nothing
}
nested();
equal(myImPoint(), myImPoint()) -- ambiguous!
```

Our idea for making the type system sound, and thus turning these run-time errors into type errors, is to simply prevent both of these problems from happening. To do this, for each declared type name, *TN*, the type system tracks two pieces of information. To prevent the problem of types acquiring new conformers, it tracks the set of objects that are known to conform to *TN* in the recursive declaration sequence in which it was declared; this set is called the *conformers of TN*. To prevent the problem of related unrelated

conformers, it tracks the inheritance relation of the recursive declaration sequence in which *TN* was declared; this is called the *inheritance relation of TN*.

Like inheritance, conformance and subtyping are also scoped in BeCecil. We say that a relationship, such as that *CN* conforms to *TN*, is *visible* in an contour *S* if that relationship holds in the type context given by the declarations of *S* (and surrounding contours). A declaration of an object or type name is *visible* in a contour if it is not shadowed.

The restrictions needed for soundness in the presence of nested contours are that in every contour, *S*, the following must hold.

- Suppose that the relationship CN_1 conforms to *TN* is visible in *S*, and that the declaration `type` *TN* does not appear in *S*. Then there must be some object CN_2, such that CN_2 is a member of the set of conformers of *TN*, and the relationship CN_1 inherits from CN_2 is visible in *S*.
- Suppose that the relationship CN_1 conforms to *TN* is visible in *S*, and that the declaration `type` *TN* does not appear in *S*. Then for all objects CN_2 and CN_3, if both CN_2 and CN_3 are conformers of *TN*, and if both of the relationships CN_1 inherits from CN_2 and CN_1 inherits from CN_3 are visible in *S*, then according to the inheritance relation of *TN*, either CN_2 inherits from CN_3 or vice versa.

The first rule above can be further simplified by omitting the check that the declaration of the type name does not appear in the contour, and letting objects declared to conform in the same contour as the type name trivially satisfy the rule. However, the second rule cannot be simplified in this way, as otherwise multiple inheritance would be prohibited. In an algorithm implementing these rules, one would want to only check new conformance relationships, excluding those that were already present in a surrounding contour.

As an example of how these rules permit one to use inheritance in a nested contour, consider the following. The example is permitted by our rules, as the two ancestors of the object `floppy` declared within the nested contour `barnum`, `Elephant_rep` and `CircusPerf`, each conforms to a different type, and the two types are unrelated. If however, `CircusPerf` was declared as a subtype of `Elephant_rep` within `barnum`, then the nested contour would be illegal.

```
type Elephant subtypes Top
object Elephant_rep inherits any conforms Elephant
fun haul(e@:Elephant, thing:Top, distance@:int): void { ... }
type CircusPerf subtypes Top
object CircusPerf_rep inherits any conforms CircusPerf
fun space_for_act(cp@:CircusPerf): int { ... }
fun barnum(): void {
   type CircusElephant subtypes Elephant, CircusPerf
   object floppy inherits Elephant_rep, CircusPerf_rep
   floppy conforms CircusElephant
   object wagon inherits any conforms Top
   haul(floppy, wagon, space_for_act(floppy))
}
barnum()
```

4 Related Work

BeCecil bears a great deal of similarity to Cecil [Chambers 92, Chambers 95]. However, there are several differences from Cecil that are worth pointing out. The major differences in the dynamic semantics are the presence, in BeCecil, of the hide declaration and block structure. A more minor difference is the first-class nature of generic functions in BeCecil. In BeCecil, assignment and generic functions are more integrated than in Cecil, but the ability to replace a storage table with an acceptor and a method follows Cecil, in which a field named `f` is accessed through two methods: `f` and `set_f`. (CLOS, Dylan, Self, and other languages also present instance variables to clients as methods.)

The type system of BeCecil relies heavily on our previous work on type systems for multimethod languages [Chambers & Leavens 94, Chambers & Leavens 95]. The major extensions with respect to that work are the inclusion of block structure, hide declarations, and directed actual arguments (which allows inheritance of methods). Another difference is the absence, in BeCecil, of a notation for declaring that objects are abstract or concrete. There are also some more subtle differences in assumptions; for example, the type system of BeCecil does not assume that there is a type that is the supertype of all other types, or that there is a class that is the superclass of all other classes (although we have used such a type and class in our examples). However, Cecil itself has a type system with parametric polymorphism.

In the rest of this section we discuss the relation of BeCecil to other languages with multimethods, Abadi and Cardelli's imperative object calculus, and to other OO languages with block structure.

4.1 The λ&-Calculus

With respect to work on multi-method semantics, the most closely-related work is the λ&-calculus [Ghelli 91, Castagna *et al.* 92, Castagna *et al.* 95]. The λ&-calculus is a statically-typed foundational calculus for multimethod languages, from which BeCecil draws several ideas.

In some respects, the λ&-calculus is simpler than BeCecil, while in other respects BeCecil is simpler. For example, in the λ&-calculus, generic functions are applied differently than regular functions, but in BeCecil, there is only one kind of function. On the other hand, BeCecil allows new objects and new inheritance relationships to be declared, even in nested scopes. In the λ&-calculus, the classes (which are also types) and their inheritance/subtyping relationships are considered to be constants (predeclared).

Although, in BeCecil, & is used only in declarations as a way to combine generic function attributes, it performs a similar function to the & operator in the λ&-calculus. To make this point more clearly, we show how the combination operator & of the λ&-calculus can be defined as an expression sugar in BeCecil. As for attributes, this combination takes the cases of both expressions, but for those that clash (have the same specializers), only the cases of the second generic function's methods remain in the result. Note that the result of such an expression has an exact type in BeCecil, and so it can be used in further combination expressions, just as in the λ&-calculus.

$\lceil GF_1$ **&** $GF_2 \rceil \equiv$

$\lceil$ **{ object** I **inherits** GenericFun_rep
I **has** GF_1 **&** GF_2
} $\rceil^I$, where I is a fresh identifier.

The λ&-calculus also has a way to write a generic function with no methods. This is another sugar in BeCecil.

```
⌈empty⌉ ≡
    ⌈{ object I inherits GenericFun_rep
     }⌉^I
```

A major difference from BeCecil is that the λ&-calculus does not deal with mutation. In some sense the λ&-calculus is indifferent to the presence of operators such as assignment, being a calculus that is primarily concerned with generic functions and their invocation. However, the semantics of the λ&-calculus would have to change to allow for mutable storage. One way this could be done is shown in the $\lambda\&^{:=}$-calculus [Castagna 96b]. In this calculus there are locations, which have reference types (much as in ML). (The $\lambda\&^{:=}$-calculus also has conversion functions, which are used to create specializable locations; this permits the modeling of covariantly-specialized instance variables. It is unclear whether this can be modeled in BeCecil, but there seems no obvious way to do it, as the type system would need to be changed.)

The type system for BeCecil uses several ideas that are also used in type system of the λ&-calculus. Like the λ&-calculus, BeCecil does not allow for dynamic inheritance and its typing rules are based on a declared subtyping relation between atomic types. The emphasis on generic function typing, and the subtyping rules for generic functions, are adapted from the λ&-calculus.

It is interesting to compare the treatment of generic function types in BeCecil and the λ&-calculus. The λ&-calculus types for generic functions are most closely related to the exact types in BeCecil. However, the λ&-calculus does not permit a generic function type to have supertypes that are single arrow types (the types of non-overloaded functions), because, in the λ&-calculus, generic functions are applied differently than regular functions. In BeCecil, since there is only one kind of function, and an exact type has single arrow types as supertypes. In BeCecil, one may also forget exact type information about a generic function using subsumption; one first translates the exact type into an inexact type, and then one may use a rule that is the same as the rule for subtyping generic function types in the λ&-calculus. In BeCecil, there are no dynamic consequences to this forgetting process (that is, each case in the generic function is still used exactly as before). The λ&-calculus also allows supertypes of generic function types; however, forgetting information about branches can have dynamic consequences, because the type indexes in a generic function type are used to select a particular method.

The difference in the treatment of generic function types reflects a more fundamental difference in the two type systems: in BeCecil classes and types are distinguished, as well as inheritance and subtyping. The λ&-calculus does not make these distinctions, and thus its type system is not separated from its dynamic semantics. In BeCecil, the dynamic semantics is independent of the type system. This separation does lead to complications in the type system of BeCecil, but the result is that BeCecil gives programmers added flexibility.

One way to prove the soundness of the BeCecil type system would be to translate BeCecil into the λ&-calculus. We chose not to do so as the translation would not be direct, because the type system of the λ&-calculus does not deal with nested scopes, mutation, or a separation of types and classes.

In summary, although there is a subset of BeCecil that is similar to the λ&-calculus, the following are the main differences.

- BeCecil treats imperative features, such as assignment.
- BeCecil has encapsulation.
- BeCecil does not require that all objects and inheritance relationships be visible globally.

- BeCecil's dynamic semantics does not rely on its type system.
- BeCecil separates the concepts of types and classes, and subtypes and subclasses.

4.2 Castagna's λ_object

The language λ_object [Castagna 95b] is a meta-language for reasoning about OO programs. In particular, λ_object is used to describe the semantics of an (unnamed) toy OO language which is "a mix of Objective-C and CLOS" (page 2). While BeCecil could be compared either to the toy language or to λ_object, BeCecil is more like λ_object, and λ_object has nearly all of the expressive power of the toy language.

The main differences between λ_object and the λ&-calculus are that in λ_object one can declare new classes and inheritance/subtyping relationships, and there is some provision for extending the methods of a generic function that allows for functional update of existing methods (i.e., a new generic function is produced). While one can declare new classes and subtype/inheritance relationships in a λ_object program, all such declarations are global to the main computation, unlike those in BeCecil, and λ_object does not allow mutation. In λ_object there is also some provision for reasoning about information hiding, but unlike BeCecil, information hiding is not enforced. (In the toy language, however, information hiding is enforced. The construct for information hiding is the explicit declaration of the interface of a class (the protocol of its objects), and this is enforced by the type system. Thus information hiding in the toy language is quite similar to the hide mechanism of BeCecil.) λ_object also has two primitives, `coerce` and `super`, that are used for inheritance of methods.

Since classes and types are not separated in λ_object, its `super` mechanism involves the type system. The use of `super` allows one to achieve an effect similar to that of directed actuals in BeCecil. It can be used more freely than directed actuals in BeCecil because of the lack of block structure in λ_object. In BeCecil one could perform the equivalent of `coerce` by creating a new proxy object, however this proxy will have a different object identity than the original object (a concept not present in λ_object), and thus will not be wholly satisfactory for the same reasons that proxy objects are not wholly satisfactory in standard OO languages.

Unlike BeCecil, objects in λ_object are records containing instance variables. In BeCecil, the fields of an object are not part of an object's value. λ_object is also class-based. BeCecil does not make a distinction between classes and instances.

Since it is like the λ&-calculus, λ_object shares several other important differences from BeCecil. These include the distinction between two kinds of applications and the main points summarized above.

4.3 Polyglot

Polyglot, a CLOS-like database type system [Agrawal *et al.* 91], is the only other statically-typed language with multimethods and mutation of which we are aware. The main differences between Polyglot and BeCecil are that Polyglot requires that all objects and inheritance relationships be visible globally, Polyglot does not separate the notions of type and class or subtype and subclass, and Polyglot does not have a way to declare the types of generic functions apart from their implementations.

In stating the type system of BeCecil in Appendix C, we have not tried to describe a type checking algorithm, but rather to describe rules that would guarantee type soundness. However, BeCecil shares an idea from the Polyglot algorithm in that it also divides the typechecking problem into implementation-side and client-side checks. In Polyglot, the implementation-side checks impose a *consistency* condition on the return types of methods of a generic function, which says that more specific methods have more specific return types. (This condition is the same as that used in the category-sorted algebras of Reynolds [Reynolds

80] and in order-sorted algebras [Goguen & Meseguer 87].) The implementation-side checks also require that each set of methods that are *confusable* (i.e., that might handle the same call) is totally ordered by the programmer. Client-side checks thus only need to find a single most-specific multimethod that is applicable to the static types of the arguments in the call, and the consistency condition ensures that at run time the result will be a subtype of the result type of this method. Compared to BeCecil, however, their type system depends on a number of restrictive assumptions:

- The methods of a generic function must be totally ordered by the programmer within each confusable set. Graph-based method lookup semantics found in most object-oriented languages with multiple inheritance [Snyder 86], where the method overriding relationship only forms a partial order, cannot be handled. BeCecil does not make any such assumption, but detects whether any ambiguously-defined messages are sent.
- Because Polyglot has no way to declare the type of a generic function aside from giving its methods, and because Polyglot has no other way to distinguish between concrete and abstract classes, there is no way to declare an “abstract” or “deferred” method.
- Inheritance and subtyping are synonymous in Polyglot. This makes code less flexible, because it prevents a class from inheriting code from some other class without being required to be a subtype [Snyder 86]. It also means that one cannot distinguish the specializer of an argument from its type, and thus one must specialize on all arguments. In BeCecil, one can use a specializer that is distinct from the formal argument type; because the specializer also does not have to conform to the argument type, one can use the specializer `any`, and thus effectively not specialize on that argument.

4.4 Kea

Kea is a higher-order, polymorphic, purely-functional language supporting multimethods [Mugridge *et al.* 91]. Objects in Kea contain various properties, which can be thought of as fields, unlike BeCecil. Information hiding is based on a class mechanism; a class has a declared public interface, and only within a class can other properties of an object be accessed. The hide declaration in BeCecil is similar.

Like Polyglot, code inheritance and subtyping in Kea are unified. Kea’s type checking includes the notion that a collection of methods must be *exhaustive* and *unambiguous*, and these notions appear in our type system as well. The semantics of typechecking in Kea is also specified formally. As with Polyglot, our contribution in the area of typechecking relative to Kea is that we typecheck several important language features not found in Kea, including mutable state, separate subtyping and inheritance graphs (which allow abstract classes), and block structure.

4.5 Rouaix’s Work

Rouaix’s work [Rouaix 90] also describes a higher-order, polymorphic, purely-functional language. This language has no mutable state variables, and does not have information hiding or block structure. This work uses a type inference system that is quite different than the type system of BeCecil. Like BeCecil, the language’s dynamic semantics is specified separately from its type system. However, since there are no type declarations, there is no way to declare abstract or deferred methods.

4.6 CLOS and Dylan

Both the Common Lisp Object System (CLOS) [Steele 90, Paepcke 93] and Dylan [Shalit 97, Feinberg *et al.* 97] are languages with generic functions and module systems. Unlike BeCecil, the generic functions in these languages must all have the same number of (required) arguments. Both module systems provide name-space management, allowing generic functions to be made local or private to packages (although encapsulation is only advisory in Common Lisp and can be circumvented). CLOS allows one to add methods to a generic function, either by making a lexical binding in a nested scope, or by modifying the generic function destructively. Dylan also allows one to declare methods in nested scopes. In these

languages inheritance relationships cannot be scoped separately from objects, and CLOS also allows inheritance relationships to be modified as the program runs. In both languages generic functions are not integrated with objects as they are in BeCecil. Both of these languages are also dynamically typed.

4.7 Abadi and Cardelli's Imperative Object Calculus

Abadi and Cardelli have defined an imperative object calculus, Imp-ς [Abadi & Cardelli 95] [Abadi & Cardelli 96, chapters 10 and 11], that consists of "objects", single-dispatch methods (as in Smalltalk [Goldberg & Robson 83] or Self [Ungar & Smith 87]), "method update, object cloning, and local definitions" [Abadi & Cardelli 95, page 1]. They view method update as a "tamed" version of dynamic inheritance. See Section B.10 for why we use storage tables instead of method update in BeCecil.

Although Imp-ς and BeCecil are not really comparable, because Imp-ς does not support multiple dispatch or inheritance, one can use Imp-ς to test of the claim that BeCecil is able to express features of the OO paradigm, by translating Imp-ς into BeCecil. The main interest is how to encode in BeCecil Imp-ς objects, which are records of methods, since BeCecil's objects are not usually thought of as containing their own methods. However, one can encode an Imp-ς object as a generic function that has storage tables specialized on each label of the record, where labels are considered to be objects that do not inherit from each other [Castagna 95b, Section 4.3]. Each such storage table would contain a generic function that encodes the corresponding method. This translation could be formalized with the following sugars. (In the sugars, *a* and *b* are expressions, *x* is an identifier, and *l* is a label. The first is the sugar for the Imp-ς object expression, which creates an object, and the third is for method replacement.)

⌈ **[** l_1=ς(x_1)b_1, ..., l_n=ς(x_n)b_n **]** ⌉ ≡
    ⌈ **{ gf** *I*
      I **has storage**(*x*@l_1) **:= anon method**(x_1) **{** b_1 **}**
      ...
      I **has storage**(*x*@l_n) **:= anon method**(x_n) **{** b_n **}**
      I
    } ⌉, where *x* and *I* are fresh.

⌈ *a*.*l* ⌉ ≡
    ⌈ **let** *I* **=** *a* **in** (*I*(*l*))(*I*) ⌉

⌈ *a*.*l* ⇐ ς(*x*)*b* ⌉ ≡
    ⌈ *a*(*l*) ⌉ **:=** ⌈ **anon method**(*x*) **{** *b* **}** ⌉

⌈ **clone**(*a*) ⌉ ≡
    ⌈ **let** I_a **=** *a*
      in { gf *I*
          I **has** I_a
          I
        } ⌉, where *I* is different than I_a.

Note that the anonymous methods in the translations do not specialize on their argument. This is because the methods of Imp-ς are not multimethods, and so have no need of specialization. We do not include type information in these sugars, because in Imp-ς the types are not declared, and because Imp-ς does not have user-defined type names. We leave a closer connection between the type systems as future work.

The translation given above assumes some way to code labels as objects in BeCecil. Labels have the property that they are not shadowed by locally-bound variables, and thus form a second namespace. However, this can be easily handled by "name mangling" in the translation of labels into BeCecil identifiers. For example, one might translate labels and identifiers of Imp-ς into BeCecil as follows.

$\lceil l \rceil \equiv$ `label_`l
$\lceil x \rceil \equiv$ `variable_`x

4.8 Block Structure

Few OO languages feature block structure to any significant extent. We know of no other statically typed OO languages with both multimethods and block structure. (Dylan allows one to define methods in nested blocks, but not classes or inheritance relationships. CLOS has lexical closures, but classes and inheritance relationships are global.) C++ [Stroustrup 91] has some amount of block structure, in that one can declare classes within blocks. However, C++ is designed to carefully avoid making static closures for methods.

The main OO languages that allow the full use of block structure are Simula 67 [Birtwistle *et al.* 73] and Beta [Madsen *et al.* 93, Chapter 8]. In Beta there are "fewer restrictions on the use of block structure than in Simula" [Madsen *et al.* 93, page 139]. Beta allows one to define local procedures and classes inside the bodies of other definitions, including procedure and class definitions. We believe that BeCecil supports all the patterns of programming that can be accomplished in Beta in a fairly direct way.

Most interesting to us is the interaction of block structure and inheritance. What happens when one defines a class in a local contour that inherits from a class in a surrounding contour? If an instance of the class is passed out of the local contour, is this inheritance relationship preserved? In Beta, the inheritance relationship is preserved; for example, messages sent to the object are handled by the superclass if the object's own class does not override them. Similarly, in BeCecil, an inheritance relationship declared in an inner block is preserved by objects created in that block. In BeCecil, the situation is a bit more complex, because of the separation of types and classes, and because inheritance relationships can be declared apart from the declaration of objects.

5 Discussion and Conclusions

In this section we discuss the expressive power of BeCecil, the importance of separating the study of a language's dynamic semantics from its type system, and current work. We finish with some conclusions that describe the contributions of this work.

5.1 The Expressive Power of BeCecil

BeCecil is an expressive language. In particular it is able to express in a direct fashion typical patterns of OO programming, including encapsulating state, extending behavior without modifying existing code, and subtype polymorphism. To demonstrate this claim, we have presented several examples, including those that show how to program:

- points with state and object identity,
- grayscale colors that encapsulate state that satisfies an invariant,
- grayscale points that inherit state and behavior from grayscale colors and points,
- booleans and lists that use subtype polymorphism and abstract classes.

Such examples are standard in the literature on OO programming; e.g., the Boolean example is taken from Smalltalk [Goldberg & Robson 83], and the list example is taken from Cook's paper comparing ADT and OO styles [Cook 90]. Such examples demonstrate that BeCecil qualifies as an OO programming language, despite using different mechanisms than Smalltalk or C++ for objects, instance variables, message passing, classes, and inheritance.

These new mechanisms support new patterns that are not possible or feasible with single-dispatching. For example, multimethods solve part of the binary method problem [Bruce *et al.* 95] by giving efficient and direct access to multiple objects when processing a message. BeCecil also supports programming in an abstract data type (ADT) style, where one has complete control over the creation of instances of a class, while still permitting the use of OO patterns within such a class. This is not possible in most purely OO languages, and a mix of styles like this is not possible in most ADT languages either. However, the ability to mix styles comes not from a large number of features, but from the combination of a very small number of orthogonal features.

5.2 The Importance of Dynamic Semantics

Although we designed BeCecil so that it could be statically type checked, we also adopted as a design principle that the language should make sense apart from its type system. While this technique is not new, it does have several benefits. The main benefit is a separation of concerns, which allows each part of the language design to be more highly polished. A prime example of this is the hide declaration in BeCecil. The hide declaration can hide information, and can even prevent impersonation, without any help from the type system. In this respect it differs from CLU [Liskov *et al.* 81] (and from the `abstype` mechanism of Standard ML [Milner *et al.* 90]), which achieves information hiding primarily by a difference between the types of objects as seen by the ADT implementation and its clients. BeCecil's hide declaration can be seen as a highly simplified and focussed version of the module mechanisms in CLOS or Dylan, which also provide information hiding for multimethod languages without the aid of a type system. Another polished feature of the BeCecil dynamic semantics is the treatment of inheritance in nested blocks.

5.3 Current Work

The most pressing current work is the proof that the type system of BeCecil is sound.

We are also working on a module system for BeCecil that will allow separate typechecking of modules and combining independently-developed modules [Chambers & Leavens 95]. This module system allows one module to import another module, or to extend it. Modules that extend some other module are called *extension modules*. Extension modules can, effectively, insert declarations into the same recursive declaration sequence as the modules they extend. This is necessary because the dynamic semantics of BeCecil can only permit extensions to objects declared in the same recursive declaration sequence, and because the type system needs to restrict what can be done in nested blocks. Our previous work did not formalize when one needed to use an extension module, and what privileges were gained by extension module. It is now clear that the privilege gained by an extension module is the ability to effectively insert declarations into the extended module's recursive declaration sequence. Hence one needs to use an extension module to gain this privilege, and does not need to otherwise.

5.4 Conclusions

We have described a new and simple mechanism for information hiding in multimethod languages. This mechanism does not compromise extensibility and can preserve representation invariants. BeCecil also supports programming in a mix of OO and ADT encapsulation styles. The ADT style gives the ability to hide data structures and preserve invariants plus full control over objects that inherit from a given class (by giving up some measure of extensibility). As CLOS and Dylan also have mechanisms for information hiding, our contribution in this area is the simplicity of our mechanism, and the analysis of its semantics and programming properties. This analysis settles the problem of information hiding for languages with multimethods noted by Cook [Cook 90].

BeCecil also demonstrates how to achieve a high degree of extensibility in an OO language with multimethods. In BeCecil, generic functions can be extended with new methods, and objects can be extended with new inheritance relationships by adding new declarations. Existing declarations do not have to be edited at all. We have also shown that it is possible for extensions to be made in nested blocks, where one can declare new classes and new inheritance and subtyping relationships. In a nested block, the extension mechanisms of BeCecil also allow one to customize and tailor generic functions defined in surrounding blocks. While the mechanism for combining the methods of a generic function is based on the λ&-calculus, we have worked out the details of method combination in the presence of nested blocks. A main contribution is the semantics of static inheritance and subtyping in nested blocks, which is flexible and entirely new.

BeCecil also demonstrates how simple and orthogonal features can interact in powerful and beneficial ways. For example, generic functions can subsume several other mechanisms in programming languages, including procedures (dispatched on zero arguments), singly-dispatched methods (dispatched on one argument), and multimethods. The use of generic functions and the integration of generic functions and objects allows one to program with higher-order procedures without sacrificing simplicity. Storage tables, which are new with this work, are another example of such a feature, as they generalize variables, instance variables (fields), and arrays.

Finally, BeCecil provides another way to understand OO programming languages, because it is an OO programming language with very different mechanisms than the usual singly-dispatched languages. For example, in BeCecil, an object does not contain either the methods that apply to it or its instance variables. Despite this, we have shown that in such a language one can program standard OO examples in a straightforward fashion. We believe that the study of such novel ways to achieve the benefits of OO programming languages can only lead to increased understanding of the design issues and semantics of such languages.

Acknowledgments

This work has been done while Leavens was a visiting scholar at the University of Washington. Leavens's research is supported in part by an NSF grant (CCR-9593168) and by a faculty improvement leave from Iowa State. Chambers's work is supported in part by an NSF Young Investigator award (CCR-9457767) and gifts from Sun Microsystems, IBM, Xerox, Pure Software, and Edison Design Group.

Thanks to Michael Ernst, Wilson Hsieh, Dave Grove, and Vassily Litvinov for comments and corrections on earlier drafts. Thanks also to Vassily for several discussions, in particular about the integration of methods and state. Thanks to other members of the Cecil group for other discussions of this work. Thanks to the Computer Science and Engineering Department at the University of Washington for their support of Leavens on his faculty improvement leave.

References

[Abadi & Cardelli 95] Martín Abadi and Luca Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, **1**(3):151-166, 1995.

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

[Ada 83] American National Standards Institute. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD 1815A, February, 1983.

[Agrawal *et al.* 91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. In Andreas Paepcke, editor, *OOPSLA'91 Conference Proceedings, Phoenix, AZ, October, 1991*, volume 26, number 11 of *ACM SIGPLAN Notices*, pp. 113-128. ACM, New York, November, 1991.

[Birtwistle et al. 73] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *SIMULA Begin*. Auerbach Publishers, Philadelphia, Penn., 1973.

[Blascheck 94] Günther Blaschek. *Object-Oriented Programming with Prototypes*. Springer-Verlag, NY, 1994.

[Bobrow, *et al.* 86] Daniel G. Bobrow, Kenneth Kahn, George Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In Norman Meyrowitz (editor), *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21, number 11 of *ACM SIGPLAN Notices*, pp. 17-29. ACM, New York, November, 1986.

[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 *et al.* 92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming, San Francisco, June, 1992*, pp. 182-192, volume 5, number 1 of *LISP Pointers*. ACM, New York, January-March, 1992.

[Castagna *et al.* 95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, **117**(2):115-135. Academic Press. February 1995.

[Castagna 95] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, **17**(3):431-447, 1995.

[Castagna 95b] Giuseppe Castagna. A Meta-Language for Typed Object-Oriented Languages. *Theoretical Computer Science*, **151**(2):297-352. Elsevier Science. November 1995.

[Castagna 96] Giuseppe Castagna. Integration of Parametric and "ad hoc" Second Order Polymorphism in a Calculus with Subtyping. *Formal Aspects of Computing*, **8**(3):247-293, 1996.

[Castagna 96b] Giuseppe Castagna. Instance variables specialization in object-oriented programming, 1996. Obtained from ftp://ftp.ens.fr/pub/dmi/users/castagna/attributes.ps.gz

[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 93] Craig Chambers. Predicate Classes. In *ECOOP '93 Conference Proceedings, Kaiserslautern, Germany, July, 1993*, volume 707 of *Lecture Notes in Computer Science*, pp. 268-296. Springer-Verlag, Berlin, 1993.

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

[Chambers & Leavens 94] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. In *OOPSLA '94 Conference Proceedings, Portland Oregon, October, 1994*, volume 29, number 10 of *ACM SIGPLAN Notices*, pp. 1-15. ACM, New York, October 1994.

[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.

[Clinger & Rees 91]William Clinger and Jonathan Rees (Editors). *Revised^4 Report on the Algorithmic Language Scheme*. November 1991. ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/standards/r4rs.ps.gz

[Cook 90] William Cook. Object-Oriented Programming versus Abstract Data Types. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop Proceedings, Noordwijkerhout, the Netherlands, May/June, 1990*, volume 489 of *Lecture Notes in Computer Science*, pp. 151-178. Springer-Verlag, New York, 1991.

[Dean *et al.* 96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pp. 83-100, San Jose, CA, October 1996.

[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.

[Ghelli 91] Giorgio Ghelli. A Static Type System for Message Passing. In Andreas Paepcke, editor, *OOPSLA '91 Conference Proceedings, Phoenix, AZ, October, 1991*, volume 26, number 11 of *ACM SIGPLAN Notices*, pp. 129-145. ACM, New York, November, 1991.

[Goguen & Meseguer 87] Joseph A. Goguen and Jose Meseguer. Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems. In *Symposium on Logic in Computer Science, Ithaca, NY*, pp. 18-29. IEEE Press, NY, June, 1987.

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

[Gosling *et al*. 96] James Gosling, Bill Joy, Guy Steele, Guy L. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Mass., 1996.

[Gunter 92] Carl Gunter. *Semantics of Programming Languages*. MIT Press, Cambridge, Mass., 1992.

[Harbison 92] Samuel P. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Harrison & Ossher 93] William Harrison and Harold Ossher, Subject-Oriented Programming (A Critique of Pure Objects). In Andreas Paepcke, editor, *OOPSLA '93 Conference Proceedings, Washington, DC, Sept.-October, 1993*, volume 28, number 10 of *ACM SIGPLAN Notices*, pp. 411-428. ACM, New York, October, 1993.

[Leavens 91] Gary T. Leavens. Modular Specification and Verification of Object-Oriented Programs. *IEEE Software* **8**(4), pp. 72-80, July, 1991.

[Liskov *et al.* 81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Volume 114 of *Lecture Notes in Computer Science*, Springer-Verlag, NY, 1981.

[Madsen *et al.* 93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Mass., 1993.

[Meyer 88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 1998.

[Meyer 92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

[Milner *et al*. 90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[Morris 73] James H. Morris, Jr. Protection in Programming Languages. *Communications of the ACM*, **16**(1):15-21, January, 1973.

[Moon 86] David A. Moon. Object-Oriented Programming with Flavors. In Norman Meyrowitz (editor), *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21, number 11 of *ACM SIGPLAN Notices*, pp. 1-8. ACM, New York, November, 1986.

[Mugridge *et al*. 91] W. B. Mugridge, J. G. Hosking, and J. Hamer. Multi-Methods in a Statically-Typed Programming Language. Technical report #50, Department of Computer Science, University of Auckland, 1991. Also appears in Pierre America, editor, *ECOOP '91 Conference Proceedings, Geneva, Switzerland, July, 1991*, volume 512 of Lecture Notes in Computer Science; Springer-Verlag, New York, 1991.

[Nelson 91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Ossher *et al.* 95] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-Oriented Composition Rules. In *OOPSLA'95 Conference Proceedings*, pages 235–250, Austin, TX, October 1995.

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

[Reenskaug & Anderson 92] Trygve Reenskaug and Egil P. Anderson. System Design by Composing Structures of Interacting Objects. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 133–152, 1992.

[Reynolds 80] John C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. In Neil D. Jones (editor), *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark*, pp. 211-258. Volume 94 of Lecture Notes in Computer Science, Springer-Verlag, NY, 1980.

[Rouaix 90] Francois Rouaix. Safe Run-Time Overloading. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, CA*, pp. 355-366. ACM, New York, 1990.

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

[Shilling & Sweeney 89] John J. Shilling and Peter F. Sweeney. Three Steps to Views: Extending the Object-Oriented Paradigm. In Norman Meyrowitz, editor, *OOPSLA '89 Conference Proceedings, New Orleans, Louisiana, October 1989*, volume 24, number 10 of *ACM SIGPLAN Notices*, pp. 353-361. ACM, New York, October, 1989.

[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.

[Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In Norman Meyrowitz, editor, *OOPSLA '87 Conference Proceedings, Orlando, FLorida*, volume 22, number 12, of *ACM SIGPLAN Notices*, pp. 227-241. ACM, New York, December, 1987.

[Van Hilst & Notkin 96] Michael Van Hilst and David Notkin. Using C++ Templates to Implement Role-Based Designs. In *Proceedings of 2nd International Symposium on Object Technologies for Advanced Software*, March 1996.

[Winskel 93] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computer Science Series, MIT Press, Cambridge, Mass., 1993.

[Wirfs-Brock & Johnson 90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, September 1990.

Appendix A Dynamic Semantics

The dynamic semantics of BeCecil is mainly concerned with the following domains: objects, inheritance relations, generic function cases, closures, storage tables, and contexts. Objects are characterized by their identity, their inheritance relation, and their sets of cases. Each object has two sets of cases: a set of invocables and a set of assignables; methods and storage tables are invocable, storage tables and acceptors are assignable. Closures are used to represent both methods and acceptors. An object's set of cases together with its inheritance relation can be used as a generic function. Objects can be named by declarations, and nothing else is a first-class domain in BeCecil. Objects can act as classes if they are statically known (i.e., named in an object declaration).

The domains used to describe the dynamic semantics are summarized in Figure A-1 below. The sequents used in the semantics are summarized in Figures A-2 and A-15. The rules in general (for example, see Figure A-3) have a label enclosed in square brackets ([]), several sequents above a horizontal line, and a single sequent below the line, and a set of non-sequent side conditions to the right following the word "where" [Winskel 93]. The sequents above the line are hypotheses of the rule, and the sequent below the line is its conclusion. A rule with no hypotheses is an axiom (or axiom scheme). The side conditions must

v	∈	*ExpressibleValue*	= *Object*
vv	∈	*ValueVec*	= *ExpressibleValue**
d	∈	*DenotableValue*	= **class**(*Object*) + **arg**(*Object*)
s	∈	*StorableValue*	= *Object*
o	∈	*Object*	= *ObjectId* × *Inherits* × *InvocableSet* × *AssignableSet*
id	∈	*ObjectId*	= *Nat*
ids	∈	*ObjectIdSet*	= *Powerset*(*ObjectId*)
idv	∈	*ObjectIdVec*	= *ObjectId**
ι	∈	*Inherits*	= *FiniteRel*[*ObjectId, ObjectId*]
is	∈	*InvocableSet*	= *PowerSet*(*Case*)
ags	∈	*AssignableSet*	= *PowerSet*(*Case*)
c	∈	*Case*	= *ObjectId** × *Branch*
cs	∈	*CaseSet*	= *PowerSet*(*Case*)
b	∈	*Branch*	= *Closure* + *StorageTable*
k	∈	*Closure*	= **closure**(*Identifier** × *Block* × *Context*)
st	∈	*StorageTable*	= **storage**(*Location*)
t	∈	*Table*	= **table**(*FiniteFun*[*ObjectId*, StorableValue*] × *StorableValue*)
aa	∈	*ActualArgPair*	= *ExpressibleValue* × *Powerset*(*ObjectId*)
aav	∈	*ActualArgPairVec*	= *ActualArgPair**
κ	∈	*Context*	= *Environment* × *Inherits*
η	∈	*Environment*	= *FiniteFun*[*Identifier, DenotableValue*]
ε	∈	*Elaborated*	= *ObjIdEnv* × *Inherits* × *HasEnv*
ω	∈	*ObjIdEnv*	= *FiniteFun*[*Identifier*, *ObjectId*]
h	∈	*HasEnv*	= *FiniteFun*[*ObjectId, InvocableSet* × *AssignableSet*]
oc	∈	*ObjCounter*	= **ocounter**(*ObjectId*)
σ	∈	*Store*	= *FiniteFun*[*Location, StoredValue*]
sv	∈	*StoredValue*	= *ObjCounter* + *Table*
l	∈	*Location*	= *Nat*

Figure A-1: Domains for the dynamic semantics of BeCecil. The domains *FiniteFun* and *FiniteRel*, and operations on *Store*, *Location*, and *Environment* are described in Appendix E.

	type			
prototype sequent	assumes (inherits)	works on (arguments)	produces (synthesizes)	defined in Figure
$\iota \vdash id_1 \leq_{inh} id_2$	*Inherits*	*ObjectId* × *ObjectId*		A-3
$\vdash id_1 \in ancestors(o)$		*ObjectId* × *Object*		A-4
$\sigma \vdash invoke(b,vs) \Rightarrow v/\sigma'$	*Store*	*Branch* × *ExpressibleValue**	*ExpressibleValue* × *Store*	A-6
$\sigma \vdash assign(b,vs,s) \Rightarrow \sigma'$	*Store*	*Branch* × *ExpressibleValue** × *StorableValue*	*Store*	A-8
$\iota \vdash c_1 \leq_{inh} c_2$	*Inherits*	*Case* × *Case*		A-9
$\iota \vdash c_1$ *overrides* c_2	*Inherits*	*Case* × *Case*		A-10
$\iota \vdash c$ *applies-to* *aav*	*Inherits*	*Case* × *ActualArgPair**		A-11
$(\iota,aav) \vdash c$ *is-best-for* *cs*	*Inherits* × *ActualArgPair**	*Case* × *PowerSet*(*Case*)		A-13

Figure A-2: Table of auxiliary sequents for the dynamic semantics. A few helping sequents have been omitted. Sequents that define the semantics of each category in the untyped subset of BeCecil syntax can be found in Figure A-15. For each kind of sequent, the types it relates are shown on three parts on its row: what it assumes, what it works on (or relates), and what it produces. Sequents that do not produce any result can be thought of as producing a boolean result (true if it is provable), or as relations on their arguments.

be satisfied for the rule to be validly applied; if they are, and if the hypotheses hold, then the conclusion holds.

In what follows we describe the semantic domains that are not concerned with type checking, and then describe the dynamic semantics of BeCecil.

A.1 Dynamic Semantics Domains, Auxiliary Sequents, and Functions

This section presents more explanation of the domains in the dynamic semantics, and describes various auxiliary functions and sequents used in the semantic rules. We also use some fairly standard notations for finite functions, stores, environments, and relations; details for these domains are found in Appendix E.

A.1.1 Objects and Object Identities

An *object* has four attributes: its identity, its direct inheritance relation, its invocables set, and its assignable set. No two objects have the same identity.

We also write *oid*(*o*) for the identity of *o*.

oid: *Object* → *ObjectId*
oid(*id*, ι, *is*) = *id*

[inh-base] $\iota \vdash id_1 \leq_{inh} id_2$ where $(id_1, id_2) \in \iota$

[inh-reflex] $\iota \vdash id \leq_{inh} id$

[inh-trans] $$\frac{\iota \vdash id_1 \leq_{inh} id_2 , \quad \iota \vdash id_2 \leq_{inh} id_3}{\iota \vdash id_1 \leq_{inh} id_3}$$

Figure A-3: Axioms and rules for inheritance relationships.

[ancestors] $$\frac{\iota \vdash id_2 \leq_{inh} id_1}{\vdash id_1 \in ancestors(id_2, \iota, is)}$$

Figure A-4: Rule for determining membership in an object's ancestry.

A.1.2 Direct Inheritance, Inheritance Relations, and Ancestry

A direct inheritance relation, ι, is a binary relation on object identities. It models information from inheritance declarations in BeCecil.

For example, if `Larch_rep` denotes the object o_L and if `SpecLang_rep` denotes the object o_S, then the declaration below would be recorded by the direct inheritance relation $\{(oid(o_L), oid(o_S))\}$.

