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

Craig Chambers and Gary T. Leavens

TR #96-17a
December 1996, Revised April 1997

An earlier version of this report, minus the appendices, appeared in the proceedings of *The Fourth International Workshop on Foundations of Object-Oriented Languages FOOL 4, Paris, France.* The full report also appears as a technical report from the department of Computer Science and Engineering, University of Washington, Seattle, WA, numbered UW-CSE-96-12-02.

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.

Copyright © Craig Chambers and Gary T. Leavens, 1996.

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

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

May 5, 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: applicable 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_obj`. (We use the suffix `_obj`, which stands for "objects," to distinguish objects from types in examples.)

```
object Point_obj
```

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_obj` inherits from an object named `any`, and declares another object, `CP_obj` (for colored points), that inherits from `Point_obj` and an object named `Color_obj`. The object named `any` is used in our examples as an object from which (almost) all objects inherit.

```
Point_obj inherits any
object CP_obj
CP_obj inherits Point_obj
CP_obj inherits Color_obj
```

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_obj` as follows.

```
object myPoint
myPoint inherits Point_obj
```

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_obj` (which itself inherits from `any`).

```
object equal
equal inherits GenericFun_obj
```

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_obj` arguments.

```
equal has method(p1@Point_obj, p2@Point_obj) {
   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_obj`, because the specializers of the method are both `Point_obj`.

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_obj` and objects that inherit from `int_obj`). 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_obj
x has storage(p@Point_obj) := 0
object y
y inherits GenericFun_obj
y has storage(p@Point_obj) := 0
```

Since `myPoint` inherits from `Point_obj`, all methods and storage tables that have formal arguments that are specialized on `Point_obj` 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_obj
xy has acceptor(p@Point_obj) := 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_obj
initialize has method(p@Point_obj, i@int_obj, j@int_obj) {
   x(p) := i;
   y(p) := j;
   p
}
object mkPoint
mkPoint inherits GenericFun_obj
mkPoint has method(x@int_obj, y@int_obj) {
   object res
   res inherits Point_obj
   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 prototype `Region_obj` are given elsewhere.)

```
object Grayscale_obj
Grayscale_obj inherits Color_obj
hide
   -- private declarations
   object scale
   scale inherits GenericFun_obj
   scale has storage(c@Grayscale_obj) := 0
in
   -- public declarations
   initialize has method(c@Grayscale_obj, intensity@float_obj) {
      scale(c) := truncate(multiply(min(max(0.0, intensity), 1.0), 255.0));
      c
   }
   intensity has method(c@Grayscale_obj) {
      divide(mkFloat(scale(c)), 255.0)
   }
   paint has method(c@Grayscale_obj, r@Region_obj) { ... }
end
intensity has acceptor(c@Grayscale_obj) := 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_obj`, 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_obj`, 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_obj
mkGrayscale has method(intensity@float_obj) {
   object res
   res inherits Grayscale_obj
   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_obj` 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 impostors 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_obj`, as a subclass of `Point_obj` and `Grayscale_obj`. Notice how the `equal` and `initialize` methods use directed actual arguments to call the methods specialized on `Point_obj` and `Grayscale_obj` as part of their work. This is similar to the use of `super` in Smalltalk, or `Point_obj::` and `Grayscale_obj::` in C++.