```
Larch_rep inherits SpecLang_rep
```

The semantics, given in Figure A-3, extends a direct inheritance relation, ι, to be a reflexive and transitive relation, $\leq_{inh}$, on object identities and objects. This relation is called an *inheritance relation*. The notation used, $\iota \vdash id_1 \leq_{inh} id_2$, means that, according to ι, id_1 is either the same as id_2, or id_1 inherits (directly or indirectly) from id_2.

As described in Appendix E.4, $cyclic(\iota)$ is true when ι is a cyclic relation, and $\circledast$ is the noncyclic union operator for relations.*

It is often convenient to think of an inheritance relation as a directed graph (with objects identities as nodes, and direct inheritance relationships as directed edges). For example, Figure A-5 shows an inheritance relation among objects as such a directed graph. In this picture, for example, `text_rep` has as ancestors `text_rep`, `string_rep`, `font_rep`, `any`, and `nothing`.

From the direct inheritance relation inside an object, one can determine the object's *ancestors*. The rule for determining whether an object identity is an ancestor of another object is given in Figure A-4.

A.1.3 Overview of Generic Functions, Invocables, Assignables, and Generic Function Attributes

A *generic function* is a function that is generic in the sense that it can work with several different classes of arguments. A generic function exhibits *ad hoc* as opposed to parametric polymorphism. In languages like CLOS, Dylan, and Cecil, a generic function is an abstraction of a set of methods, together with an

* We disallow cyclic inheritance just to be conventional; there is no necessary reason for disallowing it in BeCecil.

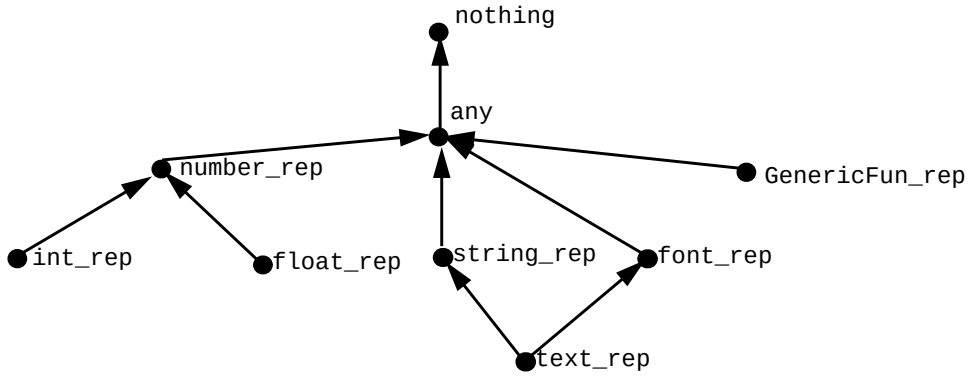

Figure A-5: A picture of an inheritance relation.

inheritance relation. In BeCecil, each object can be treated as a generic function, because it contains an inheritance relation and possibly empty sets of methods, storage tables, and acceptors. The inheritance relation is used in selecting which of these apply to a given tuple of arguments, and in selecting which one of these should be run.

In BeCecil, both methods and storage table attributes can handle applications. Recall that, for brevity, we call something an *invocable* if it is method or storage table attribute. Storage table and acceptor attributes can handle assignments, hence they are called *assignables*. Note that a storage table attribute is both invocable and assignable. A method, storage table, or acceptor attribute is modeled is called a *case*.

In the semantics the domain for cases is a pair that contains a tuple of object identities and a *branch*. A branch is either a closure or a storage table. An case's tuple of object identities is called its tuple of *specializers*; a generic function uses these specializers to determine the classes of objects to which the corresponding branch may be applied. Closures are used to model branches that are extracted from both method and acceptor attributes; storage tables are used for storage table attributes. For example, the method below is be modeled by an case with (`int_rep`, `int_rep`) as its specializers, and a branch that is a closure.

```
method(x@int_rep, y@int_rep) { less(plus(x,y), 3) }
```

A.1.4 Closures and Storage Tables

A *closure* consists of: formals (a tuple of identifiers), a block (code to execute), and a context. The process of invoking a closure is described by the rule given in Figure A-6. To invoke a closure, a new environment, η', is formed by binding the actuals to the formals, with each formal shadowing bindings in the environment, η, that is extracted from the method's closure. (Thus, this gives static scoping.) The formals must all have distinct names, otherwise the side condition that forms the environment η' will not be satisfied, because the strict equality ($=_s$) requires that both sides be proper (non-$\perp$) for the relationship to be true. In this context, the body of the method is evaluated, using the rule for blocks given in Figure A-18, to produce a value and a new store.

A *storage table* is modeled by a location containing a *table.* A storage table can be "invoked" with an *n*-tuple of arguments; such an *n*-tuple is called a *key* for the table. Keys are compared pointwise; that is, two keys are considered equal if they have the same length and the corresponding components have the same object identity. To each key the table associates a *value*. Informally, a table can be pictured (see Figure A-7) as having *n+1* columns, and as many rows as there are keys that have been defined, plus a special "default key" that is associated to the table's initializer's value (the "default value"). In each row there is an *n+1*-tuple, consisting of a key and its associated value. No two rows have the same key. To invoke the table for

[invoke closure]
$$\frac{(\eta',\iota) / \sigma \vdash B \Rightarrow v/\sigma'}{\sigma \vdash \mathit{invoke}(\mathbf{closure}((I_1,...,I_n), B, (\eta,\iota)), (v_1,...,v_n)) \Rightarrow v/\sigma'}$$
where $n \geq 0$, $\eta' =_s \eta \cup \mathit{bind}((I_1,...,I_n), (\mathbf{arg}(v_1),...,\mathbf{arg}(v_n))$

[invoke storage 1]
$$\sigma \vdash \mathit{invoke}(\mathbf{storage}(l), (v_1,...,v_n)) \Rightarrow s/\sigma$$
where $n \geq 0$, $\mathit{lookup}(\sigma, l) =_s \mathbf{table}(f, s_d)$, $((\mathit{oid}(v_1),...,\mathit{oid}(v_n)):s) \in f$

[invoke storage 2]
$$\sigma \vdash \mathit{invoke}(\mathbf{storage}(l), (v_1,...,v_n)) \Rightarrow s_d/\sigma$$
where $n \geq 0$, $\mathit{lookup}(\sigma, l) =_s \mathbf{table}(f, s_d)$, $(\mathit{oid}(v_1),...,\mathit{oid}(v_n)) \notin \mathit{dom}(f)$

Figure A-6: Rule and axioms for invocation. The symbol $=_s$ means strict equality, as explained in the text.

$id_{string,45}$	$id_{int,4}$	$id_{float,4.7}$
$id_{string,227}$	$id_{int,32}$	$id_{float,0.0}$
$id_{string,541}$	$id_{int,3}$	$id_{float,3.14}$
default key		$id_{float,0.0}$

Figure A-7: A picture of a storage table. Here $id_{string,45}$ is the identity of an object that inherits from `string_rep`, and $id_{int,4}$ is the identity of an object that inherits from `int_rep`.

[assign storage]
$$\sigma \vdash \mathit{assign}(\mathbf{storage}(l), (v_1,...,v_n), s) \Rightarrow \sigma'$$
where $n \geq 0$, $\mathit{lookup}(\sigma, l) =_s \mathbf{table}(f, s_d)$, $t = \mathbf{table}(f[(\mathit{oid}(v_1),...,\mathit{oid}(v_n)) := s], s_d)$, $\sigma' = \sigma[l := t]$

[assign acceptor]
$$\frac{(\eta',\iota) / \sigma \vdash B \Rightarrow v/\sigma'}{\sigma \vdash \mathit{assign}(\mathbf{closure}((I_1,...,I_n,I_{n+1}),B,(\eta,\iota)), (v_1,...,v_n), s) \Rightarrow \sigma'}$$
where $n \geq 0$, $\eta' =_s \eta \cup \mathit{bind}((I_1,...,I_n,I_{n+1}), (\mathbf{arg}(v_1),...,\mathbf{arg}(v_n),\mathbf{arg}(s))$

Figure A-8: Axiom and rules for processing assignments.

a given key (i.e., to read the table), one finds the row with the same key, and returns the associated value in that row; if such a row cannot be found, the default value is returned (see Figure A-6). An example, pictured in Figure A-7, is a table that accepts two arguments, whose tuple of specializers is (`string_rep`, `int_rep`).

Storage tables can also process assignments. Using the picture in Figure A-7, to assign to a table for a given key, one finds the row with the same key, and places the desired value in it, discarding the old value; if such a row cannot be found, a new row is added that associates the given key to the given value. In the formal model, given in Figure A-8, the whole table is replaced in the store's mapping for the storage table's location.

$$\text{[inh-case]}\quad \frac{\iota \vdash id_{1,1} \leq_{inh} id_{2,1},\ ...,\ \iota \vdash id_{1,n} \leq_{inh} id_{2,n}}{\iota \vdash ((id_{1,1},...,id_{1,n}),\ b_1) \leq_{inh} ((id_{2,1},...,id_{2,n}),\ b_2)} \quad \text{where } n \geq 0$$

Figure A-9: Rule for extending the inheritance ordering to cases.

$$\text{[overrides]}\quad \frac{\iota \vdash c_1 \leq_{inh} c_2}{\iota \vdash c_1\ overrides\ c_2} \quad \text{where } n \geq 0,\ specializers(c_1) \neq specializers(c_2)$$

Figure A-10: Rule for when a case overrides another.

A.1.5 The Model of Acceptors

The case that models an acceptor attribute has a branch that is a closure. This closure takes the value given in an assignment as its last argument. For example, consider the following acceptor attribute.

```
acceptor(p@Point_rep) := v { x(p) := v; y(p) := v }
```

This attribute would be modeled by a case such as the following, where id_{Point_rep} is the identity of the specializer class `Point_rep`, B is the block that forms the body of the acceptor (the code itself), and κ is an appropriate context.

$((id_{Point\_rep}),\ \textbf{closure}((p,v),B,\kappa))$

Note that the tuple of specializers in the above case has a length that is one less than the number of formals for the closure. This is because the value is passed as the last argument, but is not involved in determining applicability. The formal rule for how an acceptor processes an assignment is given in Figure A-8.

A.1.6 Orderings on Cases and Case Sets

Given a case, c, we write $specializers(c)$ for the tuple of specializers of c.

$specializers: Case \rightarrow ObjectId^*$
$specializers(idv, b) = idv$

It is often convenient to describe the set of specializer tuples of a set of cases. The following abbreviation is an instance of the general pointwise extension of functions to sets.

$$specializers(is) \equiv \{idv \mid c \in cs,\ specializers(c) = idv\}$$

As in [Chambers & Leavens 95], we extend an inheritance relation to a partial order on cases. This relation compares the specializers of cases pointwise. The rule for this relation is given in Figure A-9. This ordering reflects the method lookup semantics in Cecil. (Other rules might be needed to model other languages, such as CLOS or Dylan.) For a given direct inheritance relation, ι, a case $c_1\ overrides\ c_2$ when c_1 is strictly more specific in this ordering than c_2. The rule for this relation is given in Figure A-10.

Two cases *clash* when they have exactly the same tuples of specializers. We use the concept of nonclashing sets of cases to explicitly define what the disjoint union ($\cup$) and overriding ($\uplus$) notations for finite functions (from Appendix E) mean for case sets. These are equivalent definitions, because a set of cases is also a finite relation between tuples of specializers and branches, as it is a set of pairs of specializer tuples and branches.

$nonclashing: Powerset(Case) \times Powerset(Case) \rightarrow Boolean$
$nonclashing(cs_1, cs_2) = ((specializers(cs_1) \cap specializers(cs_2)) = \{\})$

[applies-to]
$$\frac{\vdash id_{1,a} \in ancestors(v_1),\ ...,\ \vdash id_{n,a} \in ancestors(v_n),\quad \iota \vdash id_{1,a} \leq_{inh} id_1\ ,\ ...,\ \iota \vdash id_{n,a} \leq_{inh} id_n,\quad \iota \vdash ids_{dir,1}\ ok\text{-}dir\ id_1\ ,\ ...,\ \iota \vdash ids_{dir,n}\ ok\text{-}dir\ id_n}{\iota \vdash ((id_1,...,id_n),b)\ applies\text{-}to\ ((v_1,ids_{dir,1}),\ ...,\ (v_n,ids_{dir,n}))} \quad \text{where } n \geq 0$$

Figure A-11: Rule for applicability of cases to tuples of actual argument pairs.

[ok-dir-empty]
$$\iota \vdash \{\}\ ok\text{-}dir\ id$$

[ok-dir-directed]
$$\frac{\iota \vdash id' \leq_{inh} id}{\iota \vdash ids\ ok\text{-}dir\ id} \quad \text{where } id' \in ids$$

Figure A-12: Treatment of directed sets for applicability.

$\cdot \cup \cdot$: $Powerset(Case) \times Powerset(Case) \to Powerset(Case)_\perp$
$cs_1 \cup cs_2$ = **if** $nonclashing(cs_1, cs_2)$ **then** $cs_1 \cup cs_2$ **else** $\perp$
$\cdot \uplus \cdot$: $Powerset(Case) \times Powerset(Case) \to Powerset(Case)$
$cs_1 \uplus cs_2 = cs_2 \cup \{c \mid c \in cs_1, specializers(c) \notin specializers(cs_2)\}$

A.1.7 Actual Arguments and Applicability

An *actual argument pair,* (v, ids_{dir}), consists of an expressible value, v, and a set of object identities. The set of object identities is the *set of directed classes*, ids_{dir}, which is either empty or contains the classes named in the directed form of an actual argument.

A case *c applies to* a tuple of actual argument pairs if for each actual argument pair, some ancestor of the value inherits from the corresponding specializer of *c* and if, where the corresponding directed set is nonempty, some directed class inherits from the corresponding specializer. The rules for this are defined in Figures A-11 and A-12.

For example, a method that has (`int_rep`, `int_rep`) as its specializers is applicable to the actuals (3, 7) and (3`@int_rep`, 7`@int_rep`). However, according to the inheritance relation in Figure A-5, that method would not be applicable to the actuals (3`@int_rep`, 4.7`@float_rep`), or to the actuals (3`@int_rep`, 4.7`@int_rep`) because in both cases the second value has no ancestors that inherit the method. If the actuals were (3, 7`@number_rep`), then again the method would not be applicable, but this time because the directed class set given for the second actual has no members that inherit the method. On the other hand, a method that had (`int_rep`, `number_rep`) as its specializers would be applicable to all of these actuals. (There is more discussion on the subtleties of applications in Appendix B.)

A.1.8 Processing Applications and Assignments

To process an application of an object's generic function value to a tuple of actual argument pairs, the object can be thought of as acting as follows. It first looks at each case in its invocable set to see which ones are applicable to the tuples of actual argument pairs extracted from the actuals. In this subset of applicable cases, it looks for a unique case that overrides all of the other cases. (For both applicability and for ordering cases, it uses its own inheritance relation. The inheritance relation of the actual arguments is used only to determine their ancestry.) The rules for finding the most specific case in a set, given a direct inheritance

[is-best-for]
$$\frac{(\iota, aav) \vdash c \textit{ better-than } c_1, \quad \ldots, \quad (\iota, aav) \vdash c \textit{ better-than } c_n}{(\iota, aav) \vdash c \textit{ is-best-for } cs}$$
where $cs' = \{c' \mid c' \in is, \iota \vdash c \textit{ applies-to } aav\}$, $c \in cs'$, $\{c_1, \ldots, c_n\} = cs'$, $n \geq 1$

Figure A-13: Rule for when a case is the unique most-specific that applies to a tuple of actual argument pairs.

[better-than-eq]
$$(\iota, aav) \vdash c \textit{ better-than } c$$

[better-than]
$$\frac{\iota \vdash c \textit{ overrides } c'}{(\iota, aav) \vdash c \textit{ better-than } c'}$$
where $c \neq c'$

Figure A-14:Rules for when one case is better than another. Note that these rules are mutually exclusive.

relation and a tuple of actual argument pairs, are given in Figures A-13 and A-14. Once found, the unique, best case is invoked as described above (see Section A.1.4).

There are two errors that can arise in the processing of an application. If there are no applicable cases, then the error “message not understood” occurs. If there is not a unique most-specific case, then the error “message ambiguous” occurs. These errors are not modeled explicitly by the semantics, instead the processing of such an application will get “stuck” because the rule [is-best-for] will not apply.

The generic function of an object can also process an assignment. To do this for a tuple of actual argument pairs and a desired value, it acts as follows. If first goes through the process of determining a unique most-specific case. It does this as described above with the key of the assignment playing the role of the actual parameters; however, it uses its set of assignables instead of its set of invocables. (Note that a storage table will be in both sets.) Once found, the unique most-specific case is given control, as described in Figure A-8. Errors can occur if there is not a unique most-specific case, as for processing an application.

A.1.9 Contexts, Environments, and Denotable Values

A *context* records information from declarations that can be used by other declarations or expressions. A context is modeled as a pair of an environment and a direct inheritance relation. Environments map identifiers to denotable values (see Appendix E.3 for more details on environments). Denotable values are objects.

A.1.10 Elaborateds

An *elaborated* records information from the elaboration of declarations. It is a kind of context used internally by the semantics. An elaborated is capable of recording information from any sequence of declarations. An elaborated is modeled as a triple of an “object-id-environment,” a direct inheritance relation, and a “has-environment.” An object-id-environment is a finite function from identifiers to object identities. A has-environment is a finite function from object identities to pairs of invocable and assignable sets; this tracks information from the extension (`has`) declarations in BeCecil.

The operator $\cup$ is used to collapse the elaborateds produced by declarations at the same level into a single elaborated. The object-id-environment is merged using the disjoint-union operator, $\cup$, which disallows

$$\text{[program]}\quad \frac{(\{\},\{\})/\sigma_{init} \vdash RDS \blacktriangleright \kappa/\sigma_{pre},\ \kappa/\sigma_{pre} \vdash B \Rightarrow v/\sigma'}{\vdash RDS\ ;\ B \Rightarrow v/\sigma'} \qquad \text{where } \sigma_{init} = \{l_{idcounter}\text{: } \mathbf{ocounter}(0)\}$$

Figure A-16: Dynamic semantics for programs. The conditions are explained in the text, as are details of the sequents.

duplicate object bindings. The non-cyclic union operator, $\circledcirc$, is used to merge the direct inheritance relations. However, for the has-environment, if the same object identity is bound to two pairs of invocable and assignable sets, then the sets are simply unioned together ($is = is_1 \cup is_2$ and $ags = ags_1 \cup ags_2$). If we had used the disjoint union operator for this, then clashes among the invocables or assignables bound to the same identifier would cause errors at this time (that is, when two elaborateds are combined). Instead, the regular union allows such potential errors, which may generate a "message ambiguous" error if, for example, two clashing invocables are both applicable to a given tuple of actual argument pairs.

$\cdot \cup \cdot$: $Elaborated \times Elaborated \to Elaborated_\perp$
$(\omega_1, \iota_1, h_1) \cup (\omega_2, \iota_2, h_2) = (\omega_1 \cup \omega_2, \iota_1 \circledcirc \iota_2, blendGfs(h_1, h_2))$
$blendGfs$: $HasEnv \times HasEnv \to HasEnv_\perp$
$blendGfs(h_1, h_2) = \{(id{:}(is,ags)) \mid (id{:}(is_1,ags_1)) \in h_1, (id{:}(is_2,ags_2)) \in h_2, is = is_1 \cup is_2, ags = ags_1 \cup ags_2\}$
$\cup \{(id{:}(is_1,ags_1)) \mid (id{:}(is_1,ags_1)) \in h_1, id \notin dom(h_2)\}$
$\cup \{(id{:}(is_2,ags_2)) \mid (id{:}(is_2,ags_2)) \in h_2, id \notin dom(h_1)\}$

A.1.11 Object Counters and the Store

The *store* maps locations to storable values. A storable value is either an object counter or a storage table. There is only one location that stores an object counter, $l_{idcounter}$. The value of the counter stored in this location is used to assign unique identities to objects.

Locations that hold storage tables should be thought of as pointers to blocks of storage reserved for storing the table.

A.2 Dynamic Semantics

In this section we explain the dynamic semantics rules given in Figures A-16 through A-25. The order of presentation follows the order of the syntax of Figure 2-1. The rules form a "big step" semantics [Gunter 92] for BeCecil. The judgements used in this semantics are summarized in Figure A-15. The idea behind the different kinds of arrows used in the judgements is that the arrows of the form ➪ are used for judgements that yield values, arrows of the form ➤ are used for judgements that elaborate declarations and yield contexts or elaborateds, and arrows of the form ➨ are use for judgements that yield actual-argument pairs. The slashes in sequents are used to separate parts of inherited and synthesized attributes. Thus a sequent of the form $\kappa/\sigma_1 \vdash B \Rightarrow v/\sigma_2$ can be read as: "assuming the context κ and the store σ_1, one can prove that B evaluates to the expressible value v, with side-effects recorded in the new store σ_2."

A.2.12 Programs

The dynamic semantics for BeCecil programs is given in Figure A-16. The evaluation of a program produces an expressible value and a final store.

A BeCecil program consists of a recursive declaration sequence, which represent a "standard prelude" (containing objects that are considered to be built-in to the language), and a block. The declarations in the standard prelude are elaborated starting from an empty context with an initial store, σ_{init}. The initial store is

	type			
prototype sequent	assumes (inherits)	works on (category)	produces (synthesizes)	defined in Figure
$\vdash P \Rightarrow v/\sigma$		*Program*	*ExpressibleValue* $\times$ *Store*	A-16
$\kappa/\sigma \vdash RDS \blacktriangleright \kappa'/\sigma'$	*Context* $\times$ *Store*	*Recursive-Declaration-Sequence*	*Context* $\times$ *Store*	A-17
$\kappa/\sigma \vdash B \Rightarrow v/\sigma'$	*Context* $\times$ *Store*	*Block*	*ExpressibleValue* $\times$ *Store*	A-18
$\kappa/\sigma \vdash D^* \blacktriangleright \varepsilon/\sigma'$	*Context* $\times$ *Store*	*Declaration**	*Elaborated* $\times$ *Store*	A-19
$\kappa/\sigma \vdash D \blacktriangleright \varepsilon/\sigma'$	*Context* $\times$ *Store*	*Declaration*	*Elaborated* $\times$ *Store*	A-20
$\eta \vdash CN \Rightarrow_{class} o$	*Environment*	*Class-Name*	*Object*	A-21
$\kappa/\sigma \vdash GF \Rightarrow (is,ags)/\sigma'$	*Context* $\times$ *Store*	*Generic-Function*	*InvocableSet* $\times$ *AssignableSet* $\times$ *Store*	A-22
$\kappa/\sigma \vdash E \Rightarrow v/\sigma'$	*Context* $\times$ *Store*	*Expression*	*ExpressibleValue* $\times$ *Store*	A-23
$\eta \vdash I \Rightarrow v$	*Environment*	*Identifier*	*ExpressibleValue*	A-24
$\kappa/\sigma \vdash AA \blacktriangleright (v, ids)/\sigma'$	*Context* $\times$ *Store*	*Actual-Argument*	*ExpressibleValue* $\times$ *Powerset(ObjectId)* $\times$ *Store*	A-25

Figure A-15: Table of sequents for the dynamic semantics. This table includes a sequent for each syntactic category; see Figure A-2 for auxiliary sequents. For each kind of sequent, the types it relates are shown in three parts on its row: what it assumes, what it works on, and what it produces (or, in the jargon, what it inherits, what syntactic categories it manipulates, and what it synthesizes).

empty except that it maps the location for the object counter to an object counter containing 0. The details of the elaboration process are described below. This elaboration produces a context, κ, and store, σ_{pre}, which are used to evaluate the block.

A.2.13 Recursive Declaration Sequences

The elaboration of a recursive declaration sequence in an assumed context and store produces a context and a new store. The formal rule is given in Figure A-17.

Technically, the semantics of forming the context, (η',ι'), from a recursive declaration sequence is a bit tricky. The main trick that allows mutual recursion among the declarations is that the context assumed for

[rec-decl-sequence]

$$\frac{(\eta',\iota') / \sigma_0 \vdash D^* \blacktriangleright (\omega, \iota, h)/\sigma'}{(\eta_0,\iota_0)/\sigma_0 \vdash D^* \blacktriangleright (\eta',\iota') /\sigma'}$$

where $\iota' =_s \iota_0 \circledcirc \iota$,
$\eta' = closeGfsWith(\omega, \iota', h)$,
$\eta' = \eta_0 \uplus \eta$,
$\neg\, hasOrphans(\omega, \iota, h)$

Figure A-17: Dynamic semantics of recursive declaration sequences.

[block]

$$\frac{\kappa_0/\sigma_0 \vdash RDS \blacktriangleright \kappa/\sigma,\ \kappa/\sigma \vdash E \Rightarrow v/\sigma'}{\kappa_0/\sigma_0 \vdash RDS\ E \Rightarrow v/\sigma'}$$

Figure A-18: Dynamic semantics for blocks.

the elaboration of the declaration sequence, (η',ι'), is the same context as that produced by as part of the result of the rule. (One way to think of this is that the reader must pick (η',ι') to satisfy the rule.)

The elaboration of a declaration sequence, *D**, like the elaboration of an individual declaration, does not produce a context, but an elaborated (see Section A.1.10). This elaborated, (ω,ι,h), contains all the information from the declarations in *D**, but invocables and assignables still have to be combined with the objects they extend, and these objects must also be closed with the appropriate direct inheritance relation. The context (η',ι') is formed from this elaborated in two steps. First, the direct inheritance relation, ι', is formed as the noncyclic union of the surrounding contour's inheritance relation, ι_0, and the direct inheritance from the elaborated declarations. Recall that the notation, $=_s$, is a strict equality relation, meaning it only holds if both sides are proper (non-$\perp$). Thus $\iota' =_s \iota_0 \circledcirc \iota$ means that ι' is the union of ι_0 and ι, and that the union is not cyclic.

Then an environment, η, is formed. In this environment, the objects declared in the recursive declaration sequence are closed with the inheritance relation, ι', and the information from the has-environment, *h*. The auxiliary function for doing this is defined below.

$$closeGfsWith: ObjIdEnv \times Inherits \times HasEnv \to Environment$$
$$closeGfsWith(\omega, \iota, h) = \{(I{:}\mathbf{class}(id, \iota, is, ags)) \mid (I{:}id) \in \omega, (id{:}(is,ags)) \in h\} \cup \{(I{:}\mathbf{class}(id, \iota, \{\}, \{\})) \mid (I{:}id) \in \omega, id \notin dom(h)\}$$

The environment η' is formed from η as described in the rule. In this environment, the declarations in η shadow those in the surrounding contour's environment, η_0.

It is possible that an object identity bound to an extension in the has-environment is not one of the objects declared in the recursive declaration sequence. This could happen, for example, if one tried to extend an object declared in a surrounding contour. We call such extensions "orphans." An orphaned extension is an error in BeCecil.* See Section B.3 for a discussion on why orphans are considered errors.

$$hasOrphans: Elaborated \to Boolean$$
$$hasOrphans(\omega, \iota, h) = (\exists id\ .\ id \in dom(h) \wedge id \notin range(\omega))$$

A.2.14 Blocks

The evaluation of a block in an assumed context and store produces an expressible value and a new store. The formal semantic rule is given in Figure A-18.

* Orphans could just be ignored, but programs containing them are clearly malformed, and so that would be confusing.

$$\text{[declaration sequence]}\quad \frac{\kappa_0/\sigma_0 \vdash D_1 \blacktriangleright \varepsilon_1/\sigma_1,\ ...,\ \kappa_0/\sigma_{n-1} \vdash D_n \blacktriangleright \varepsilon_n/\sigma_n}{\kappa_0/\sigma_0 \vdash (D_1\ ...\ D_n) \blacktriangleright \varepsilon'/\sigma_n} \quad \text{where } n \geq 0,\ \varepsilon' =_s \cup_{i \in \{1,...,n\}} \varepsilon_i$$

Figure A-19: Dynamic semantics of declaration sequences.

In a block, of the form *RDS E*, the recursive declaration sequence *RDS* is elaborated as described above to produce a new context, κ, and a new store, σ, that records any side effects in the elaboration of the declarations. This context and store are used to evaluate the expression, *E*.

A.2.15 Declaration Sequences

The elaboration of a declaration sequence in an assumed context and store produces an elaborated and a new store. The formal semantics is given in Figure A-19.

In a declaration sequence, the individual declarations each elaborate to an elaborated and a new store. (The store is needed because some declarations have side-effects, such as allocation and initialization of storage tables.)

When a declaration sequence is complete, the elaborateds that result from the elaboration of individual declarations are collapsed into a single elaborated, which combines all of their information. The error checking that is done at this time is the following.

- Each identifier *I* is bound to at most one object.
- The union of the direct inheritance relations must not have any cycles.

The operation for actually combining the elaborateds produced by *n* declarations is the disjoint union operator on elaborateds, $\cup$. The error checking is done by using strict equality ($=_s$) in the side condition, as the operator returns $\perp$ if such an error is present.

The side-effects happen left-to-right in a declaration sequence, including allocation of storage. Thus, although declaration sequences in BeCecil are always mutually recursive, one cannot use storage tables until they are initialized. For example, in the following, the use of `x` in an expression before its storage table is allocated causes an error. (In the formal semantics, this errors occur when the elaboration gets "stuck.")

```
y has storage() := plus(x(), 1)   -- error (uninitialized)!
x has storage() := 3
```

In the above example, reordering the declarations makes the error go away. Another way to fix the error is to delay the execution of the expression, by using a method instead of a storage table.

```
y has method() { plus(x(), 1) }
x has storage() := 3
object x
x inherits GenericFun_rep
object y
y inherits GenericFun_rep
y()
```

Except for side-effects, the order of declarations in a declaration sequence does not matter. For example, the declarations in the block above could be arbitrarily reordered, without affecting the final value.

[object declaration]

$$\kappa/\sigma \vdash \textbf{object}\ I \Rightarrow (\{I{:}id\}, \{\}, \{\})/\sigma'$$

where $lookup(\sigma, l_{idcounter}) =_s$ **ocounter**(id), $\sigma' = \sigma[l_{idcounter} := \textbf{ocounter}(id+1)]$

[inherits declaration]

$$\frac{\eta \vdash CN_1 \rightarrowtriangle_{class} o_1, \qquad \eta \vdash CN_2 \rightarrowtriangle_{class} o_2}{(\eta,\iota)/\sigma \vdash CN_1\ \textbf{inherits}\ CN_2 \Rightarrow (\{\},\{(id_1, id_2)\},\{\})/\sigma}$$

where $id_1 = oid(o_1)$, $id_2 = oid(o_2)$

[extension declaration]

$$\frac{\eta \vdash CN \rightarrowtriangle_{class} o, \qquad (\eta,\iota)/\sigma \vdash GF \rightarrowtriangle (is,ags)/\sigma',}{(\eta,\iota)/\sigma \vdash CN\ \textbf{has}\ GF \Rightarrow (\{\}, \{\}, \{id{:}(is,ags)\})/\sigma'}$$

where $id = oid(o)$

[hide declaration]

$$\frac{(\eta_0,\iota_0)/\ \sigma_0 \vdash RDS \Rightarrow (\eta',\iota')/\sigma, \quad (\eta',\iota')/\ \sigma \vdash D^* \Rightarrow (\omega'',\iota'',h'')/\sigma'}{(\eta_0,\iota_0)/\sigma_0 \vdash \textbf{hide}\ RDS\ \textbf{in}\ \ D^*\ \textbf{end} \Rightarrow (\omega'',\iota'',h'')/\sigma'}$$

where $(\forall I\ .\ I \in dom(\omega'') \Rightarrow \eta_0(I) = \eta'(I))$

Figure A-20: Dynamic semantics of declarations.

A.2.16 Declarations

The elaboration of a declaration in an assumed context and store produces an elaborated and a new store. The formal semantics is given in Figure A-20.

An object declaration, of the form **`object`** *I*, is elaborated by allocating a fresh object identity, and binding the identifier *I* to it.

An inheritance declaration, of the form CN_1 **`inherits`** CN_2, is elaborated by producing a direct inheritance relation that relates the first class named to the second.

An extension declaration, of the form *CN* **`has`** *GF*, the generic function attribute *GF* is elaborated to a pair of an invocable set and an assignable set. This pair is bound to the identity of *CN* in the has-environment.

In a hide declaration, of the form **`hide`** *RDS* **`in`** *D** **`end`**, both sets of declarations are mutually recursive with each other. However, only the effect of the public declarations, *D**, is seen in the resulting elaborated, as the recursive declaration sequence *RDS* is hidden. The declarations in *D** see each other as usual, because they are part of a larger recursive declaration sequence, but they also see the hidden declarations, *RDS*. The hidden declarations, *RDS*, also see the declarations in *D**, which appear through the surrounding context (as the surrounding contour is formed by a recursive declaration sequence). This is illustrated by the following example, in which both `even` and `odd` can call each other.