```
object GrayPoint_obj -- grayscale points
GrayPoint_obj inherits Point_obj
GrayPoint_obj inherits Grayscale_obj
initalize has method(gsp@GrayPoint_obj, i@int_obj, j@int_obj,
                     intensity@float_obj) {
   initialize(gsp@Grayscale_obj, intensity);     -- calls initialize for Grayscale_obj
   initialize(gsp@Point_obj, i, j);              -- calls initialize for Point_obj
   gsp
}
object mkGSP
mkGSP inherits GenericFun_obj
mkGSP has method(i@int_obj, j@int_obj, intensity@float_obj) {
   object res
   res inherits GrayPoint_obj
   initalize(res, i, j, intensity)
}
equal has method(gsp1@GrayPoint_obj, gsp2@GrayPoint_obj) {
   and(equal(intensity(gsp1), intensity(gsp2)),
       equal(gsp1@Point_obj, gsp2@Point_obj))     -- calls equal for Point_obj
}
```

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 inherits GenericFun_obj
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_obj
while has method(c@GenericFun_obj, s@GenericFun_obj) {
   object loop
   loop inherits GenericFun_obj
   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.

2.3.1 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.

⌈**object** *I* **inherits** $CN_1,...,CN_n$⌉ ≡
object *I*
⌈*I* **inherits** $CN_1,...,CN_n$⌉

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_obj`⌉

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.

2.3.2 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`⌉

2.3.3 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 make 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_obj) {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_obj) { 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*.

$$\lceil \textbf{new } CN \rceil \equiv \lceil \{\ \textbf{object } I \textbf{ inherits } CN \ \}^{I} \rceil,$$ 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).

$$\lceil \textbf{anon } GF \rceil \equiv \lceil \{\ \textbf{object } I \textbf{ inherits } \texttt{GenericFun_obj}\ I \textbf{ has } GF \ \}^{I} \rceil,$$ 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`.)

$$\lceil \textbf{let } I_1 = E_1, \ \ldots, \ I_n = E_n \textbf{ in } E_0 \rceil \equiv \lceil (\textbf{anon method}(I_1, \ldots, I_n)\{E_0\})(E_1, \ldots, E_n) \rceil$$

$$\lceil \textbf{let } I_1 = E_1; \ \ldots; \ I_n = E_n \textbf{ in } E_0 \rceil \equiv \lceil \textbf{let } I_1 = E_1 \textbf{ in } \ldots \textbf{ let } I_n = E_n \textbf{ in } E_0 \rceil$$

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

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

2.4 Some Small Examples

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

2.4.1 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_obj inherits any

object number_obj inherits any
object int_obj inherits number_obj
object float_obj inherits number_obj
object 1 inherits int_obj               -- numeric literals are considered to be identifiers
object 2 inherits int_obj
object 3 inherits int_obj
   ...
object 4.7 inherits float_obj
   ...
```

2.4.2 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; for an example, see the `and` method below.

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

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

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

2.4.4 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_obj` and `nonempty_obj`, a concrete object, `nil`, and a concrete class `cons_obj`. The abstract class `list_obj` 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_obj inherits collection_obj
gf head       -- to be implemented by subclasses
gf tail       -- to be implemented by subclasses
do has method(c@list_obj, b@GenericFun_obj) {
   if(isEmpty(c), nothing, [b(head(c)); do(tail(c), b)])
}
```

2.4.5 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_obj`.

```
object nil inherits list_obj
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!
```

2.4.6 nonempty lists and cons

The following gives the code for `nonempty_obj` and `cons_obj`. We use two classes so that one can inherit the `isEmpty` method from `nonempty_obj` more easily (in other nonempty list representations that do not want to inherit from `cons_obj`). The "fields" of `cons_obj` are hidden. The hiding of the fields makes `cons_obj` 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_obj` to initialize the hidden fields.

```
object nonempty_obj inherits list_obj
isEmpty has method(l@nonempty_obj) { false }

object default_list_elem inherits any

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

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

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

2.4.7 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_obj`. 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_obj`, `lower`(*o*) ≤ `upper`(*o*).

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

```
      l
   }
   tail has method(l@interval_obj) {
      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_obj) { lower(l) }
   equal has method(x@interval_obj, y@interval_obj) {
      and(equal(lower(x), lower(y)), equal(upper(x), upper(y)))
   }
end
fun mkInterval(lb@int_obj, ub@int_obj) {
   initialize(new interval_obj, lb, ub)
}
gf clone
clone has method(r@interval_obj) { assign(new interval_obj, r) }
```

2.4.8 gray_interval

The following example, which builds on the `interval` and `Grayscale_obj` 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_obj::equal` would in C++. By overriding `assign` and `clone`, the `tail` method is inherited without further rewriting.

```
object gray_interval_obj inherits interval_obj, Grayscale_obj
initialize has method(gi@gray_interval_obj, lb@int_obj, ub@int_obj,
                      intensity@float_obj) {
   initialize(gi@Grayscale_obj, intensity);
   initialize(gi@interval_obj, lb, ub)
}
assign has method(l@gray_interval_obj, r@gray_interval_obj) {
   initialize(l@Grayscale_obj, intensity(r));
   assign(l@interval_obj, r@interval_obj)
}
fun mkGrayInterval(lb@int_obj, ub@int_obj, intensity@float_obj) {
   initialize(new gray_interval_obj, lb, ub, intensity)
}
clone has method(gi@gray_interval_obj) {
   assign(new gray_interval_obj, gi)
}
equal has method(x@gray_interval_obj, y@gray_interval_obj) {
   and(equal(intensity(x), intensity(y)),
       equal(x@interval_obj, y@interval_obj))
}
```