```
hide
   object even
   even inherits GenericFun_rep
   even has method(i@int_rep) { or(equal(i,0), [odd(minus(i,1))] }
in
   object odd
   odd inherits GenericFun_rep
   odd has method(i@int_rep) { or(equal(i,1), [even(minus(i,1))] }
end
```

[class name] $\eta \vdash I \rightarrowtriangle_{class} o$ where $(I{:}\textbf{class}(o)) \in \eta$

Figure A-21: Dynamic semantics of class names.

The side condition in the rule for the hide declaration prohibits declaring the same object in both parts of a hide declaration. The reason for this is that it allows the declarations in the public part of the hide declaration to see each other through the surrounding contour' s context.

As in CLOS and Dylan, hiding is not done on a per-method basis, but on a per-object (i.e., generic function) basis. That is, if a generic function is hidden, so are all of its methods. Thus it would be misleading to allow one to swap the places of the extension declarations for `even` and `odd` in the above example, as it would look like the method for `odd` was being hidden, when it must be public if the object `odd` is public. The semantic rule for BeCecil checks for this by only allowing extension declarations to appear in the same part of a hide declaration as the object declarations they refer to.

The objects declared in the hidden declarations of a hide declaration are closed with the direct inheritance relation of their recursive declaration sequence (with the hidden declarations added to the surrounding contour's inheritance relation), just as if they were the declarations in a block. In the rule given in Figure A-20, they are closed with the direct inheritance relation, ι'. On the other hand, the direct inheritance relation closed with the objects declared in the public part (ι_0) is the one from their own recursive declaration sequence, which does not include the hidden inheritance declarations. See Section 2.5 for more discussion on this topic.

A.2.17 Class Names

A class name is looked up in the assumed environment, and its value is returned. The formal axiom is given in Figure A-21. Note that the identifier in question must denote a class, not an argument.

A.2.18 Generic Function Attributes

The evaluation of a generic function attribute in an assumed context and store produces an pair of an invocable set and an assignable set, and a new store. The formal semantics is given in Figure A-22.

An identifier attribute, of the form *I*, produces a pair of sets containing the cases of the object denoted by *I*, with each storage table being allocated a fresh location. The auxiliary functions that allocate new locations and make copies of each storage table are defined below.

$copyStorage$: $CaseSet \times Store \rightarrow (CaseSet \times Store)_\perp$
$copyStorage(\{c_1,...,c_n\}, \sigma) =$
  let $((c'_1,...,c'_n),\sigma_n) = copylist((c_1,...,c_n), \sigma)$ **in** $(\{c'_1,...,c'_n\}, \sigma_n)$
$copylist$: $Case^* \times Store \rightarrow (Case^* \times Store)_\perp$
$copylist((),\sigma) = ((),\sigma)$
$copylist((c_1,c_2,...,c_n),\sigma) =$
  let $((c'_1,\sigma_1) = copy1(c_1,\sigma)$ **in let** $((c'_2,...,c'_n),\sigma_n) = copylist((c_2,...,c_n),\sigma_1)$ **in** $((c'_1,c'_2,...,c'_n),\sigma_n)$
$copy1$: $Case \times Store \rightarrow (Case \times Store)_\perp$
$copy1((idv,\textbf{closure}((I_1,...,I_m),B,\kappa)), \sigma) = ((idv,\textbf{closure}((I_1,...,I_m),B,\kappa)), \sigma)$
$copy1((idv,\textbf{storage}(l)), \sigma) =$
  let $(l',\sigma') = allocate(\sigma, lookup(\sigma,l))$ **in** $((idv,\textbf{storage}(l')), \sigma')$

Since each storage table is in both the invocable and assignable sets, care must be taken to only make a copy of the storage tables in one of these.

[identifier attribute]

$$\frac{\eta \vdash I \Rightarrow (id, \iota, is, ags)}{(\eta,\iota)/\sigma \vdash I \Rightarrow (is', ags')/\sigma'}$$

where
$(is', \sigma') =_s copyStorage(is, \sigma)$
$ags' = \{(idv, b) \mid b \in is', b = \textbf{storage}(l)\}$
$\cup \{(idv, b) \mid b \in ags, b = \textbf{closure}((I_1,...,I_n), B, \kappa)\}$

[method attribute]

$$\frac{\eta \vdash CN_1 \Rightarrow_{class} o_1, ..., \eta \vdash CN_n \Rightarrow_{class} o_n}{(\eta,\iota)/\sigma \vdash \texttt{method(}I_1\texttt{@}CN_1,...,I_n\texttt{@}CN_n\texttt{) \{}B\texttt{\}} \Rightarrow (is, \{\})/\sigma}$$

where $n \geq 0$,
$is = \{((oid(o_1),...,oid(o_n)), \textbf{closure}((I_1,...,I_n), B, \kappa))\}$

[storage attribute]

$$\frac{\eta \vdash CN_1 \Rightarrow_{class} o_1, ..., \eta \vdash CN_n \Rightarrow_{class} o_n, (\eta,\iota)/\sigma \vdash E \Rightarrow v/\sigma'}{(\eta,\iota)/\sigma \vdash \texttt{storage(}I_1\texttt{@}CN_1,...,I_n\texttt{@}CN_n\texttt{) :=} E \Rightarrow (cs, cs)/\sigma''}$$

where $n \geq 0$,
$(l, \sigma'') = allocate(\sigma', \textbf{table}(\{\}, v))$,
$cs = \{((oid(o_1),...,oid(o_n)), \textbf{storage}(l))\}$

[acceptor attribute]

$$\frac{\eta \vdash CN_1 \Rightarrow_{class} o_1, ..., \eta \vdash CN_n \Rightarrow_{class} o_n}{(\eta,\iota)/\sigma \vdash \texttt{acceptor(}I_1\texttt{@}CN_1,...,I_n\texttt{@}CN_n\texttt{) :=} I\ \texttt{\{}B\texttt{\}} \Rightarrow (\{\}, ags)/\sigma}$$

where $n \geq 0$,
$ags = \{((oid(o_1),...,oid(o_n)), \textbf{closure}((I_1,...,I_n,I), B, \kappa))\}$

[combined -attribute]

$$\frac{\kappa/\sigma \vdash GF_1 \Rightarrow (is_1, ags_2)/\sigma_1, \kappa/\sigma_1 \vdash GF_2 \Rightarrow (is_2, ags_2)/\sigma_2}{\kappa/\sigma \vdash GF_1 \ \&\ GF_2 \Rightarrow (is, ags)/\sigma_2}$$

where $is = is_1 \uplus is_2$,
$ags = ags_1 \uplus ags_2$

Figure A-22: Dynamic semantics of generic function attributes.

A method attribute, of the form **`method(`***F**`){`***B***`}`**, produces a pair whose invocable set contains a single case. Its specializers are determined by the formals, *F**. The formals have the form *I*`@`*CN*, where *CN* names the formal's *specializer*. The invocable's specializer tuple is the tuple, in order, of the identities of these specializers.

The closure of this case, when called, will evaluate the block, *B*, in a context that extends the enclosing context by binding each formal directly to the given actual. (Details are given in the semantics of application expressions below.) For example, the following acts like the identity function of the λ-calculus.

```
method(x@any) {x}
```

A storage table attribute, of the form **`storage(`***F**`):=`***E**, produces a pair of sets, both of which contain the same case. The specializers of this case are determined by the formals, *F**, as above. The branch of the case is a storage table with the result of the initializer expression, *E*, as its default value. The initializer expression is first evaluated, then the location containing the table is allocated.

An acceptor attribute, of the form, **`acceptor(`***F**`):=`***I** **`{`***B***`}`**, produces a pair of sets whose assignable set contains a single case. The specializers of this case are determined by the formals, *F**, as above. The branch of the case is a closure. This closure has as its tuple of formals the identifiers in *F** followed by *I*.

[identifier expression]

$$\frac{\eta \vdash I \Rightarrow v}{(\eta,\iota)/\sigma \vdash I \Rightarrow v/\sigma}$$

[application expression]

$$\frac{\begin{array}{c}\kappa/\sigma \vdash E_0 \Rightarrow (id_0,\iota_0,is_0,ags_0)/\sigma_0,\\ \kappa/\sigma_0 \vdash AA_1 \blacktriangleright (v_1, ids_{dir,1})/\sigma_1, \ldots, \kappa/\sigma_{n-1} \vdash AA_n \blacktriangleright (v_n, ids_{dir,n})/\sigma_n,\\ (\iota_0, ((v_1,ids_{dir,1}),\ldots,(v_n,ids_{dir,,n}))) \vdash (ids,b)\ \textit{is-best-for}\ is_0,\\ \sigma_n \vdash \textit{invoke}(b, (v_1,\ldots,v_n)) \Rightarrow v/\sigma_{n+1}\end{array}}{\kappa/\sigma \vdash E_0(AA_1,\ldots,AA_n) \Rightarrow v/\sigma_{n+1}} \quad \text{where } n \geq 0$$

[assignment expression]

$$\frac{\begin{array}{c}\kappa/\sigma \vdash E_0 \Rightarrow (id_0,\iota_0,is_0,ags_0)/\sigma_0,\\ \kappa/\sigma_0 \vdash AA_1 \blacktriangleright (v_1, ids_{dir,1})/\sigma_1, \ldots, \kappa/\sigma_{n-1} \vdash AA_n \blacktriangleright (v_n, ids_{dir,n})/\sigma_n,\\ \kappa/\sigma_n \vdash E_{n+1} \Rightarrow v_{n+1}/\sigma_{n+1},\\ (\iota_0, ((v_1,ids_{dir,1}),\ldots,(v_n,ids_{dir,,n}))) \vdash (ids,b)\ \textit{is-best-for}\ ags_0\\ \sigma_{n+1} \vdash \textit{assign}(b, (v_1,\ldots,v_n), v_{n+1}) \Rightarrow \sigma_{n+2}\end{array}}{\kappa/\sigma \vdash E_0(AA_1,\ldots,AA_n) := E_{n+1} \Rightarrow (id_0,\iota_0,is_0,ags_0)/\sigma_{n+2}} \quad \text{where } n \geq 0$$

[sequence expression]

$$\frac{\kappa/\sigma \vdash E_1 \Rightarrow v_1/\sigma_1, \quad \kappa/\sigma_1 \vdash E_2 \Rightarrow v_2/\sigma_2}{\kappa/\sigma \vdash E_1 \ ; \ E_2 \Rightarrow v_2/\sigma_2}$$

Figure A-23: Dynamic semantics of expressions.

The closure of this case, when used in an assignment of a value to a key, will evaluate the block, *B*, in a context that extends the enclosing context by binding each formal directly to the corresponding key, and by binding the value identifier, *I*, to the value. (Details are given in the semantics of assignment expressions below.)

A combination attribute, of the form GF_1 **&** GF_2, produces a pair of invocable and assignable sets that contains the cases of GF_2 and those from GF_1 that do not clash with those in GF_2. (Recall that cases clash when they have the same specializers. Thus, if there are no clashes, then the cases of both generic functions are found in the resulting pair of sets.)

A.2.19 Expressions

The evaluation of an expression in an assumed context and store produces an expressible value and a new store. The formal semantics is given in Figure A-23.

An identifier used as an expression simply produces the value to which it is bound in the surrounding context. (See Figure A-24 for the rule used in the hypothesis.)

In an application, E_0(*AA**), first the operator, E_0, and then the *Actual-Argument*s, *AA**, are evaluated left-to-right. The *Actual-Argument*s each produce actual argument pairs, as described below, which are made into a tuple. The invocable set of the generic function denoted by E_0 is used to search for a unique most-specific applicable case. Note that the inheritance relation assumed for the search is ι_0, the inheritance relation of E_0. If is found, the most specific case's branch is invoked (using rules given in Figure A-6).

For example, the following block returns 27.

```
object f
f inherits GenericFun_rep
f has method(x@int_rep, y@int_rep) { y }
f(3, 27@int_rep)
```

In the following block, the application produces a “message ambiguous” error, because there are two applicable methods in the generic function being applied, and neither is more specific than the other.

```
object f
f inherits GenericFun_rep
f has method(x@int_rep, y@any) { x }
f has method(x@any, y@int_rep) { y}
f(3, 27)          -- message ambiguous error!
```

However, this ambiguity could be resolved by using directed actual arguments. For example, the following application would call the second method defined above.

```
f(3@any, 27)
```

An expression of the form E_0`(`AA*`):=`E_{n+1} is an assignment expression. It first evaluates the operator, E_0, then the operands, AA*, in left-to-right order, and then evaluates the desired value, E_{n+1}. The tuple of the operands’ identities forms a key. The key is used to search for a unique most-specific case in the assignable set of the generic function denoted by E_0. This unique most specific case is then called upon to perform the assignment. The result of the expression is the value of E_0.

As an example, the following expression returns 227. It does this by first updating the table for the key 2, this table is returned by the method in the generic function value of `first`, and it is then asked for the value of the table for that key (2) in the outermost expression, which is an application.

```
object my_table
my_table inherits field_rep
my_table has storage(x@int_rep) := 641
first has method() {my_table(2) := 227}
(first())(2)
```

An expression of the form E_1`;`E_2 evaluates the two expressions in sequence, and returns the value of the second one.

A.2.20 Identifiers

An identifier is looked up in the assumed environment, and its value is returned. The formal axiom is given in Figure A-24. Unlike class names, an identifier can denote either a class or an argument.

A.2.21 Actual Arguments

The evaluation of an actual argument in an assumed context and store produces an actual argument pair and a new store. Recall that an actual argument pair, written (v, ids_{dir}), consists of an expressible value, v, and a set of directed classes, ids_{dir}. The formal rule is given in Figure A-25.

For an undirected actual argument, of the form E, the expression is evaluated to produce an expressible value, v, and an actual argument pair is formed using the empty set for the set of directed classes.

For a directed actual argument, of the form E`@`CN*, E is evaluated to produce an expressible value, v. The actual argument pair is formed from v and the identities of the classes CN*.

[identifier 1] $\eta \vdash I \rightarrowtriangle o$ where $(I\!:\!\mathbf{class}(o)) \in \eta$

[identifier 2] $\eta \vdash I \rightarrowtriangle o$ where $(I\!:\!\mathbf{arg}(o)) \in \eta$

Figure A-24: Dynamic semantics of identifiers.

[undirected actual]
$$\frac{\kappa/\sigma \vdash E \rightarrowtriangle v/\sigma'}{\kappa/\sigma \vdash E \blacktriangleright (v, \{\}) / \sigma'}$$

[directed actual]
$$\frac{(\eta,\iota)/\sigma \vdash E \rightarrowtriangle v/\sigma', \quad \eta \vdash CN_1 \rightarrowtriangle_{class} o_1, ..., \eta \vdash CN_n \rightarrowtriangle_{class} o_n}{(\eta,\iota)/\sigma \vdash E@CN_1,...,CN_n \blacktriangleright (v, \{id_1,...,id_n\}) / \sigma'}$$
where $n \geq 0$, $id_1 = oid(o_1)$, ..., $id_n = oid(o_n)$

Figure A-25: Dynamic semantics of actual arguments.

Appendix B Design Decisions and Alternatives for the Untyped Subset of BeCecil

In this section we describe various design decisions and alternatives for the untyped subset of BeCecil.

B.1 Focus on Extensions, Nested Scopes, and Information Hiding

Although we are ultimately interested in module systems that achieve namespace control, information hiding, and that allow integration of independently developed code [Chambers & Leavens 95], we decided to separate concerns by focusing first on information hiding, and leaving namespace control and concerns about integration to a later stage of our investigation. Our earlier module system ideas seemed too complex to deal with in depth, if we also wanted to investigate the issues of extensibility and nested scopes. Nested scopes, or something like the hide declaration of BeCecil, are also an important part of the semantics of module systems. We do believe, however, that aside from the integration problems of modules, the addition of namespace control poses no fundamental complications.

B.2 Generic Functions are Named, not Methods

One thing we learned from the dynamic semantics, and from a study of Dylan, CLOS, and the λ&-calculus, is that the primary semantic entity is not a multimethod but a generic function. In BeCecil, an object is simply a generic function with a unique identity. The alternative view would be that multimethod cases themselves could be named; but this would mean that the same name would be used for different things in the same contour, or, more reasonably, that names for cases would consist of an identifier and a tuple of specializers. Let us call such a name a *message name*. However, if message names were used, then one would be able to call an individual method, using its message name, which would not take dynamic dispatch and inheritance into account.

Since it is possible to have a generic function containing a single method, there is no loss in expressive power in naming generic functions. Indeed, there is a gain in expressive power from using named generic functions. This gain comes because one can shadow an entire generic function by redeclaring its name in a nested contour, whereas to do the equivalent with message names would require several declarations (in general). Also, one would not have the same power by shadowing individual message names, because if one added more methods with the same name in a surrounding contour, then to achieve the same shadowing, one would have to write yet more declarations in the nested contour.

B.3 No Changes to Generic Function Values of Objects in Nested Contours

In BeCecil, one can extend the direct inheritance relation in any contour, but it seems that one cannot directly extend a generic function (by adding new cases) in a nested contour. This may prompt one to ask why the extension declarations cannot act on generic functions declared in surrounding contours.

The first answer is that one can achieve the same effect without changing the generic function values of objects defined in surrounding contours. That is, it is possible in a nested contour to make a new generic function with the same name as the one in the surrounding contour, but with some branches of its generic function replaced and new ones added. The following example shows how, in a nested block, to replace one method and add another to the generic function value of an object, `f`.

```
object f
f has method() {3}
f has method(x@int_rep) {4}
object nested
nested has method() {
   object f_new
   f_new has f & method(x@int_rep){5} -- replace one of f's methods
```

```
      f_new has method(y@float_rep}{6}    -- add yet another method
      object more_nested
      more_nested has method() {
         object f
         f has f_new
            f(f(3.7))                     -- returns 5
      }
      more_nested()
   }
   nested()
```

The second answer is that, in the semantics, one wants the generic function value of an object to be a property of the object itself, not a property of a contour. If the generic function value of an object were a property of a contour, that would permit each contour to extend the value by adding new branches, etc., but it would also make it difficult for a program to pass the generic function value of an object from one contour to another. So it seems desirable to make the generic function value of an object a property of the object, instead of its contour. (It still might be possible to allow mutation of the generic function value of an object, but we do not know how to type check such dynamic updates, and they seem hard to coordinate.)

B.4 Use of Declarations instead of Expressions

In BeCecil, there are as many declaration forms as expressions. We originally sought a core language with just expressions, but the division between declarations and expressions seems useful. The primary reason is that we wanted to avoid the complexities of dynamic inheritance. To do this, we wanted to have inheritance relationships be statically known. One way to do this would have been to make the declaration of parents part of the construction of objects (which still could have been an expression). But we also wanted to allow for inheritance relationships to be extensible, and if they were made part of object construction forms, then the inheritance relationships of objects would be fixed at the moment of their creation.* Declaration forms allow inheritance relationships to be made apart from the construction of objects, while ensuring that the requisite information is statically known.

Declaration forms also allow the definition of an object's methods to be textually spread throughout a program (as in the module system we hope to study), instead of having to be assembled by the programmer (as in the λ&-calculus). Declaration forms also avoid the problem of tracking negative information (as in extensible record calculi), which would be needed if an object's methods could be extended dynamically.

Finally, as can be seen from the patterns and sugars described in Section 2, there is no loss of expressiveness in using declaration forms, because of BeCecil's block structure.

B.5 Distinction between Class Names and Identifiers

In some places in the grammar for BeCecil, a class name, *CN*, is required instead of an identifier. The exact distinction is that a class name is required when the identity associated with the name must be statically known. Class names are required in the syntax in places to avoid the complexities of dynamic inheritance or dynamic object extensions. However, when an object is used for its generic function value, its identity does not need to be known; hence for identifiers are allowed to be used as generic function attributes.

* In BeCecil, the ancestry of an object is fixed by the inheritance relation of the recursive declaration sequence in which it is declared, so it is, in some sense, fixed at the time of the object's creation. However, it may yet participate in other inheritance relationships in nested contours.

B.6 Mutual Recursion in Declarations

The declarations in a block, and in the private part of a hide declaration in BeCecil, are allowed to be mutually recursive. One might consider that this is unnecessarily complex for a core language. In particular, because object declarations resemble "forward" declarations (as in Pascal), one might think that there is no absolute need for mutual recursion in declarations. In this section we discuss the reasons that led us to prefer mutual recursion in BeCecil, although the arguments are not indisputable.

One thing that is needed in BeCecil is the concept of a single "checkpoint" in each declaration sequence where type checking is done. This checkpoint is needed because each object's methods and storage tables have to be checked for consistency and completeness [Chambers & Leavens95], which can only be done once all the methods and storage tables associated with an object have been seen. For example, consider the following declaration sequence. This sequence will define a consistent set of methods for `addc`, but if one tried to check this after processing each declaration, then there would be points, such as the one marked with the comment, where the declarations processed thus far would be inconsistent (or incomplete, if one had type declarations).

```
object complex_rep inherits any
object cartesian_rep inherits complex_rep
object polar_rep inherits complex_rep
object addc inherits GenericFun_rep
addc has method(x@cartesian_rep, y@complex_rep) {...}
addc has method(x@complex_rep, y@polar_rep) {...}
-- the above is not consistent by itself
addc has method(x@complex_rep, y@complex_rep) {...}
addc has method(x@polar_rep, y@polar_rep) {...}
```

The above justifies the notion of a declaration sequence as a syntactic construct in BeCecil. Having such declaration sequences makes it easy for language designers to consider mutual recursion as a possibility. Allowing mutual recursion is convenient for programming, in that it allows declarations to be introduced in any order. But that alone is not a sufficient justification for the use mutual recursion in a core language like BeCecil, considering that the semantics is more complex than it would be otherwise. However, there is an additional reason for mutual recursion, namely that it is needed if hide declarations are to adequately mimic the encapsulation aspects of modules. The problem is that in a hide declaration the hidden declarations come before the declarations that are not hidden; thus mutual recursion is especially convenient, at least for the hide declaration, to ensure that the methods in each part of a hide declaration can call those in the other part. While it is possible to put object declarations for the generic functions that are not to be hidden before the hide declaration, doing so would not allow a single hide declaration to model a single module. (Rearranging the hide declaration so that the non-hidden part was elaborated first would not allow public declarations to depend on the hidden ones at all, unless the hide declaration was mutually recursive.)

Since it is sensible to allow mutual recursion in declaration sequences, and since it is especially convenient for hide declarations, and since Cecil itself allows mutual recursion, we decided to put up with the extra complexity involved.

B.7 The Model of Objects

We considered several models for the dynamic semantics of objects in BeCecil. The current semantics is that an object is a generic function (a set of invocables, a set of assignables, and an inheritance relation) that has a unique identity.

A fundamentally different semantics would have been to have two different kinds of values: objects and generic functions. In this case objects would not have any value at all, aside from their identity. However,

unless generic functions were treated differently than objects, there would be no point in making the distinction. For example, one could say that generic functions did not inherit from any objects, which would prohibit them from being stored or passed around. But if that were the case, the language would be rigidly first-order, and other constructs would have to be added to handle control structures, etc.

At various points, we considered using the store to track various parts of objects: their ancestry, their generic function cases, or both. However, since these aspects of objects do not change once all the declarations in recursive declaration sequence are processed, there is no reason to use the store for these aspects of objects. On the other hand, if one had wanted a semantics in which method cases could be replaced at run-time, then it would be necessary to keep the generic function cases of an object in the store. With storage tables, there is not a great need for method replacement (at least in a theoretical sense), because storage tables can contain generic functions, but this is an avenue we did not explore. (See Section B.10 for more about storage tables.)

B.8 Inheritance

We originally experimented with a language in which inheritance relationships were stored in objects, and could be changed at run-time. However, this kind of dynamic inheritance has well-known problems for type checking. In the context of BeCecil, it would mean that implementation-side type checking would be impossible to perform statically in a reasonable fashion. Hence inheritance in BeCecil is statically declared.

Notice that, in the semantics for applications, the inheritance relation of the calling contour (the contour where the text of the call appears) plays no role. For example, in the following, the inheritance relationship declared in the function `nested3` does not affect the application of the value of `g` to the value of `x`. The only inheritance relations used are those recorded in `x` (which determines its ancestry) and in `g` (which determines the applicability and relative specificity of its cases.

```
var g := ...
var x := ...
fun nested1() { ... g() := ... }
fun nested2() { ... x() := ... }
fun nested3() {
   foo_rep inherits baz_rep
   g()(x())
}
nested1();
nested2();
nested3()
```

An alternative semantics would be to union the inheritance relation of the actual arguments with the calling contour's inheritance relation, for purposes of determining each object's ancestors. The problem is that this would allow an object to be passed as an actual argument to a method, but inside of the method the object would no longer inherit from the specializer given for its formal argument.

Another alternative would be to have some more complex way of combining the inheritance relations of the actual arguments with those of the calling contour. For example, one might take the ancestors of the actual that are known in the calling contour, and all of their ancestors according to the calling contour's inheritance relation. However, this would potentially result in a large amount of information loss versus the current semantics, as the calling contour would be filtering out information about objects and inheritance relationships that both actual arguments and the generic function being called could know about. We leave the investigation of such combinations as an area for future work.

B.9 Extension vs. Inheritance

The notion of extending an object's generic function cases is an orthogonal notion from inheritance, which gives BeCecil a flexibility that is appropriate for a core language. For example, one can either copy cases from several other objects without saying which ones have preference (by using two separate extension declarations), or have some methods shadow others (by using a single declaration with a combination attribute). Combination attributes are as in the λ&-calculus, but can involve named objects. This additional flexibility is useful to allow renaming of generic functions, and explicit combinations of named entities.

B.10 Storage Tables

Storage tables are a generalization of variables, fields, and arrays. However, storage tables are somewhat complex for a core language, and so we have considered several alternatives for them.

One alternative would be to replace storage tables with the simpler constructs that they generalize: local variables, fields (instance variables), and arrays. This is what is done, for example, in Smalltalk and Cecil itself. However, this does not seem to simplify things to any great extent in terms of describing the core language.

Another alternative, which we considered at some length, would be to delete storage tables from BeCecil, and to instead use method replacement, as in Abadi and Cardelli's Theory of Objects [Abadi & Cardelli 96]. This would require a semantic change, in that the cases of an object's generic function value would have to be found by the use of a location and the store, making the semantics of recursive environments more complex. We consider the relevant syntactic sugars below to give the idea of how this alternative would work out.

By itself, method replacement would take the place of variables. For example, one would have the following syntactic sugar for the declaration of a variable using this alternative. In the sugars discussed below, we have tried to keep the meaning as close as possible to the sugars in Section 2.3, to allow better comparison. Note, however, that in this sugar, the initialization expression is not eagerly evaluated, as it was previously.

$$\lceil \texttt{var } I \texttt{ := } E \rceil \equiv \lceil \texttt{gf } I \rceil \; I_1 \texttt{ has method()\{} \lceil E \rceil \texttt{\}}$$

In this alternative, the sugar for assignment to a variable, given below, uses **&:=** as the primitive operator for replacing a method. To do such a method replacement, in general, it would be necessary to have exact information about the specializers of an object's methods. However, since a zero-argument method has no specializers, one could, as a special case, allow such updates to arbitrary expressions, even if exact information about specializers was not known.

$$\lceil E_{var} \texttt{() := } E_{val} \rceil \equiv \lceil \texttt{let } I_{val} \texttt{ = } E_{val} \texttt{ in } E_{var} \texttt{ \&:= method()\{} I_{val} \texttt{\}} \rceil, \quad \text{where } I_{val} \text{ is a fresh identifier.}$$

In this alternative, to model fields (and more interesting kinds of storage tables), BeCecil would also need a primitive that tested equality of object identities. If that were done, then some model of the booleans would have to be built-in to BeCecil. Adding the booleans to BeCecil's initial context would be only a small complication, however. The following would be the appropriate syntactic sugar for declaring a field of an object.

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

The syntactic sugar for field updates would probably be something like the following, in which **eq** is used as the equality primitive on objects (a test of object identities).

⌈ E_{fld}**(**E_{obj}@CN_{obj}**):=** E_{val} ⌉ ≡
    ⌈ **let** I_{obj} **=** E_{obj}**;** I_{val} **=** E_{val}
     in {
          gf old_fld
          old_fld **has** E_{fld}
          fun new_fld**(***x*@CN_{obj}**) {** if(**eq**(*x*,I_{obj}),**[**I_{val}**]**,**[**old_fld(*x*)**]**) **}**
          E_{fld} **&:=** new_fld**;**
          E_{fld}
       } ⌉, where *x*, I_{obj}, and I_{val} are fresh identifiers.

This sugar is somewhat more clumsy than the assignment primitive in BeCecil, because the sugar requires the class name CN_{obj} to be given. This class name is needed as the specializer for function `new_fld` that is used in the sugar, as otherwise it would not be clear which method of E_{fld} should be replaced. Furthermore, unlike storage tables, the type system would require exact information about the set of cases for E_{fld} to ensure that this was really a replacement, which would make the BeCecil a bit less flexible. However, it seems that this alternative would work, and we leave its exploration as future work.

B.11 Initialization of Storage

Many languages do not allow initialization of variables in declaration forms, or (as in Modula-3), interpret such initializations as syntactic sugar for code that is to be executed at the beginning of a block, before the usual code is executed. This was not an important semantic point for us, but we believe that all storage should be initialized, if nothing else, to avoid dealing with nonproper values in the semantics, or having to allocate storage before it is initialized.

B.12 Copying of Storage Tables during Extensions

Consider the following example declarations.

```
object x
x has storage() := 3
object y
y has x
```

What should be the meaning of these declarations? In the current semantics, the storage table allocated for `x` is copied when the identifier `x` is used as a generic function attribute for `y`. Thus in the current semantics, `y()` and `x()` are not aliases. For example, after executing `y():=4`, the value of `x()` would still be 3. An alternative semantics would be to make `y` have the same storage tables as `x`, without allocating a new location for `y`'s storage table. This alternative would make `x()` and `y()` aliases.

Although preventing aliasing may make programs easier to reason about, the current semantics may also make BeCecil less expressive. We believe that there are times when, in a nested contour, one wants to make a customized version of a generic function that shares storage tables with the original. But as yet we have too little experience with this aspect of BeCecil to make a firm decision about the semantics. It may turn out that, in a real language based on BeCecil, one needs two primitives, one with each semantics.

B.13 No Linearization

In BeCecil, as in Cecil but unlike CLOS or Dylan, there is no linearization of the inheritance relationship, and thus the ordering imposed on a set of applicable cases is not total. This results in the language recognizing as errors situations that would not be errors in CLOS or Dylan. In particular, when there is more than one method that is applicable to a given tuple of actual arguments, there are situations in which the CLOS or Dylan ordering would pick one method out of a set of applicable methods that, according to the BeCecil or Cecil semantics, are mutually incomparable (i.e., none overrides the others).

First, it should be noted that the BeCecil semantics could easily be revised to accommodate linearizations, as in CLOS or Dylan, by redefining the extension of an inheritance ordering to cases. We did not choose to do this, however, because we are philosophically opposed to such linearizations. That is, we believe that no one fixed ordering of such (for us) incomparable cases will always be right for all situations. If one accepts that premise, then one concludes that the language might, at least warn the user about all such situations, or provide a tool to find them. From that point of view, our type system is a tool that can be used to give such warnings or find such situations.

Appendix C Static Semantics

C.1 Domains for the Static Semantics

The static semantics (type system) of BeCecil is mainly concerned with the following domains: classes, named types, type attributes, inheritance relations, conformance relations, subtype relations, type contexts, and type elaborateds. *Classes* are objects that are statically known (i.e., named in an object declaration), as opposed to formal arguments.

The domains used in the static semantics are summarized in Figure C-1. The sequents used in the static semantics are summarized in Figures C-2 and C-16. In what follows we describe the semantic domains that have not already been discussed, and then the type checking rules.

cl	∈	*Class*	= **class**(*ObjectId*)
id	∈	*ObjectId*	= *Nat*
ids	∈	*ObjectIdSet*	= *Powerset*(*ObjectId*)
idv	∈	*ObjectIdVec*	= *ObjectId**
μ	∈	*NamedType*	= **namedtype**(*TypeId*)
tid	∈	*TypeId*	= *Nat*
π	∈	*TypeEnv*	= *FiniteFun*[*Identifier, TypeDenotable*]
θ	∈	*TypeDenotable*	= *Class* + **formal**(*TypeAttribute*) + *NamedType*
τ	∈	*TypeAttribute*	= *NamedType* + *InvocableType* + *AssignableType* + *ExactType* + *ConjunctiveType* + *DisjunctiveType*
τv	∈	*TypeAttributeVec*	= *TypeAttribute**
it	∈	*InvocableType*	= **arrow**(*TypeAttribute** × *TypeAttribute*)
at	∈	*AssignableType*	= **assignable**(*TypeAttribute** × *TypeAttribute*)
et	∈	*ExactType*	= **exact**(*InvokeCaseSet* × *AssignCaseSet*)
ics	∈	*InvokeCaseSet*	= *Powerset*(*ExactCase*)
acs	∈	*AssignCaseSet*	= *Powerset*(*ExactCase*)
ec	∈	*ExactCase*	= *ObjectId** × *ExactBranch*
ecs	∈	*ExactCaseSet*	= *Powerset*(*ExactCase*)
eb	∈	*ExactBranch*	= *InvocableType* + *AssignableType*
α	∈	*ConjunctiveType*	= **andtype**(*PowerSet*(*TypeAttribute*))
δ	∈	*DisjunctiveType*	= **ortype**(*PowerSet*(*TypeAttribute*))
ζ	∈	*Subtypes*	= *FiniteRel*[*TypeId, TypeAttribute*]
ι	∈	*Inherits*	= *FiniteRel*[*ObjectId, ObjectId*]
χ	∈	*Conforms*	= *FiniteRel*[*ObjectId, TypeAttribute*]
K	∈	*TypeContext*	= *TypeEnv* × *Inherits* × *ClassInfo* × *Conforms* × *Subtypes* × *TypeInfo*
ϕ	∈	*ClassInfo*	= *FiniteFun*[*ObjectId, Inherits*]
ψ	∈	*TypeInfo*	= *FiniteFun*[*TypeId, Inherits* × *Powerset*(*ObjectId*)]
ξ	∈	*TypeElaborated*	= *TypeEnv* × *Inherits* × *HasTypes* × *Conforms* × *Subtypes*
H	∈	*HasTypes*	= *FiniteFun*[*ObjectId, InvokeCaseSet* × *AssignCaseSet*]
ta	∈	*TypedActual*	= *TypeAttribute* × *Powerset*(*ObjectId*)
aa	∈	*ActualArgPair*	= *ObjectId* × *Powerset*(*ObjectId*)
aav	∈	*ActualArgPairVec*	= *ActualArgPair**

Figure C-1: Domains for the static semantics of BeCecil. The domains *FiniteFun*, *FiniteRel*, and operations on them are described in Appendix E.

	type			
prototype sequent	assumes (inherits)	works on (arguments)	produces (synthesizes)	defined in Figure
$\iota \vdash id_1 \leq_{inh} id_2$	*Inherits*	(*ObjectId* + **wild**) × *ObjectId*		A-3, C-20
$\pi \vdash \tau$ *is-well-formed*	*TypeEnv*	*TypeAttribute*		C-4, C-5
$\zeta \vdash \tau_1 \leq_{sub} \tau_2$	*Subtypes*	*TypeAttribute* × *TypeAttribute*		C-6
$(\chi,\zeta) \vdash id <: \tau$	*Conforms* × *Subtypes*	(*ObjectId* + **wild**) × *TypeAttribute*		C-8, C-21
$(\chi,\zeta) \vdash$ *id one-exact-type*	*Conforms* × *Subtypes*	*ObjectId*		C-9
$\iota \vdash ec_1 \leq_{inh} ec_2$	*Inherits*	*ExactCase* × *ExactCase*		C-10
$\iota \vdash ec_1$ *overrides* ec_2	*Inherits*	*ExactCase* × *ExactCase*		C-11
$\iota \vdash$ *ec applies-to aav*	*Inherits*	*ExactCase* × *ActualArgPair**		C-12
$(\iota,$ *aav*$) \vdash$ *ec is-best-for ecs*	*Inherits* × *ActualArgPair**	*ExactCase* × *ExactCaseSet*)		C-13
$(\chi,\zeta,\iota) \vdash$ *ids type-info-is* ψ	*Conforms* × *Subtypes* × *Inherits*	*Powerset*(*ObjectId*)	*TypeInfo*	C-15

Figure C-2: Table of auxiliary sequents for the static semantics. A few helping definitions have been omitted. Sequents that define the static semantics of each category in the BeCecil grammar can be found in Figure C-16. For each kind of sequent, the types it relates are shown in three parts in its row: what it assumes, what it works on (or relates) and what it produces. Sequents that do not produce any result can be thought of as producing a boolean result (true if it is provable), or as relations on their arguments.

C.2 Static Semantics Domains and Auxiliary Functions

This section explains the domains in the static semantics, and describes various auxiliary functions used in the type checking rules.

C.2.1 Type Denotables, Class and Type Identities

A *type denotable* records information about classes, formal arguments, and named types.

As far as the type system is concerned, both a *class* and a *named type* have only a single intrinsic attribute: their identity. No two classes or named types have the same identity.

Syntax (T)	Semantics (τ)
TN	**namedtype**(*tid*)
(T_1,...,T_n **) -> ** T_{n+1}	**arrow**((τ_1,...,τ_n), τ_{n+1})
(T_1,...,T_n **) := ** T_{n+1}	**assignable**((τ_1,...,τ_n), τ_{n+1})
exact{(*CN* **:** *T* **)->** *T* **,(** *CN* **:** *T* **):=** *T* **}**	**exact**({((*id*),**arrow**((τ),τ))},{((*id*),**assignable**((τ),τ))})
T_1 **&** T_2	**andtype**({τ_1, τ_2})
T_1 **|** T_2	**ortype**({τ_1, τ_2})
(CN_1**:** T_1,...,CN_n**:** T_n **) -> ** T_{n+1}	((id_1,...,id_n), **arrow**((τ_1,...,τ_n), τ_{n+1}))
(CN_1**:** T_1,...,CN_n**:** T_n **) := ** T_{n+1}	((id_1,...,id_n), **assignable**((τ_1,...,τ_n), τ_{n+1}))

Figure C-3: Correspondence between syntax for types and type domains.

C.2.2 Type Environments

A *type environment* is a finite function from identifiers to type denotables. We use the same notation for type environments as for environments. (See Section E.3 for details.) It will sometimes be useful to extract the class and type name parts of a type environment separately.

classes: *TypeEnv* → *FiniteFun*[*Identifier*, *ObjectId*]
classes(π) = {(*I*:*id*) | (*I*:**class**(*id*)) ∈ π}
types: *TypeEnv* → *FiniteFun*[*Identifier*, *TypeId*]
types(π) = {(*I*:*tid*) | (*I*:**namedtype**(*tid*)) ∈ π}

C.2.3 Models of Types

Because type names (and classes) have unique identities, the domains used to model types in the static semantics are not the same as the syntax of types. The correspondence is given in Figure C-3.

The identities in a type attribute cannot be random. Instead, they must be object and type identities that are known in a given type environment. The rules that check this well-formedness condition for type attributes are given in Figures C-4 and C-5. These rules are used to check that type of a block is well-formed with respect to the type environment in which the block is checked (see Section C.3.12).

C.2.4 Direct Subtype and Subtype Relations

A *direct subtype relation*, ζ, is a binary relation that relates the identities of named types to type attributes. It models information from **`subtypes`** declarations in BeCecil.

For example, if `Larch` denotes **namedtype**(tid_L) and if `SpecLang` denotes **namedtype**(tid_S), then the declaration below would be recorded by the direct subtype relation {(tid_L, **namedtype**(tid_S))}.

```
Larch subtypes SpecLang
```

The static semantics extends a direct subtype relation, ζ, to be a reflexive and transitive relation, $\leq_{sub}$, on type attributes, as shown in Figure C-6. We also extend a subtype relation to relate tuples pointwise in Figure C-7. The rule [sub-gf] is taken from the λ&-calculus [Ghelli 91, Castagna *et al.* 92, Castagna *et al.* 95]. The rule [sub-arrow] was originally described by Cardelli [Cardelli 88].

[wf-namedtype] $\pi \vdash \textbf{namedtype}(tid)$ *is-well-formed* where $tid \in range(types(\pi))$

[wf-arrow] $$\frac{\pi \vdash \tau_1 \text{ is-well-formed}, ..., \pi \vdash \tau_n \text{ is-well-formed}, \pi \vdash \tau_r \text{ is-well-formed}}{\pi \vdash \textbf{arrow}((\tau_1,...,\tau_n),\tau_r) \text{ is-well-formed}}$$ where $n \geq 0$

[wf-assignable] $$\frac{\pi \vdash \tau_1 \text{ is-well-formed}, ..., \pi \vdash \tau_n \text{ is-well-formed}, \pi \vdash \tau_r \text{ is-well-formed}}{\pi \vdash \textbf{assignable}((\tau_1,...,\tau_n),\tau_r) \text{ is-well-formed}}$$ where $n \geq 0$

[wf-andtype] $$\frac{\pi \vdash \tau_1 \text{ is-well-formed}, ..., \pi \vdash \tau_n \text{ is-well-formed}}{\pi \vdash \textbf{andtype}(\{\tau_1,...,\tau_n\}) \text{ is-well-formed}}$$ where $n \geq 1$

[wf-ortype] $$\frac{\pi \vdash \tau_1 \text{ is-well-formed}, ..., \pi \vdash \tau_n \text{ is-well-formed}}{\pi \vdash \textbf{ortype}(\{\tau_1,...,\tau_n\}) \text{ is-well-formed}}$$ where $n \geq 1$

[wf-exact] $$\frac{\pi \vdash ec_1 \text{ is-well-formed}, ..., \pi \vdash ec_n \text{ is-well-formed}, \pi \vdash ec'_1 \text{ is-well-formed}, ..., \pi \vdash ec'_m \text{ is-well-formed}}{\pi \vdash \textbf{exact}(\{ec_1,...,ec_n\},\{ec'_1,...,ec'_m\}) \text{ is-well-formed}}$$ where $n \geq 0$, $m \geq 0$

Figure C-4: Rules for well-formedness of types. These depend on the rules given below.

[wf-ec-arrow] $$\frac{\pi \vdash \textbf{arrow}((\tau_1,...,\tau_n),\tau_r) \text{ is-well-formed}}{\pi \vdash ((id_1,...,id_n),\textbf{arrow}((\tau_1,...,\tau_n),\tau_r)) \text{ is-well-formed}}$$ where $n \geq 0$, $(\forall i \,.\, i \in \{1,...,n\} \Rightarrow id_i \in range(classes(\pi)))$

[wf-ec-assignable] $$\frac{\pi \vdash \textbf{assignable}((\tau_1,...,\tau_n),\tau_r) \text{ is-well-formed}}{\pi \vdash ((id_1,...,id_n),\textbf{assignable}((\tau_1,...,\tau_n),\tau_r)) \text{ is-well-formed}}$$ where $n \geq 0$, $(\forall i \,.\, i \in \{1,...,n\} \Rightarrow id_i \in range(classes(\pi)))$

Figure C-5: Rules for well-formedness of exact cases.

[subtypes-tuple] $$\frac{\zeta \vdash \tau_1 \leq_{sub} \tau'_1, ..., \zeta \vdash \tau_n \leq_{sub} \tau'_n}{\zeta \vdash (\tau_1,...,\tau_n) \leq_{sub} (\tau'_1,...,\tau'_n)}$$ where $n \geq 0$

Figure C-7: Rule for pointwise subtyping of tuples.

[sub-base] $\zeta \vdash \textbf{namedtype}(tid) \leq_{sub} \tau$ where $(tid,\tau) \in \zeta$

[sub-refl] $\zeta \vdash \tau \leq_{sub} \tau$

[sub-trans] $$\frac{\zeta \vdash \tau_1 \leq_{sub} \tau_2, \quad \zeta \vdash \tau_2 \leq_{sub} \tau_3}{\zeta \vdash \tau_1 \leq_{sub} \tau_3}$$

[sub-and1] $$\frac{\zeta \vdash \tau \leq_{sub} \tau_1, \ldots, \zeta \vdash \tau \leq_{sub} \tau_n}{\zeta \vdash \tau \leq_{sub} \textbf{andtype}(\{\tau_1,\ldots,\tau_n\})}$$ where $n \geq 0$

[sub-and2] $\zeta \vdash \textbf{andtype}(\{\tau_1,\ldots,\tau_n\}) \leq_{sub} \tau_i$ where $n \geq 0$, $\tau_i \in \{\tau_1,\ldots,\tau_n\}$

[sub-or1] $$\frac{\zeta \vdash \tau_1 \leq_{sub} \tau, \ldots, \zeta \vdash \tau_n \leq_{sub} \tau}{\zeta \vdash \textbf{ortype}(\{\tau_1,\ldots,\tau_n\}) \leq_{sub} \tau}$$ where $n \geq 0$

[sub-or2] $\zeta \vdash \tau_i \leq_{sub} \textbf{ortype}(\{\tau_1,\ldots,\tau_n\})$ where $n \geq 0$, $\tau_i \in \{\tau_1,\ldots,\tau_n\}$

[sub-arrow] $$\frac{\zeta \vdash \tau'_1 \leq_{sub} \tau_1,\ldots, \zeta \vdash \tau'_n \leq_{sub} \tau_n, \quad \zeta \vdash \tau_r \leq_{sub} \tau'_r}{\zeta \vdash \textbf{arrow}((\tau_1,\ldots,\tau_n),\tau_r) \leq_{sub} \textbf{arrow}((\tau'_1,\ldots,\tau'_n),\tau'_r)}$$ where $n \geq 0$

[sub-assignable] $$\frac{\zeta \vdash \tau'_1 \leq_{sub} \tau_1,\ldots, \zeta \vdash \tau'_n \leq_{sub} \tau_n, \quad \zeta \vdash \tau'_r \leq_{sub} \tau_r}{\zeta \vdash \textbf{assignable}((\tau_1,\ldots,\tau_n), \tau_r) \leq_{sub} \textbf{assignable}((\tau'_1,\ldots,\tau'_n), \tau'_r)}$$ where $n \geq 0$

[sub-exact-translate] $\zeta \vdash \textbf{exact}(\{(idv_1,it_1),\ldots,(idv_n,it_n)\}, \{(idv'_1,at_1),\ldots,(idv'_m,at_m)\}) \leq_{sub} \textbf{andtype}(\{it_1,\ldots,it_n, at_1,\ldots,at_m\})$ where $n \geq 0$, $m \geq 0$

[sub-gf] $$\frac{\zeta \vdash it_{f(1)} \leq_{sub} it'_n, \ldots, \zeta \vdash it_{f(n)} \leq_{sub} it'_n, \quad \zeta \vdash at_{g(1)} \leq_{sub} at'_1, \ldots, \zeta \vdash at_{g(o)} \leq_{sub} at'_o}{\zeta \vdash \textbf{andtype}(\{it_1,\ldots,it_k, at_1,\ldots,at_l, \tau_1,\ldots,\tau_m\}) \leq_{sub} \textbf{andtype}(\{it'_1,\ldots,it'_n, at'_1,\ldots,at'_o, \tau_1,\ldots,\tau_m\})}$$ where $k \geq 0$, $l \geq 0$, $m \geq 0$, $n \geq 0$, $o \geq 0$, $(\forall i \in \{1,\ldots,n\} . (\exists f(i) \in \{1,\ldots,k\} . it'_i = \textbf{arrow}(\tau v_i,\tau_{ir})$ and $it_{f(i)} = \textbf{arrow}(\tau v_{f(i)},\tau_{f(i)r}))$, $(\forall j \in \{1,\ldots,o\} . (\exists g(j) \in \{1,\ldots,l\} . at'_j = \textbf{assignable}(\tau v'_j,\tau'_{jr})$ and $at_{g(j)} = \textbf{assignable}(\tau v'_{g(j)},\tau'_{g(j)r})))$

Figure C-6: Axioms and inference rules for the subtype relation.

C.2.5 Direct Conformance and Conformance Relations

A direct conformance relation is a binary relation that relates object identities to type attributes. It models information from **`conforms`** declarations in BeCecil.

$$\text{[conforms]} \qquad \frac{\zeta \vdash \tau' \leq_{sub} \tau}{(\chi,\zeta) \vdash id <: \tau} \qquad \text{where } (id,\tau') \in \chi$$

Figure C-8:Inference rule for conformance.

$$\text{[one-exact]} \qquad (\chi,\zeta) \vdash id \textit{ one-exact-type} \qquad \text{where } s = \{(ics,acs) \mid (\chi,\zeta) \vdash id <: \textbf{exact}(ics,acs)\},\ |s| = 1$$

Figure C-9: Rule for checking that a class has just one exact type.

For example, if `Larch_rep` denotes **class**(id_L) and if `Larch` denotes the type attribute μ_L, then the declaration below would be recorded by the direct conformance relation $\{(id_L, \mu_L)\}$.