2.4.9 array1

The following example shows how a storage table with two arguments can be used as a one-dimensional array. The class `array1_obj` 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_obj inherits collection_obj
hide
   object a_stor inherits any
   a_stor has storage(a@array1_obj, i@int_obj) := 0.0
   field max_i of array1_obj := 9
in
   initialize has method(a@array1_obj, n@int_obj) {
      max_i(a) := n;
      do(a, anon method(i){a_stor(a,i) := 0.0});
      a
   }
   isEmpty has method(a@array1_obj) {less(max_i(a), 0)}
   do has method(a@array1_obj, b@GenericFun_obj) {
      foreach(a, anon method(i){b(a_stor(a,i))})
   }
   fun foreach(a@array1_obj, b@GenericFun_obj) {
      var i := 0
      while([leq(i,max_i)], [b(i); i() := plus(i(), 1)])
   }
   gf sub -- short for "subscript"
   sub has method(a@array1_obj, i@int_obj) {
      ifTrue(and(leq(0,i), [leq(i, max_i(a))]),
             [a_stor(a, i)])
   }
   sub has acceptor(a@array1_obj, i@int_obj) := e {
      ifTrue(and(leq(0,i), [leq(i, max_i(a))]),
             [a_stor(a, i) := e; a])
   }
end
fun mkArray(n@int_obj) { initialize(new array1_obj, 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

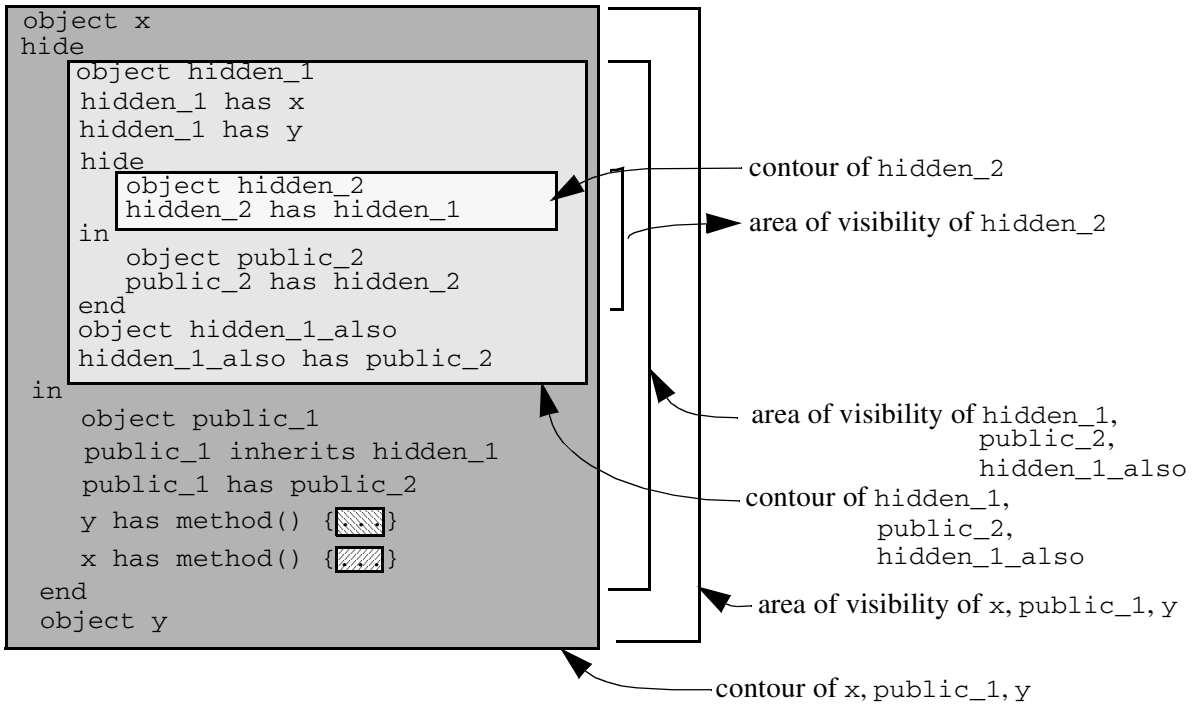

Figure 2-2: Illustration of the contours and areas of visibility of some declarations.