```
Larch_rep conforms Larch
```

Figure C-8 extends a direct conformance relation, χ, to a relation, <:, that takes subtyping into account.

C.2.6 Exact Types for Generic Functions and Exact Cases

An *exact type* contains all the information about an object's generic function value. That is, an object *o* conforms to an exact type, *et*, just when the object's generic function value has all the invocable and assignable cases described in *et*, and no additional cases. Because this information is exact, from *o* <: *et*, one can conclude that some cases are not present in the generic function value of *o*. This kind of negative information cannot be obtained from inexact types, because subtyping can be used to forget the types of some cases. Because of this requirement, exact types do not have interesting subtypes. Since a given object conforms to only one exact type, the following declarations are in error.

```
-- at least one of these is wrong
foo conforms exact{(int_rep:int)->int}
foo conforms exact{(int_rep:int)->int, (myclass:mytype)->foo}
```

The rule for checking that an object conforms to just one each type is given in Figure C-9.

As noted above, a programer would not normally declare conformance to an exact type, since the exact type of an object's generic function value is inferred by the type system.

An exact type is composed of a pair of sets: an *invocable case set* and an *assignable case set*. Each is a set of *exact cases*, which are contain a tuple of specializers and an *exact branch*. An exact branch is either an invocable type or an assignable type. In the invocable case set the exact branches are all for invocable types, and similarly in the assignable case set the exact branches are all assignable types. Given an exact case, *ec*, we write *specializers*(*ec*) for the tuple of specializers of *ec*, *argtypes*(*ec*) for the tuple of argument types of *ec*, and *restype*(*ec*) for the result type of *ec*.

specializers: ExactCase → *ObjectId**
specializers((id_1,...,id_m), *eb*)) = (id_1,...,id_m)

argtypes: ExactCase → *TypeAttribute**
argtypes((id_1,...,id_m), **arrow**(τv,τ_r))) = τv
argtypes((id_1,...,id_m), **assignable**(τv,τ_r))) = τv

restype: ExactCase → *TypeAttribute*
restype((id_1,...,id_m), **arrow**(τv,τ_r))) = τ_r
restype((id_1,...,id_m), **assignable**(τv,τ_r))) = τ_r

[inh-eb] $$\frac{\iota \vdash id_{1,1} \leq_{inh} id_{2,1},\ ...,\ \iota \vdash id_{1,n} \leq_{inh} id_{2,n}}{\iota \vdash ((id_{1,1},...,id_{1,n}), eb_1) \leq_{inh} ((id_{2,1},...,id_{2,n}), eb_2)}$$ where $n \geq 0$

Figure C-10: Rule for extending a direct inheritance relation to an ordering on exact cases.

[overrides-eb] $$\frac{\iota \vdash ec_1 \leq_{inh} ec_2}{\iota \vdash ec_1 \text{ overrides } ec_2}$$ where $n \geq 0$, $specializers(ec_1) \neq specializers(ec_1)$

Figure C-11: Rule for when one exact case overrides another.

As for cases in the dynamic semantics, we extend an inheritance relation to a partial order on exact cases by comparing their specializers pointwise. The rule for this relation is given in Figure C-10. The rule for when one exact case overrides another is also analogous to the rule for invocables. It is given in Figure C-11.

As with invocables, two exact cases *clash* when they have the same tuples of specializers. We also give analogous definitions for the disjoint union ($\cup$) and overriding ($\uplus$) notations for sets of exact cases.

nonclashing: Powerset(*ExactCase*) × *Powerset*(*ExactCase*) → *Boolean*
nonclashing(ecs_1, ecs_2) = ((*specializers*(ecs_1) ∩ *specializers*(ecs_2)) = {})

$\cdot \cup \cdot$*: Powerset*(*ExactCase*) × *Powerset*(*ExactCase*) → *Powerset*(*ExactCase*)$_\perp$

$ecs_1 \cup ecs_2$ = **if** *nonclashing*(ecs_1, ecs_2) **then** $ecs_1 \cup ecs_2$ **else** $\perp$

$\cdot \uplus \cdot$*: Powerset*(*ExactCase*) × *Powerset*(*ExactCase*) → *Powerset*(*ExactCase*)
$ecs_1 \uplus ecs_2 = ecs_2 \cup \{ec \mid ec \in ecs_1, specializers(ec) \notin specializers(ecs_2)\}$

The rules for when an exact case applies to a tuple of actual argument pairs are given in Figure C-12; this is nearly identical to a similar rule in the dynamic semantics, and uses a helping definition from the dynamic semantics. Also similar to the dynamic semantics are the rules for finding the most specific exact case in a set of exact cases; these rules are given in Figures C-13 and C-14.

C.2.7 Type Contexts, Type Information, and Type Environments

A *type context* records information from declarations that can be used in type checking. A type context is modeled as a tuple of a type environment, a direct inheritance relation, some information about classes, a direct conformance relation, a direct subtype relation, and some information about type names. We sometimes use the following auxiliary functions to extract parts of a type context.

tenv: *TypeContext* → *TypeEnv*
tenv(π, ι, ϕ, χ, ζ,ψ) = π

directSubtypes: *TypeContext* → *Subtypes*
directSubtypes(π, ι, ϕ, χ, ζ,ψ) = ζ

Recall, from the informal discussion in Section 3, that the information kept for each type name is the inheritance relation of the recursive declaration sequence in which the name was declared, and the set of objects that are known to conform to it in the recursive declaration sequence in which it was declared. The way in which the type information for a set of declared classes is formed from a direct conformance relation, a direct subtypes relation, and a direct inheritance relation is given in Figure C-15.

Similarly, for each class, the inheritance relation of the recursive declaration sequence in which it was declared is kept.

$$\text{[applies-to]}\quad \frac{\iota \vdash id_{1,a} \leq_{inh} id_1\ ,\ ...,\ \iota \vdash id_{n,a} \leq_{inh} id_n,\quad \iota \vdash ids_{dir,1}\ ok\text{-}dir\ id_1\ ,\ ...,\ \iota \vdash ids_{dir,n}\ ok\text{-}dir\ id_n,}{\iota \vdash ((id_1,...,id_n),\ eb)\ applies\text{-}to\ ((id_{1,a},ids_{dir,1}),\ ...,\ (id_{n,a},ids_{dir,n}))} \quad \text{where } n \geq 0$$

Figure C-12: Rule for when an exact case applies to a tuple of typed actuals. See Figure A-12 for the rules for *ok-dir*.

$$\text{[is-best-for]}\quad \frac{(\iota,\ aav) \vdash ec\ better\text{-}than\ ec_1,\quad ...,\quad (\iota,\ aav) \vdash ec\ better\text{-}than\ ec_n}{(\iota,\ aav) \vdash ec\ is\text{-}best\text{-}for\ ecs} \quad \begin{array}{l}\text{where } ecs' = \{ec' \mid ec' \in ecs,\ \iota \vdash ec\ applies\text{-}to\ aav\},\\ ec \in ecs',\\ \{ec_1,\ ...,\ ec_n\} = ecs',\\ n \geq 1\end{array}$$

Figure C-13: Rule for when an exact case is best in a set for a given inheritance relation and assumed tuple of typed actuals.

$$\text{[better-than-reflex]}\quad (\iota,\ aav) \vdash ec\ better\text{-}than\ ec$$

$$\text{[better-than]}\quad \frac{\iota \vdash ec\ overrides\ ec'}{(\iota,\ aav) \vdash ec\ better\text{-}than\ ec'} \quad \text{where } ec \neq ec'$$

Figure C-14: Axiom and rule for when one actual is better than others. Note that these are mutually exclusive.

$$\text{[type-info-is]}\quad (\chi,\zeta,\iota) \vdash \{id_1,...,id_n\}\ type\text{-}info\text{-}is\ \psi \quad \begin{array}{l}\text{where } n \geq 0,\\ (\forall i\ .\ i \in \{1,...,n\} \Rightarrow\\ \quad ids_i = \{id_c \mid (\chi,\zeta) \vdash id_c <: id_i\}),\\ \psi = \cup_{i\ \in\ \{1,...,n\}}\ \{(id_i{:}(\iota,ids_i))\}\end{array}$$

Figure C-15: Rule for finding the type information for a set of types.

C.2.8 Type Elaborateds

A *type elaborated* records type information from the elaboration of declarations. It is a kind of type context used internally by the semantics. It differs from a type context in that it lacks the type information component, and instead contains a "has-type-environment" that records information from extension (`has`) declarations in BeCecil. The following is sometimes used to extract the type environment of a type elaborated.

tenv: *TypeElaborated* → *TypeEnv*
tenv(π, ι, ρ, *H*, χ, ζ) = π

The operator ∪ is used to collapse the type elaborateds produced by declarations at the same level into a single elaborated. For finite functions, and thus for the type environment, ∪ is a disjoint union operator. As for type contexts, the noncyclic union of the direct inheritance relations is used. The direct conformance relations are simply unioned together, as are the direct subtype relations. However, for the has-type-environment, if the same object identity has two pairs of invocable and assignable case sets, then the corresponding sets of cases must not clash, to ensure that certain obvious "message ambiguous" errors

cannot occur. Hence disjoint union is used for blending the has-type-environments, as described in the auxiliary function *blendGfTypes* below.

$$\cdot \cup \cdot : \mathit{TypeElaborated} \times \mathit{TypeElaborated} \to \mathit{TypeElaborated}_{\perp}$$
$$(\pi_1, \iota_1, H_1, \chi_1, \zeta_1) \cup (\pi_2, \iota_2, H_2, \chi_2, \zeta_2) =$$
$$(\pi_1 \cup \pi_2,\ \iota_1 \circledast \iota_2,\ \mathit{blendGfTypes}(H_1, H_2),\ \chi_1 \cup \chi_2,\ \zeta_1 \cup \zeta_2)$$
$$\mathit{blendGfTypes}: \mathit{HasTypes} \times \mathit{HasTypes} \to \mathit{HasTypes}_{\perp}$$
$$\mathit{blendGfTypes}(H_1, H_2) = \{(id{:}(ics,acs)) \mid (id{:}(ics_1,acs_1)) \in H_1,\ (id{:}(ics_2,acs_2)) \in H_2,$$
$$ics = ics_1 \cup ics_2,\ acs = acs_1 \cup acs_2\}$$
$$\cup \{(id{:}(ics_1,acs_1)) \mid (id{:}(ics_1,acs_1)) \in H_1,\ id \notin dom(H_2)\}$$
$$\cup \{(id{:}(ics_2,acs_2)) \mid (id{:}(ics_2,acs_2)) \in H_2,\ id \notin dom(H_1)\}$$

C.3 Type Checking Rules

In this section we explain the type checking rules given in Figures C-22 through C-38. The order of presentation follows the order of the syntax of Figure 3-1. The table in Figure C-16 gives an overview of the types of sequents used in the type system.

C.3.9 Implementation-Side Type Checks

For each class declared, its generic function value must conformingly, completely, and consistently implement the inexact types that it conforms to. In BeCecil, this means that for every inexact generic function type that a class conforms to, if there are classes that conform to the argument types, then there must be a unique, most-specific applicable case that can handle a possible call that type checks against that inexact type. The check for arrow types is a translation of the condition given in [Chambers & Leavens 95, Section 4.2]. The check for assignable types takes into account that the "result type" for an assignable type really describes the value used in an assignment, and hence describes an additional argument.

The rules for implementation-side type checking are given in Figures C-17 to C-19. The rule in Figure C-17 generates all the inexact types that a given class conforms to, and then uses the rule given in Figure C-18 to check that each such inexact type is implemented. The rule given in Figure C-18 generates all tuples of wildcards or classes that conform to the argument types of the inexact type in question, using the following auxiliary functions, and tests that for each possible tuple, there is a single best exact case and that the best exact case passes the conformance tests. Wildcards, written as **wild**, are needed because the type system places no restrictions on conformance to invocable, assignable, or exact types; hence regardless of whether there are classes in the assumed type context that conforms to such types, there is always the possibility that there will be such one. However, such conformance relationships may also be declared in the assumed type context, and thus invocables may be specialized on particular objects having some such type; thus the first auxiliary function below allows for both wildcards and declared classes that conform to such types. Technically, wildcards are used whenever a type has a subtype that is an invocable, assignable, or exact type. This is sensible because every such type allows arguments that are not restricted to known classes by the type system.

$$\mathit{conformers}: \mathit{TypeEnv} \times \mathit{Conforms} \times \mathit{Subtypes} \times \mathit{TypeAttribute} \to \mathit{Powerset}(\mathit{ObjectId} + \mathbf{wild})$$
$$\mathit{conformers}(\pi, \chi, \zeta, \tau) = \{id_c \mid id_c \in \mathit{range}(\mathit{classes}(\pi)),\ (\chi,\zeta) \vdash id_c <: \tau\}$$
$$\cup \{\mathbf{wild} \mid \zeta \vdash \mathbf{arrow}(\tau v, \tau_r) \leq_{sub} \tau\} \cup \{\mathbf{wild} \mid \zeta \vdash \mathbf{assignable}(\tau v, \tau_r) \leq_{sub} \tau\}$$
$$\cup \{\mathbf{wild} \mid \zeta \vdash \mathbf{exact}(ebs) \leq_{sub} \tau\}$$

Tuples of wildcards and conforming classes are generated using the following auxiliary function.

$$\mathit{conformers}: \mathit{TypeEnv} \times \mathit{Conforms} \times \mathit{Subtypes} \times \mathit{TypeAttribute}^* \to \mathit{Powerset}((\mathit{ObjectId} + \mathbf{wild})^*)$$
$$\mathit{conformers}(\pi, \chi, \zeta, (\tau_1,...,\tau_n)) = \{(id_1,...,id_n) \mid id_i \in \mathit{conformers}(\pi,\chi,\zeta,\tau_i),\ i \in \{1,...,n\}\}$$

	type			
prototype sequent	assumes (inherits)	works on (category)	produces (synthesizes)	defined in Figure
$(\pi,\iota,\chi,\zeta) \vdash$ *id impl-type-checks*	*TypeEnv* $\times$ *Inherits* $\times$ *Conforms* $\times$ *Subtypes*	*ObjectId*		C-17
$\vdash P : \tau$		*Program*	*TypeAttribute*	C-22
$K \vdash RDS \succ K'$	*TypeContext*	*Recursive-Declaration-Sequence*	*TypeContext*	C-23
$K \vdash (\pi,\iota,\chi,\zeta)$ *nests-ok*	*TypeContext*	*TypeEnv* $\times$ *Inherits* $\times$ *Conforms* $\times$ *Subtypes*		C-24
$K \vdash B : \tau$	*TypeContext*	*Block*	*TypeAttribute*	C-27
$K \vdash D^* \succ \xi$	*TypeContext*	*Declaration**	*TypeElaborated*	C-28
$K \vdash D \succ \xi$	*TypeContext*	*Declaration*	*TypeElaborated*	C-29
$\pi \vdash CN \Rightarrow_{class} \tau$	*TypeEnv*	*Class-Name*	*TypeAttribute*	C-30
$K \vdash GF \Rightarrow (ics,acs)$	*TypeContext*	*Generic-Function*	*InvokeCaseSet* $\times$ *AssignCaseSet*	C-31
$\pi \vdash T \Rightarrow \tau$	*TypeContext*	*Type-expression*	*TypeAttribute*	C-32, C-33
$\pi \vdash TN \Rightarrow_{type} \tau$	*TypeEnv*	*Type-Name*	*TypeAttribute*	C-34
$K \vdash E : \tau$	*TypeContext*	*Expression*	*TypeAttribute*	C-35, C-36
$\pi \vdash I \Rightarrow \theta$	*TypeEnv*	*Identifier*	*TypeDenotable*	C-37
$K \vdash AA \blacktriangleright (\tau, ids)$	*TypeContext*	*Acutal-Argument*	*TypeAttribute* $\times$ *Powerset(ObjectId)*	C-38

Figure C-16: Sequents used in the static semantics. This table includes sequents for implementation-side type checking and a sequent for every syntactic category; see Figure C-2 for auxiliary sequents. For each kind of sequent, the types it relates are shown in three parts on its row; what it assumes, what it works on, and what it produces.

To make wildcards act as their name implies, they must inherit from every class and conform to every suitable type. Thus the axiom in Figure C-20 is added to those already given (Figure A-3) for inheritance relationships, and we consider $\leq_{inh}$ to be a binary relation on the extended domain *ObjectId* + **wild** from now on. Similarly, the rules in Figure C-21 are added to those already given for conformance relationships. These rules make wildcards conform to every type for which they can be introduced by *conformers*. We hereinafter consider conformance relations to also relate **wild** to type attributes.

$$\text{[impl-type-checks]}\quad \frac{(\pi,\iota,\chi,\zeta) \vdash id\ impls\ it_1,\ ...,\ (\pi,\iota,\chi,\zeta) \vdash id\ impls\ it_n,\ (\pi,\iota,\chi,\zeta) \vdash id\ impls\ at_1,\ ...,\ (\pi,\iota,\chi,\zeta) \vdash id\ impls\ at_m}{(\pi,\iota,\chi,\zeta) \vdash id\ impl\text{-}type\text{-}checks}$$

where $n \geq 0$, $m \geq 0$,
$\{it_1,...,it_n\} = \{\mathbf{arrow}(\tau v_i,\tau_{i,r}) \mid (\chi,\zeta) \vdash id <: \mathbf{arrow}(\tau v_i,\tau_{i,r})\}$,
$\{at_1,...,at_m\} = \{\mathbf{assignable}(\tau v_j,\tau_{j,r}) \mid (\chi,\zeta) \vdash id <: \mathbf{assignable}(\tau v_j,\tau_{j,r})\}$

Figure C-17: Rule for when an implementation type checks.

$$\text{[impls-arrow]}\quad \frac{(\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{arrow}(\tau v,\tau_r)\ for\ idv_1,\ ...,\ (\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{arrow}(\tau v,\tau_r)\ for\ idv_m}{(\pi,\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{arrow}((\tau_1,...,\tau_n),\tau_r)}$$

where $m \geq 0$,
$\{idv_1,...,idv_m\} = conformers(\pi,\chi,\zeta,\tau v)\}$

$$\text{[impls-assignable]}\quad \frac{(\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{assignable}(\tau v,\tau_r)\ for\ idv_1,\ ...,\ (\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{assignable}(\tau v,\tau_r)\ for\ idv_m}{(\pi,\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{assignable}(\tau v,\tau_r)}$$

where $m \geq 0$,
$\{idv_1,...,idv_m\} = conformers(\pi,\chi,\zeta,\tau v)\}$

Figure C-18: Rules for when a class implements an inexact type.

$$\text{[impls-for-arrow 1]}\quad \frac{(\chi,\zeta) \vdash id <: \mathbf{exact}(ics, acs),\ (\iota,aav) \vdash (idv, \mathbf{arrow}((\tau'_1,...,\tau'_n),\tau'_r))\ is\text{-}best\text{-}for\ ics,\ (\chi,\zeta) \vdash id_1 <: \tau'_1,\ ...,\ (\chi,\zeta) \vdash id_n <: \tau'_n,\ \zeta \vdash \tau'_r \leq_{sub} \tau_r}{(\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{arrow}((\tau_1,...,\tau_n),\tau_r)\ for\ (id_1,...,id_n)}$$

where $n \geq 0$,
$aav = ((id_1,\{\}),...,(id_n,\{\}))$

$$\text{[impls-for-assignable]}\quad \frac{(\chi,\zeta) \vdash id <: \mathbf{exact}(ics, acs),\ (\iota,aav) \vdash (idv, \mathbf{assignable}((\tau'_1,...,\tau'_n),\tau'_r))\ is\text{-}best\text{-}for\ acs,\ (\chi,\zeta) \vdash id_1 <: \tau'_1,\ ...,\ (\chi,\zeta) \vdash id_n <: \tau'_n,\ \zeta \vdash \tau_r \leq_{sub} \tau'_r}{(\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{assignable}((\tau_1,...,\tau_n),\tau_r)\ for\ (id_1,...,id_n)}$$

where $n \geq 0$,
$aav = ((id_1,\{\}),...,(id_n,\{\}))$

Figure C-19: Rules for when a class implements an inexact type for a particular vector of argument classes.

$$\text{[tc program]}\quad \frac{(\{\}, \{\}, \{\}, \{\}, \{\}, \{\}) \vdash RDS \succ \mathrm{K},\ \mathrm{K} \vdash B : \tau}{\vdash RDS\ ;\ B\ :\ \tau}$$

Figure C-22: Static semantics for programs.

C.3.10 Type Checking Programs

The type checking of a program produces a type for the block that makes up the program's body. The formal rule is given in Figure C-22.

[inh-wild] $\iota \vdash \textbf{wild} \leq_{inh} id$

Figure C-20: Axiom that a wild card inherits from every object.

$$\text{[conforms-wild1]}\quad \frac{\zeta \vdash \textbf{arrow}(\tau_v,\tau_r) \leq_{sub} \tau}{(\chi,\zeta) \vdash \textbf{wild} <: \tau}$$

$$\text{[conforms-wild2]}\quad \frac{\zeta \vdash \textbf{assignable}(\tau_v,\tau_r) \leq_{sub} \tau}{(\chi,\zeta) \vdash \textbf{wild} <: \tau}$$

$$\text{[conforms-wild3]}\quad \frac{\zeta \vdash \textbf{exact}(ics, acs) \leq_{sub} \tau}{(\chi,\zeta) \vdash \textbf{wild} <: \tau}$$

Figure C-21: Rules for when a wild card conforms to a type.

$$\text{[tc rec-decl-sequence]}\quad \frac{\begin{array}{c}(\pi',\iota',\phi',\chi',\zeta',\psi') \vdash D^* \succ (\pi,\iota,H,\chi,\zeta),\\ (\chi',\zeta') \vdash id_1\ \textit{one-exact-type},\\ ...,\\ (\chi',\zeta') \vdash id_n\ \textit{one-exact-type},\\ (\pi',\iota',\chi',\zeta') \vdash id_1\ \textit{impl-type-checks},\\ ...,\\ (\pi',\iota',\chi',\zeta') \vdash id_n\ \textit{impl-type-checks},\\ (\chi',\zeta',\iota') \vdash ids_{types}\ \textit{type-info-is}\ \psi,\\ (\pi_0,\iota_0,\phi_0,\chi_0,\zeta_0,\psi_0) \vdash (\pi,\iota,\chi,\zeta)\ \textit{nests-ok}\end{array}}{(\pi_0,\iota_0,\phi_0,\chi_0,\zeta_0,\psi_0) \vdash D^* \succ (\pi',\iota',\phi',\chi',\zeta',\psi')}$$

where $n \geq 0$,
$\pi' = \pi_0 \uplus \pi$,
$\iota' =_s \iota_0 \otimes \iota$,
$\phi = \{(id{:}\iota') \mid (I{:}\textbf{class}(id)) \in \pi\}$,
$\phi' =_s \phi_0 \cup \phi$,
$\chi' = \chi_0 \cup \chi \cup \textit{exactConforms}(\pi,H)$,
$\zeta' = \zeta_0 \cup \zeta$,
$\psi' =_s \psi_0 \cup \psi$,
$\neg \textit{hasOrphans}(\pi,H)$,
$\{id_1,...,id_n\} = \textit{range}(\textit{classes}(\pi))$,
$ids_{types} = \textit{range}(\textit{types}(\pi))$

Figure C-23: Static Semantics of recursive declaration sequences.

The declarations in the recursive declaration sequence that forms the prelude are checked starting from an empty type context. The details of the checking process are described below. This process produces a type context κ that is used to check the block.

C.3.11 Type Checking Recursive Declaration Sequences

The type checking of a recursive declaration sequence in an assumed type context produces a new type context. The formal rule is given in Figure C-23.

As in the dynamic semantics, the rule for checking a recursive declaration sequence is a bit tricky. The checking of the declaration sequence, *D**, like the checking of an individual declaration, does not produce a type context, but a type elaborated (see Section C.2.8). The type elaborated, (π,ι,H,χ,ζ), that results from the checking of the declaration sequence is used to form a new type context, $(\pi',\iota',\phi',\chi',\zeta',\psi')$, in which names declared in *D** shadow those from the assumed context (hence $\pi' = \pi_0 \uplus \pi$) and in which the other information is essentially unioned together. Assuming this new type context when checking the declaration sequence itself allows the declarations to be mutually recursive.

The following auxiliary function uses the type environment and the has-type-environment of the elaborated to produce a direct conformance relation that contains exact type information for each class declared *D**.

$$\text{[tc block]}\quad \frac{(\pi_0,\iota_0,\phi_0,\chi_0,\zeta_0,\psi_0) \vdash \mathit{RDS} \succ \mathrm{K},\quad \mathrm{K} \vdash E : \tau,\quad \pi_0 \vdash \tau\ \mathit{is\text{-}well\text{-}formed}}{(\pi_0,\iota_0,\phi_0,\chi_0,\zeta_0,\psi_0) \vdash \mathit{RDS}\ E : \tau}$$

$$\text{[subsume-block]}\quad \frac{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash B : \tau',\quad \zeta \vdash \tau' \leq_{sub} \tau}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash B : \tau}$$

Figure C-27: Static semantics for blocks.

$$\mathit{exactConforms}: \mathit{TypeEnv} \times \mathit{HasTypes} \rightarrow \mathit{Conforms}$$
$$\mathit{exactConforms}(\pi, H) = \{(id{:}\textbf{exact}(ics,acs)) \mid (id{:}(ics,acs)) \in H\} \cup \{(id{:}\textbf{exact}(\{\},\{\})) \mid id \notin \mathit{dom}(H),\ id \in \mathit{range}(\mathit{classes}(\pi))\}$$

The class information for each class declared in *D** is formed by pairing each class with the direct inheritance relation ι′. The direct inheritance relation ι′ is formed as the noncyclic union of the inheritance relation of the surrounding contour and the direct inheritance relation of the elaborated. (Recall that $=_s$ is strict equality; hence if the combined inheritance relation is cyclic, the side condition is false.) The process of extracting the type information for the types declared is given in Figure C-15.

Several checks are performed on the elaborated at this time. Every declared class must have just one exact type, and each class's implementation must type check against the inexact types to which it conforms. These checks are described in Figures C-9 and C-17. As in the dynamic semantics, there is the possibility of "extension orphans:" object identities bound to extensions in the has-type-environment that are not for objects declared in the block. This error is checked with the aid of the following auxiliary function.

$$\mathit{hasOrphans}: \mathit{TypeEnv} \times \mathit{HasTypes} \rightarrow \mathit{Boolean}$$
$$\mathit{hasOrphans}(\pi, \rho, H) = (\exists id\ .\ id \in \mathit{dom}(H) \wedge id \notin \mathit{range}(\mathit{classes}(\pi)))$$

Recall from Section 3.5 that we also impose certain restrictions on nested contours to ensure that assumptions used in type checking their surrounding contours remain valid. These restrictions are stated in Figures C-24 through C-26. The rule for *nests-ok* given in Figure C-25 checks that for each class that conforms to a type declared in an outer contour, the class singly-inherits from a class known (in the surrounding contour) to conform to that type. The rule given in Figure C-25 checks that the class inherits from some class that is known in the surrounding contour, and that for each pair of classes that it inherits from, those classes are ordered.

C.3.12 Type Checking Blocks

Type checking a block in an assumed type context produces a type attribute for the block's value. The formal rules are given in Figure C-27. The second of these rules merely allows the type produced to be converted to a supertype. We now explain the details of the first rule.

In a block, of the form *RDS E*, the recursive declaration sequence *RDS* is checked as described above and produces a new type context, K. This context is used to check the expression, *E*.

In a block, the type of the expression cannot involve type names declared in the block itself. This condition is tested by the sequent $\pi_0 \vdash \tau$ *is-well-formed*. The rules for this sequent are given in Figures C-4 and C-5, which simply checks that all named types and classes in τ are known in the assumed type environment π_0.

[nests-ok]
$$\frac{(\iota',\psi_0) \vdash (id_1,tid_1)\ \textit{inherits-above}, \quad ..., \quad (\iota',\psi_0) \vdash (id_n,tid_n)\ \textit{inherits-above}}{(\pi_0,\iota_0,\phi_0,\chi_0,\zeta_0,\psi_0) \vdash (\pi,\iota,\chi,\zeta)\ \textit{nests-ok}}$$
where $n \geq 0$,
$\iota' =_s \iota_0 \circledast \iota$,
$\{(id_1,tid_1),...,(id_n,tid_n)\} =$
$\{(id,tid) \mid (I{:}\textbf{namedtype}(tid)) \in \pi_0,$
$id \in \textit{conformers}(\pi_0 \uplus \pi, \chi_0 \cup \chi, \zeta_0 \cup \zeta, \textbf{namedtype}(tid))\}$

Figure C-24: Rule for when a new context nests correctly with respect to an assumed context.

[inherits-above]
$$\frac{\iota \vdash id \leq_{inh} id_c, \quad \iota' \vdash (id_{1,1},id_{1,2})\ \textit{ordered}, \quad ..., \quad \iota' \vdash (id_{n,1},id_{n,2})\ \textit{ordered}}{(\iota,\psi) \vdash (id,tid)\ \textit{inherits-above}}$$
where $n \geq 0$,
$(tid{:}(\iota',ids)) \in \psi$,
$id_c \in ids$,
$\{(id_{1,1},id_{1,2}),...,(id_{n,1},id_{n,2})\} =$
$\{(id_a,id_b) \mid id_a \in ids,\ id_b \in ids,$
$\iota \vdash id \leq_{inh} id_a,\ \iota \vdash id \leq_{inh} id_b\}$

Figure C-25: Rule for when a class singly inherits from a class that is known to conform to a type.

[ordered1]
$$\frac{\iota \vdash id_1 \leq_{inh} id_2}{\iota \vdash (id_1,id_2)\ \textit{ordered}}$$

[ordered2]
$$\frac{\iota \vdash id_2 \leq_{inh} id_1}{\iota \vdash (id_1,id_2)\ \textit{ordered}}$$

Figure C-26: Rules for when two objects are ordered by an inheritance relation.

[tc declaration sequence]
$$\frac{K_0 \vdash D_1 \succ \xi_1, ..., K_0 \vdash D_n \succ \xi_n}{K_0 \vdash (D_1 \ldots D_n) \succ \xi}$$
where $n \geq 0$,
$\xi =_s \cup_{i \in \{1,...,n\}} \xi_i$

Figure C-28: Static semantics of declaration sequences.

C.3.13 Type Checking Declaration Sequences

The checking of a declaration sequence in an assumed type context produces a type elaborated. The type checking rule is given in Figure C-28.

When a declaration sequence is complete, the type elaborateds that result from the checking of the individual declarations are collapsed into a single type elaborated, which combines all of their information. The error checking that is done at this time is analogous to that done in the dynamic semantics; it is accomplished by the disjoint union operator on type elaborateds, $\cup$.

C.3.14 Type Checking Declarations

The checking of a declaration in an assumed type context produces a type elaborated. The formal semantics is given in Figure C-29.

[tc object declaration]
$$K \vdash \textbf{object}\ I \succ (\{I\text{:}\textbf{class}(id)\}, \{\}, \{\}, \{\}, \{\})$$
where $\pi = tenv(\kappa)$, $id \notin range(classes(\pi))$

[tc inherits declaration]
$$\frac{\pi \vdash CN_1 \Rightarrow_{class} id_1, \quad \pi \vdash CN_2 \Rightarrow_{class} id_2}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash CN_1\ \textbf{inherits}\ CN_2 \succ (\{\},\{(id_1, id_2)\},\{\},\{\},\{\})}$$

[tc extension declaration]
$$\frac{\pi \vdash CN \Rightarrow_{class} id, \quad (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash GF \Rightarrow (ics,acs)}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash CN\ \textbf{has}\ GF \succ (\{\}, \{\}, \{id\text{:}(ics,acs)\}, \{\}, \{\})}$$

[tc hide declaration]
$$\frac{(\pi_0,\iota_0,\phi_0,\chi_0,\zeta_0,\psi_0) \vdash RDS \succ (\pi',\iota',\phi',\chi',\zeta',\psi'), \quad (\pi',\iota',\phi',\chi',\zeta',\psi') \vdash D* \succ \xi}{(\pi_0,\iota_0,\phi_0,\chi_0,\zeta_0,\psi_0) \vdash \textbf{hide}\ RDS\ \textbf{in}\ D*\ \textbf{end} \succ \xi}$$
where $\pi = tenv(\xi)$, $(\forall I\ .\ I \in dom(\pi) \Rightarrow \pi_0(I) = \pi'(I))$

[tc type declaration]
$$K \vdash \textbf{type}\ I \succ (\{I\text{:}\textbf{namedtype}(tid)\}, \{\}, \{\}, \{\}, \{\})$$
where $\pi = tenv(\kappa)$, $tid \notin range(types(\pi))$

[tc conforms declaration]
$$\frac{\pi \vdash CN \Rightarrow_{class} id, \quad \pi \vdash T \Rightarrow \tau}{K \vdash CN\ \textbf{conforms}\ T \succ (\{\}, \{\}, \{\}, \{(id, \tau)\}, \{\})}$$
where $\pi = tenv(\kappa)$

[tc subtypes declaration]
$$\frac{\pi \vdash TN \Rightarrow_{type} tid, \quad \pi \vdash T \Rightarrow \tau}{K \vdash TN\ \textbf{subtypes}\ T \succ (\{\}, \{\}, \{\}, \{\}, \{(tid, \tau)\})}$$
where $\pi = tenv(\kappa)$

Figure C-29: Static semantics of declarations.

An object declaration, of the form **`object`** *I*, is elaborated by allocating a fresh object identity, and binding the identifier *I* to it. A type declaration, of the form **`type`** *I*, is handled similarly.

An inheritance declaration is elaborated by producing a direct-inheritance relation that relates the first class named to the second. The rules for elaborating conformance and subtyping declarations are similar. The classes involved in these rules cannot be formal parameters, since the identity of a formal parameter cannot, in general, be known statically, and thus is not tracked by the type system.

An extension declaration, of the form *CN* **`has`** *GF*, is elaborated by evaluating the generic function attribute *GF*, obtaining a pair of exact case sets, and then binding the identity of *CN* to that pair.

In a hide declaration, of the form **`hide`** *RDS* **`in`** *D** **`end`**, recall that both sets of declarations are mutually recursive with each other. As in the dynamic semantics, extension declarations can only appear in the same part of a hide declaration as the object declarations they refer to. The side condition checks that the same name is not declared in both parts of the hide declaration.

C.3.15 Static Semantics of Class Names

A class name is looked up in the assumed type context, and its identity is returned. The formal axiom is given in Figure C-30. Note that the identifier in question must denote a class, not a type or a formal.

[tc class name] $\pi \vdash I \Rightarrow_{class} id$ where $(I{:}\mathbf{class}(id)) \in \pi$

Figure C-30: Static semantics of class names.

[tc identifier attribute 1]
$$\frac{\pi \vdash I \Rightarrow \mathbf{class}(id),\quad (\chi,\zeta) \vdash id <: \mathbf{exact}(ics,acs)}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I \Rightarrow (ics,acs)}$$

[tc identifier attribute 2]
$$\frac{\pi \vdash I \Rightarrow \mathbf{formal}(\tau),\quad \zeta \vdash \tau \leq_{sub} \mathbf{exact}(ics,acs)}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I \Rightarrow (ics,acs)}$$

[tc method attribute]
$$\frac{\pi \vdash CN_1 \Rightarrow_{class} id_1,\ ...,\ \pi \vdash CN_n \Rightarrow_{class} id_n,\quad \pi \vdash T_1 \Rightarrow \tau_1,\ ...,\ \pi \vdash T_n \Rightarrow \tau_n,\quad \pi \vdash T_r \Rightarrow \tau_r,\quad (\pi'',\iota,\phi,\chi,\zeta,\psi) \vdash B : \tau_r}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash \texttt{method(}I_1@CN_1\texttt{:}T_1,...,I_n@CN_n\texttt{:}T_n\texttt{):}T_r\ \texttt{\{}B\texttt{\}} \Rightarrow (\{((id_1,...,id_n),\mathbf{arrow}((\tau_1,...,\tau_n),\tau_r))\},\{\})}$$
where $n \geq 0$, $\pi' =_s \{I_1{:}\mathbf{formal}(\tau_1)\} \cup ... \cup \{I_n{:}\mathbf{formal}(\tau_n)\}$, $\pi'' = \pi \uplus \pi'$

[tc storage attribute]
$$\frac{\pi \vdash CN_1 \Rightarrow_{class} id_1,\ ...,\ \pi \vdash CN_n \Rightarrow_{class} id_n,\quad \pi \vdash T_1 \Rightarrow \tau_1,\ ...,\ \pi \vdash T_n \Rightarrow \tau_n,\quad \pi \vdash T_r \Rightarrow \tau_r,\quad (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash E : \tau_r}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash \texttt{storage(}I_1@CN_1\texttt{:}T_1,...,I_n@CN_n\texttt{:}T_n\texttt{) :=}E\texttt{:}T_r \Rightarrow (\{((id_1,...,id_n),\mathbf{arrow}((\tau_1,...,\tau_n),\tau_r))\},\{((id_1,...,id_n),\mathbf{assignable}((\tau_1,...,\tau_n),\tau_r))\})}$$
where $n \geq 0$, the $I_1,...,I_n$ are distinct

[tc acceptor attribute]
$$\frac{\pi \vdash CN_1 \Rightarrow_{class} id_1,\ ...,\ \pi \vdash CN_n \Rightarrow_{class} id_n,\quad \pi \vdash T_1 \Rightarrow \tau_1,\ ...,\ \pi \vdash T_n \Rightarrow \tau_n,\quad \pi \vdash T_r \Rightarrow \tau_r,\quad (\pi'',\iota,\phi,\chi,\zeta,\psi) \vdash B : \tau_r}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash \texttt{acceptor(}I_1@CN_1\texttt{:}T_1,...,I_n@CN_n\texttt{:}T_n\texttt{):=}\ I\texttt{:}T_r\ \texttt{\{}B\texttt{\}} \Rightarrow (\{\},\{((id_1,...,id_n),\mathbf{assignable}((\tau_1,...,\tau_n),\tau_r))\})}$$
where $n \geq 0$, $\pi' =_s \{I_1{:}\mathbf{formal}(\tau_1)\} \cup ... \cup \{I_n{:}\mathbf{formal}(\tau_n)\} \cup \{I{:}\mathbf{formal}(\tau_r)\}$, $\pi'' = \pi \uplus \pi'$

[tc combination attribute]
$$\frac{K \vdash GF_1 \Rightarrow (ics_1,acs_1),\quad K \vdash GF_2 \Rightarrow (ics_2,acs_2)}{K \vdash GF_1\ \&\ GF_2 \Rightarrow ebs}$$
where $ics = ics_1 \uplus ics_2$, $acs = acs_1 \uplus acs_2$

Figure C-31: Static semantics of generic function attributes.

C.3.16 Type Checking Generic Function Attributes

The evaluation of a generic function attribute in an assumed type context produces a set of exact cases. The formal semantics is given in Figure C-31.

[tc type name] $\pi \vdash I \Rightarrow_{type} tid$ where $(I{:}\textbf{namedtype}(tid)) \in \pi$

Figure C-34: Static semantics of type names.

An identifier attribute, of the form *I*, produces a pair consisting of an invocable case set and an assignable case set. These are taken from the exact type of *I*. Notice that a formal parameter that has an exact type can be used as an identifier attribute. This is because with an exact type, the exact type information needed for this rule is known.

A method attribute produces a pair whose invocable case set contains a single exact case. Its specializers and argument types are determined by the formals. Storage table attributes are treated similarly, except that for methods, the block must be checked, while for storage tables, it is the initialization expression that must be checked. The checking of an acceptor is similar to that of a method, but the block has access to the value parameter as an additional formal. The type of the block or expression may be a subtype of the stated type, as allowed by the subsumption rules (given in Figures C-27 and C-36).

A combination attribute, of the form GF_1 **&** GF_2, produces a pair of exact case sets that contain the exact cases of GF_2 and those from GF_1 that do not clash with those in GF_2. Each set of exact cases produced includes all the exact cases from GF_2. For both the invocables and assignables, if an exact case of GF_1 clashes with an exact case in GF_2, then only the exact case from GF_2 remains, but the exact cases of GF_1 that do not clash with those of GF_2 are also included. (Thus, if there are no clashes, then the exact cases of both generic functions are returned.)

C.3.17 Static Semantics of Type Expressions

The static semantics of a type expression in an assumed type context produces a type attribute. The formal rules are given in Figures C-32 and C-33.

C.3.18 Static Semantics of Type Names

A type name is looked up in the assumed type environment, and its identity is returned. The formal rule is given in Figure C-34.

C.3.19 Type Checking Expressions

The checking of an expression in an assumed type context produces a type attribute. The type checking rules are given in Figures C-35 and C-36.

An identifier used as an expression may either be the name of a class or a formal. If it is a class name that denotes a class with identity, *id*, then its type is the type to which *id* conforms. If it is a formal parameter, its type is the type to which the formal is bound in the surrounding type context. (The first of these rules could be made more deterministic by producing the conjunction of all the types to which the name conforms, but the type system can already generate such a type by using the structural rules for conjunctive types.)

For type checking an application expression, of the form, $E_0(AA*)$, there are two cases. If none of the actuals are directed, then the operator, E_0, must have an arrow type and each actual argument must have a type that is the same (or by subsumption, a subtype of) the corresponding formal argument type. Otherwise, the operator must be a class name, and a more complex process is used to determine whether there is the possibility of an error in the invocation, and if not, what the result type will be. The idea is that for each tuple of conforming classes that could be arguments (i.e., that conform to the types of the actuals), there

[named type]
$$\frac{\pi \vdash TN \Rightarrow_{type} tid}{\pi \vdash TN \Rightarrow \textbf{namedtype}(tid)}$$

[arrow type]
$$\frac{\pi \vdash T_1 \Rightarrow \tau_1,\ ...,\ \pi \vdash T_n \Rightarrow \tau_n,\quad \pi \vdash T_n \Rightarrow \tau_r}{\pi \vdash \textbf{(}T_1,...,T_n\textbf{)->}T_r \Rightarrow \textbf{arrow}((\tau_1,...,\tau_n),\tau_r)}$$ where $n \geq 0$

[assignable type]
$$\frac{\pi \vdash T_1 \Rightarrow \tau_1,\ ...,\ \pi \vdash T_n \Rightarrow \tau_n,\quad \pi \vdash T_n \Rightarrow \tau_r}{\pi \vdash \textbf{(}T_1,...,T_n\textbf{):=}T_r \Rightarrow \textbf{assignable}((\tau_1,...,\tau_n),\tau_r)}$$ where $n \geq 0$

[and type]
$$\frac{\pi \vdash T_1 \Rightarrow \tau_1,\quad \pi \vdash T_2 \Rightarrow \tau_2}{\pi \vdash T_1 \textbf{ \& } T_2 \Rightarrow \textbf{andtype}(\{\tau_1, \tau_2\})}$$

[or type]
$$\frac{\pi \vdash T_1 \Rightarrow \tau_1,\quad \pi \vdash T_2 \Rightarrow \tau_2}{\pi \vdash T_1 \textbf{ | } T_2 \Rightarrow \textbf{ortype}(\{\tau_1, \tau_2\})}$$

[exact type]
$$\frac{\begin{array}{c}\pi \vdash ET_{f(1)} \Rightarrow (idv_1,\textbf{arrow}(\tau v_1,\tau_1)),\\ ...,\\ \pi \vdash ET_{f(n)} \Rightarrow (idv_n,\textbf{arrow}(\tau v_n,\tau_n)),\\ \pi \vdash ET_{f(n+1)} \Rightarrow (idv_{n+1},\textbf{assignable}(\tau v_{n+1},\tau_{n+1})),\\ ...,\\ \pi \vdash ET_{f(n+k)} \Rightarrow (idv_{n+k},\textbf{assignable}(\tau v_{n+k},\tau_{n+k}))\end{array}}{\begin{array}{c}\pi \vdash \textbf{exact\{}ET_1\textbf{, ..., }ET_{n+k}\textbf{\}} \Rightarrow\\ \textbf{exact}(\{(idv_1,\textbf{arrow}(\tau v_1,\tau_1)),...,(idv_n,\textbf{arrow}(\tau v_n,\tau_n))\},\\ \{(idv_{n+1},\textbf{assignable}(\tau v_{n+1},\tau_{n+1})),\\ ...,(idv_{n+k},\textbf{assignable}(\tau v_{n+k},\tau_{n+k}))\})\end{array}}$$ where $n \geq 0$, $k \geq 0$, f is a bijection on $\{1,...,n+k\}$, all of the ET_i are distinct.

Figure C-32: Static semantics of type expressions.

[exact arrow]
$$\frac{\begin{array}{c}\pi \vdash CN_1 \Rightarrow_{class} id_1,\ ...,\ \pi \vdash CN_n \Rightarrow_{class} id_n,\\ \pi \vdash T_1 \Rightarrow \tau_1,\ ...,\ \pi \vdash T_n \Rightarrow \tau_n,\quad \pi \vdash T_r \Rightarrow \tau_r\end{array}}{\pi \vdash \textbf{(}CN_1\textbf{:}T_1\textbf{, ..., }CN_n\textbf{:}T_n\textbf{)->}T_r \Rightarrow ((id_1,...,id_n),\textbf{arrow}((\tau_1,...,\tau_n),\tau_r))}$$ where $n \geq 0$

[storage type]
$$\frac{\begin{array}{c}\pi \vdash CN_1 \Rightarrow_{class} id_1,\ ...,\ \pi \vdash CN_n \Rightarrow_{class} id_n,\\ \pi \vdash T_1 \Rightarrow \tau_1,\ ...,\ \pi \vdash T_n \Rightarrow \tau_n,\quad \pi \vdash T_r \Rightarrow \tau_r\end{array}}{\pi \vdash \textbf{(}CN_1\textbf{:}T_1\textbf{, ..., }CN_n\textbf{:}T_n\textbf{):=}T_r \Rightarrow ((id_1,...,id_n),\textbf{assignable}((\tau_1,...,\tau_n),\tau_r))}$$ where $n \geq 0$

Figure C-33: Static semantics of exact types.

must be a most specific applicable exact case. From the set of all these selected cases, the least upper bound of their result types is returned as the result type of the invocation.

The auxiliary function zip, used in the rules for applications and assignments with directed actuals, is defined below.

zip: $ObjectId^* \times Powerset(ObjectId)^* \rightarrow ActualArgPair^*$
$zip((id_1,...,id_n),(ids_1,...,ids_n)) = ((id_1,ids_1),...,(id_n,ids_n))$

[tc identifier expression 1]

$$\frac{\pi \vdash I \Rightarrow \textbf{class}(id)}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I : \tau}$$

where $(id,\tau) \in \chi$

[tc identifier expression 2]

$$\frac{\pi \vdash I \Rightarrow \textbf{formal}(\tau)}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I : \tau}$$

[tc application expression 1]

$$\frac{\mathrm{K} \vdash E_0 : \textbf{arrow}((\tau_1,...,\tau_n),\tau_r),\ \mathrm{K} \vdash AA_1 \blacktriangleright (\tau_1, \{\}) , ..., \mathrm{K} \vdash AA_n \blacktriangleright (\tau_n, \{\})}{\mathrm{K} \vdash E_0(AA_1,...,AA_n) : \tau_r}$$

where $n \geq 0$

[tc application expression 2]

$$\frac{\begin{array}{c}(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I \Rightarrow_{class} id_0 \\ (\chi,\zeta) \vdash id_0 <: \textbf{exact}(ics,acs), \\ (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash AA_1 \blacktriangleright (\tau_1, ids_1) , \\ ..., \\ (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash AA_n \blacktriangleright (\tau_n, ids_n), \\ (\iota_0, aav_1) \vdash ec_1 \textit{ is-best-for } ics, \\ ..., \\ (\iota_0, aav_m) \vdash ec_m \textit{ is-best-for } ics, \\ \zeta \vdash (\tau_1,...,\tau_n) \leq_{sub} \tau\mathrm{v}_1, ..., \zeta \vdash (\tau_1,...,\tau_n) \leq_{sub} \tau\mathrm{v}_m\end{array}}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I\ (AA_1,...,AA_n) : \tau_r}$$

where $n \geq 0$, $m \geq 0$,
$\iota_0 = \phi(id_0)$
$\{idv_1,...,idv_m\} = conformers(\pi,\chi,\zeta,(\tau_1,...,\tau_n))$,
$(\forall i \,.\, i \in \{1,...,m\} \Rightarrow$
$ec_i \in ics$ and
$\tau\mathrm{v}_i = argtypes(ec_i)$ and
$aav_i = zip(idv_i, (ids_1,...,ids_n)))$,
$\tau_r = \textbf{ortype}(\{restype(ec_1), ..., restype(ec_m)\})$

[tc assignment expression 1]

$$\frac{\kappa \vdash E_0 : \tau,\ \zeta \vdash \tau \leq_{sub} \textbf{assignable}((\tau_1,...,\tau_n),\tau_{n+1}),\ \kappa \vdash AA_1 \blacktriangleright (\tau_1, \{\}) , ..., \kappa \vdash AA_n \blacktriangleright (\tau_n, \{\}),\ \kappa \vdash E_{n+1} : \tau_{n+1}}{\kappa \vdash E_0(AA_1,...,AA_n) \textbf{:=} E_{n+1} : \tau}$$

where $n \geq 0$,
$\zeta = directSubtypes(\kappa)$

[tc assignment expression 2]

$$\frac{\begin{array}{c}(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I \Rightarrow_{class} id_0 \\ (\chi,\zeta) \vdash id_0 <: \textbf{exact}(ics,acs), \\ (\chi,\zeta) \vdash id_0 <: \tau, \\ (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash AA_1 \blacktriangleright (\tau_1, ids_1) , \\ ..., \\ (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash AA_n \blacktriangleright (\tau_n, ids_n), \\ (\pi,\iota,\phi,\xi,\zeta,\psi) \vdash E_{n+1} : \tau_{n+1} \\ (\iota_0, aav_1) \vdash ec_1 \textit{ is-best-for } acs, \\ ..., \\ (\iota_0, aav_m) \vdash ec_m \textit{ is-best-for } acs, \\ \zeta \vdash (\tau_1,...,\tau_n) \leq_{sub} \tau\mathrm{v}_1, ..., \zeta \vdash (\tau_1,...,\tau_n) \leq_{sub} \tau\mathrm{v}_m, \\ \zeta \vdash \tau_{n+1} \leq_{sub} \tau_{r,1}, ..., \zeta \vdash \tau_{n+1} \leq_{sub} \tau_{r,m}\end{array}}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I\ (AA_1,...,AA_n) \textbf{:=} E_{n+1} : \tau}$$

where $n \geq 0$, $m \geq 0$,
$\iota_0 = \phi(id_0)$
$\{idv_1,...,idv_m\} = conformers(\pi,\chi,\zeta,(\tau_1,...,\tau_n))$,
$(\forall i \,.\, i \in \{1,...,m\} \Rightarrow$
$ec_i \in acs$ and
$\tau\mathrm{v}_i = argtypes(ec_i)$ and
$\tau_{r,i} = restype(ec_i)$ and
$aav_i = zip(idv_i, (ids_1,...,ids_n)))$

[tc sequence expression]

$$\frac{\mathrm{K} \vdash E_1 : \tau_1, \quad \mathrm{K} \vdash E_2 : \tau_2}{\mathrm{K} \vdash E_1 \textbf{;} E_2 : \tau_2}$$

Figure C-35: Static semantics of expressions.

[tc subsume-expression] $$\frac{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash E : \tau', \quad \zeta \vdash \tau' \leq_{sub} \tau}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash E : \tau}$$

Figure C-36: Subsumption rule for expressions.

[tc identifier 1] $\pi \vdash I \Rightarrow \mathbf{class}(id)$ where $(I{:}\mathbf{class}(id)) \in \pi$

[tc identifier 2] $\pi \vdash I \Rightarrow \mathbf{formal}(\tau)$ where $(I{:}\mathbf{formal}(\tau)) \in \pi$

Figure C-37: Static semantics of identifiers

[tc undirected actual] $$\frac{K \vdash E : \tau}{K \vdash E \blacktriangleright (\tau, \{\})}$$

[tc directed actual] $$\frac{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash E : \tau\ , \quad \pi \vdash CN_1 \Rightarrow_{class} id_1, \ldots, \pi \vdash CN_n \Rightarrow_{class} id_n}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash E@CN_1,\ldots,CN_n \blacktriangleright (\tau, \{id_1,\ldots,id_n\})}$$ where $n \geq 0$

Figure C-38: Static semantics of actual arguments.

For type checking an assignment, the process is similar to type checking an invocation. If none of the actuals are directed, then the first form may be used, in which the operator must have an assignable type. The actuals and the value being assigned must be the same as (or subtypes of) the argument and result types of this assignable type. If some of the actuals are directed, then the operator must again be a class name, and each selected case must be a storage table with appropriate types for arguments and the result.

A sequence expression has the type of the second expression in the sequence.

There is also a subsumption rule for expressions, given in Figure C-36.

C.3.20 Static Semantics of Identifiers

An identifier is looked up in the assumed type environment, and a type denotable is returned. The identifier must denote either a class or a formal. The axioms are given in Figure C-37.

C.3.21 Static Semantics and Type Checking of Actual Arguments

The static semantics of an actual argument in an assumed type context produces a typed actual. A typed actual is a pair of a type attribute and a set of class identities. The type checking rules are given in Figure C-38. For both kinds of actuals, the expression must type check, and its type attribute, τ, is returned as part of the pair. For a directed actual, each named class must be a class, not just a formal parameter with an exact type, so that its identity is known.

Appendix D Design Decisions and Alternatives for the BeCecil Type System

In this section we describe some alternatives in the design of the BeCecil type system, and the reasons for our decisions.

D.1 Separation of types and classes

A fundamental design decision in the type system is the separation of the notions of type and class, and thus of subtype and subclass. Although it would simplify the type system if it relied on the identification of types and classes, there are well-known limitations in languages that identify the notions [Snyder 86]. However, it might be reasonable to only "half" separate the two notions, by making types be classes, but introducing two notions of inheritance (public and private). For example, in C++, every subtype must be a subclass, but not all subclasses are subtypes. This compromise also has well-known limitations. However, from the point of view of a theoretical study, there are two different kinds of relationships in either case (subtyping and subclassing, or public and private inheritance), and it seems to be cleaner to use the distinction between subtyping and subclassing. We use the notion of conformance to bridge the gap between the two worlds of objects and types.

D.2 Structural vs. by Name Subtyping and Conformance

Another fundamental distinction is that between structural and "by name" type checking. This decision subsumes the finer distinction between structural and by name subtyping. In a language that distinguishes types and classes, one can also distinguish between structural and by name conformance; that is, does an object conform to a type simply by virtue of supporting the appropriate methods, or is an explicit declaration of conformance needed? The advantage of structural notions are that they work well with separate typechecking and with independent development of code; the disadvantage is that there sometimes there may be superficial matches (subtype relationships, conformance relationships) where the semantics of the types involved would say that the types should not be treated as matching. Languages like Modula-3 [Nelson 91, Harbison 92] allow the programmer to have it both ways, using structural relationships by default and allowing the programmer to explicitly "brand" types to prevent superficial matches.

In BeCecil, a fundamental problem with structural notions of typing, subtyping, and conformance is that named types and objects have no structure. Inexact types, that is invocable types and assignable types, do have structure, and there are structural rules that apply to them. There are also structural subtyping rules for conjunctive and disjunctive types. However, a named type has no such structure. Similarly, except in its capacity as a generic function, an object itself has no structure on which to base structural conformance rules. In the absence of such explicit structure, structural rules would have to be based on declared interfaces (protocols) for types. Such declarations are available in BeCecil, and thus structural subtyping could be based on the declared types of generic functions that are applicable to the types in question. However, such a notion is less fixed than similar notions for ADT or single-dispatch OO languages, because there is no clear distinction between "primitive" and "client" operations that apply to a type. Thus adding a new generic function that applied to a given type could change many subtype relationships in a program, possibly in subtle ways. Similarly, one could decide whether an object conforms to a type on the basis of what generic functions apply to it, but again this could change if new generic functions were defined. Furthermore, type names, since they correspond to ADTs, seem more likely to want "brands" to prevent superficial matches, but we wanted to avoid the extra complexity of dealing with both structural and by name relationships for type names in our theoretical language. For these reasons, we have only investigated by name subtyping and conformance for type names. We leave the investigation of structural relationships, and ways to explicitly declare type interfaces as future work.

D.3 Dealing with Nested Scopes

The trickiest aspect of the type system is in dealing with nested scopes. Recall from the discussion in Section 3 that the problem of types acquiring new conformers can lead to incompleteness, and the problem of relating unrelated conformers can lead to inconsistency. These problems have two different origins in our type system. The problem of types acquiring new conformers is caused by our separation of types and classes (i.e., by our separation of the notions of subtyping and inheritance). If every subtype had to be implemented by a class that inherited from a class that implemented the supertype in question, then this problem would disappear. Indeed this would also be a problem in single dispatch languages that had block structure and separated subtyping and inheritance.* The problem of relating unrelated conformers is due to multiple dispatch and our refusal to linearize the type hierarchy. With multiple dispatch and without linearization, the possibility of inconsistency in handling applications arises.

It may be instructive to briefly consider some ideas that we rejected for making the type system sound in the face of these problems. An early idea was to recheck that each generic function of a surrounding contour was properly implemented, in the context of each nested contour; the idea was that if the new object and inheritance declarations in the nested recursive declaration sequence did not invalidate the assumptions used in type checking the outer contour, then when objects created in the inner contour were passed out to surrounding contours, they would not cause problems. This is similar to what is the current type system does, but it is less conservative, and also less constructive. It is less constructive in the sense that it does not tell the programmer what situations to avoid in as much detail as our present rule.

Another idea was to prevent the “escape” from nested contours of all objects that did not inherit from classes known in outer contours. This would be sound, but was too restrictive, as it prevented almost all uses of inheritance in nested contours. The current rule is less restrictive, because it only prevents the uses of inheritance in nested contours that may cause the two problems mentioned above.

It should be noted, however, that nested contours are still not very useful. One basic feature of the dynamic semantics is that from within a nested contour, one cannot extend the generic functions or inheritance relationships of the surrounding contour in a way that will directly affect the surrounding contour. Thus, most programs will make most of their declarations at top-level.

D.4 Dealing with Directed Actual Arguments

In type checking applications with directed actual arguments, we chose to require that the operator be statically known (a class name). The reason for this is that to do the type checking for directed actuals, the inheritance relation closed with the class is needed to mimic the dynamic semantics.

We briefly considered putting information about the inheritance relation of a generic function into some form of exact type; the problem is that the classes that might be related by such inheritance relationships would not necessarily be statically known, which could happen if the generic function was defined in a contour other than a surrounding contour. Thus it is hard to envision how to write down such inheritance relations involving unknown class names; even class names that look as if they referred to classes in the surrounding contour might not be intended to do so. In any case the syntax would be messy and there seemed no great need for the capability of using directed actuals with unknown generic functions. Indeed it is somewhat against the spirit of OO programming to be able to invoke a particular method in an unknown generic function; it would be like extracting and calling a particular method of an unknown object in a single-dispatch language.

* Beta [Madsen et al. 93] is a block-structured OO language, but does not separate types and class.

D.5 Inference vs. Declaration for Generic Function Types

The typing rules for BeCecil could either infer the generic function types of objects or require them to be declared. The present type system infers the exact types of objects (i.e., the types of all generic function attributes). Thus programmers are not required to declare generic function types for objects, as they can always be inferred. However, programmers can still declare that generic functions must conform to a given type, and such declarations are checked against the declared exact types during implementation-side type checking. This has an advantage, in that one can specify that all objects that conform to a given type must implement a particular operation.

There is a small disadvantage to inferring inexact types from exact types, however. It is that programmers have to be careful to not make the result types of methods too specific, relative to their declared argument types, as described in Section 3.2.2. Language changes that would prevent such problems would be to require inexact type declarations instead of inferring them, or to allow the programmer to have some way to turn off the inference for particular methods. This second solution would complicate the language in a way that does not seem orthogonal, and so was rejected. The first solution seems workable, but would be farther away from a practical language, because most generic function type declarations would be redundant with generic function attributes. We leave investigation of this solution as an alternative for future work.

D.6 Object Roles (abstract vs. concrete objects)

In early versions of BeCecil (and in our previous work [Chambers & Leavens 94, Chambers & Leavens 95]), we distinguished two *roles* for objects. An *abstract* object could not be used in an expression, while a concrete object could be used in an expression. This distinction mimicked the distinction between abstract and concrete classes in languages such as Eiffel [Meyer 88, Meyer 92], C++, Dylan, and many others.

In a language where classes are types and subclasses generate instances that are subtypes, such a distinction is necessary. The reason is so that the programmer does not have to implement all methods that are specialized on abstract classes, since such classes cannot be directly used to create instances. A method that is not implemented in an abstract class, and which must be overridden in all subclasses is called a “pure virtual method” in C++, and a “deferred method” in Eiffel. Examples of such methods include the `if` method in the classic coding of the Booleans with an abstract class `Boolean` and two subclasses, `True` and `False` [Goldberg & Robson 83]. Forcing the programmer to implement such methods in the abstract class would cause programmers to write methods that looped or gave errors when called; but such code is completely worthless. At best such code will never be called, and at worst a run-time error will occur, essentially reducing what should be a static error to a dynamic error. It is better to follow languages like C++, where one can declare that an abstract class has some pure virtual methods, and thus inform the type system that the method does not have to be implemented in that class.

However, a distinction between abstract and concrete objects turned out to be an unnecessary complication in BeCecil, because in BeCecil objects and types are completely separated. To declare that an object is an abstract class, one simply does not declare that it conforms to any type. Such an object cannot be used as an object in an expression (although it can be used as a generic function), and therefore it is abstract. This also simplifies the type system, which no longer has to deal with abstract objects that conform to various types.

D.7 Should Specializers Conform to Declared Argument Types?

It might seem natural to require that specializers of formal arguments should conform to their declared argument types. For example, in the following method, the first argument conforms in this way, but the second does not, because `GenericFun_rep` does not conform to `()->Top`.

```
ifTrue has method(b@boolean_rep:boolean,
                  c@GenericFun_rep:()->Top): Top {...}
```

Methods that use the specializer `any` would also not conform to their declared argument type. Such examples are common, and the use of `any` in particular is a pattern that makes an argument effectively "unspecialized" (if every class is assumed to inherit from `any`, as in Cecil).

The above discussion shows, however, only that it is a considerable programming convenience to permit specializers to conform not conform to their declared argument types. One could, less conveniently, declare a less restrictive type, or a more restrictive specializer class. In a language in which classes and types were identified (and subtyping is identified with subclassing), such inconveniences would seem more natural.

However, in BeCecil, there is also another reason for permitting specializers that do not conform to their declared argument types. The reason is that if one programs using an ADT pattern, then one will hide object names but make conformance relationships public. (See Section 2.6 for an example.) Doing this would prevent clients from inheriting certain operations, but that is a property of ADT languages (such as Ada 83 and CLU) as well. If one did that, then it would be necessary for client functions to use a specializer other than the class name, because the class name would be hidden. So to support this ADT pattern of completely hiding an implementation, and for reasons of convenience, the specializers of formal arguments need not conform to their declared types.

Appendix E Notations for Finite Functions, Stores, Environments, and Relations

E.1 Finite Functions

Finite functions are widely used in semantics to model environments and stores. In this section we describe notation for finite functions and finite, binary relations. There is an embedding of finite functions into relations, which we exploit to more easily describe finite functions. This embedding is just the traditional mathematical view that a function, f, is the same as its graph, the relation $\{(x, f(x)) \mid x \in dom(f)\}$.

We first describe finite functions and how to extract their graphs.

$f \in FiniteFun[A,B] = A \to B_\perp$
$dom: FiniteFun[A,B] \to Powerset(A)$
$dom(f) = \{a \mid f(a) \neq \perp\}$
$graph: FiniteFun[A,B] \to FiniteRel[A,B]$
$graph(f) = \{(a, b) \mid a \in dom(f), f(a) = b\}$

Finite relations are modeled as sets of ordered pairs. We consider the empty set, $\{\}$, singleton sets, and set union as generators for sets when writing inductive definitions over sets.

$R \in FiniteRel[A,B] = Powerset(A \times B)$
$dom: FiniteRel[A,B] \to Powerset(A)$
$dom(R) = \{a \mid (a, b) \in R\}$
$maps: FiniteRel[A,B] \times A \to Powerset(B)$
$maps(R, a) = \{b \mid (a, b) \in R\}$
$isFunction: FiniteRel[A,B] \to Boolean$
$isFunction(R) = (\forall a\ .\ a \in dom(R) \Rightarrow |maps(R, a)| = 1)$
$apply: FiniteRel[A,B] \times A \to B_\perp$
$apply(R, a) = \textbf{if}\ isFunction(R) \wedge a \in dom(R)\ \textbf{then}\ (\textbf{let}\ \{b\} = maps(R, a)\ \textbf{in}\ b)\ \textbf{else}\ \perp$
$toFun: FiniteRel[A,B] \to FiniteFun[A,B]$
$toFun(R) = \lambda a\ .\ apply(R, a)$

It is well known that $toFun(graph(f)) = f$, and that if $isFunction(R)$ holds, then $graph(toFun(R)) = R$. Thus throughout the rest of this paper, we freely use this dual point of view, writing definitions from whichever point of view is suitable, omitting the various uses of *graph* and *toFun*.

Some of the following notation is from David A. Schmidt's book *The Structure of Typed Programming Languages* (MIT Press, 1994).

$\cdot[\cdot := \cdot]: FiniteFun[A,B] \times A \times B \to FiniteFun[A,B]$
$f[a_1 := b] = \lambda a_2\ .\ \textbf{if}\ a_2 = a_1\ \textbf{then}\ b\ \textbf{else}\ f(a_2)$
$disjoint: FiniteRel[A,B] \times FiniteRel[A,B] \to Boolean$
$disjoint(f_1, f_2) = ((dom(f) \cap dom(f)) = \{\})$
$\cdot \cup \cdot: FiniteFun[A,B] \times FiniteFun[A,B] \to FiniteFun[A,B]_\perp$
$f_1 \cup f_2 = \textbf{if}\ disjoint(f_1, f_2)\ \textbf{then}\ f_1 \cup f_2\ \textbf{else}\ \perp$
$\cdot \uplus \cdot: FiniteFun[A,B] \times FiniteFun[A,B] \to FiniteFun[A,B]$
$f_1 \uplus f_2 = \lambda a\ .\ \textbf{if}\ a \in dom(f_2)\ \textbf{then}\ f_2(a)\ \textbf{else}\ f_1(a)$
$bind: A^* \times B^* \to FiniteFun[A,B]_\perp$
$bind((a_1,...,a_n), (b_1,...,b_n)) = \{(a_1,b_1)\} \cup ... \cup \{(a_n,b_n)\}$

Following Schmidt, we also use the notation $a{:}b$ for the ordered pair (a, b). Such a pair is called a *binding*. Thus we have the following abbreviations.

$(a{:}b) \in f \equiv (a, b) \in f$
$\{a_1{:}b_1,...,a_n{:}b_n\} \equiv \{(a_1,b_1),...,(a_n,b_n)\}.$

E.2 Stores

A *store* maps locations to stored values.

$\sigma \in Store = FiniteFun[Location, StoredValue]$
$l \in Location = Nat$
emptystore: Store
$emptystore = \{\}$
$allocate : Store \times StoredValue \to Location \times Store$
$allocate(\sigma,v) = (l, \sigma[l := v])$, **where** $l \notin dom(\sigma)$
$lookup: Store \times Location \to StoredValue_\perp$
$lookup(\sigma,l) = \sigma(l)$

E.3 Environments

An *environment* maps identifiers to denotable values.

$\eta \in Environment = FiniteFun[Identifier, DenotableValue]$
$emptyenv: Environment$
$emptyenv = \{\}$
$applyenv: Environment \times Identifier \to DenotableValue_\perp$
$applyenv(\eta, I) = \eta(I)$

E.4 Relations

The following additional operations on the type *FiniteRel*[*A*,*A*] are also useful.

$\cdot^+ : FiniteRel[A,A] \to FiniteRel[A,A]$
$R^+ = R \cup \{(a_1, a_2) \mid a_1\ R\ b$ and $b\ R^+\ a_2$, for some $b \in A\}$
$\cdot* : FiniteRel[A,A] \to FiniteRel[A,A]$
$R* = R^+ \cup \{(a, a) \mid a \in dom(R))\}$
$cyclic: FiniteRel[A,A] \to Boolean$
$cyclic(R) = (\exists a\ .\ a \in dom(R) \wedge a\ R^+\ a)$
$\cdot \circledast \cdot : FiniteRel[A,A] \times FiniteRel[A,A] \to FiniteRel[A,A]_\perp$
$R_1 \circledast R_2 =$ **if** $cyclic(R_1 \cup R_2)$ **then** $\perp$ **else** $R_1 \cup R_2$