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_obj`. 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_obj` 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_obj, l2@interval_obj) {
                        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 affects inheritance. As a 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_obj, n2@int_obj) {...}
equal has method(n1@int_obj, n2@number_obj) {...}
equal has method(n1@int_obj, n2@int_obj) {...} -- 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_obj` 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_obj`. This is true, but the object `myStack` does know that it inherits from `Stack_obj` (and from `any`). So the application works if some ancestor of `myStack` is known to be `Stack_obj`.

```
object Stack_obj inherits any
gf push
push has method(s@Stack_obj, j@int_obj) {...}
fun f() {
   object myStack inherits Stack_obj
   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_obj` and `collection_obj` 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_obj` inherits from `collection_obj`. The variable `x` is assigned an object that inherits from `Stack_obj`, but which does not have `collection_obj` 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_obj inherits any
object Stack_obj inherits any
var g := anon method(x@collection_obj) {...}
var x := new Stack_obj
fun nested() {
   Stack_obj inherits collection_obj
   gf size
   size has method(s@collection_obj) {...}
   g() := size
}
fun nested_also() { x() := new Stack_obj }
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_obj` (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_obj`. Thus, behavioral properties cannot be guaranteed to hold for all objects that inherit from `Grayscale_obj`.

```
object bad inherits Grayscale_obj
initialize has method(b@bad, f@float_obj) { initialize(b,f) } -- loop forever!
intensity has method(b@bad) { intensity(b) }                  -- loop forever!
paint has method(b@bad, r@Region_obj) { 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_obj`, one can make a class that is like `Grayscale_obj`, but prevents impostors. Note that the acceptor for the generic function `intensity` has been moved inside the hide declaration, since otherwise it could not specialize on `GS2_obj`.

```
hide
   -- private declarations
   object GS2_obj
   field scale of GS2_obj := 0   -- sugared version of the previous declaration of scale
in
   -- public declarations
   GS2_obj inherits Color_obj
   initialize has method(c@GS2_obj, intensity@float_obj) {
      scale(c) := truncate(multiply(min(max(0.0, intensity), 1.0), 255));
      c
   }
   gf mkGS2
   mkGS2 has method(intensity@float_obj) {
      initialize(new GS2_obj, intensity)
   }
   intensity has method(c@GS2_obj) {
      divide(mkFloat(scale(c)), 255.0)
   }
   intensity has acceptor(c@GS2_obj) := v { initialize(c, v) }
   paint has method(c@GS2_obj, r@Region_obj) { ... }
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_obj` conforms to the type `Point`. This declaration makes `Point_obj` a prototype (or concrete object): it can be used in any place that any other object that conforms to `Point` can be used.

```
Point_obj 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_obj:int, j@int_obj: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_obj:Point)->int, (Point_obj: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_obj
Point_obj inherits any
Point_obj conforms Point -- Point_obj is a prototype
object equal
equal inherits GenericFun_obj
equal has method(p1@Point_obj:Point, p2@Point_obj:Point): boolean {
   and(equal(x(p1), x(p2)), equal(y(p1), y(p2)))
}
object x
x inherits GenericFun_obj
x has storage(p@Point_obj:Point) := 0:int
object y
```

```
y inherits GenericFun_obj
y has storage(p@Point_obj:Point) := 0:int
object initialize
initialize inherits GenericFun_obj
initialize has method(p@Point_obj:Point,
                      i@int_obj:int, j@int_obj:int): Point {
   x(p) := i;
   y(p) := j;
   p
}
object mkPoint
mkPoint inherits GenericFun_obj
mkPoint has method(x@int_obj:int, y@int_obj:int): Point{
   object res
   res inherits Point_obj
   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_obj
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 $(T^*) \rightarrow T_r$ 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_obj
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, $(T_1,...,T_n)->T_r$ and $(S_1,...,S_n)->S_r$, 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 $(T_1,...,T_n)->T_r$ is a subtype of $(S_1,...,S_n)->T_r$ by the usual contravariant subtyping rule for function types [Cardelli 88]. Therefore, if an object conforms to $(T_1,...,T_n)->T_r$ it must also conform to $(S_1,...,S_n)->T_r$. 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_obj
negate has method(x@int_obj:number): int {...}      -- these do not type check
negate has method(x@float_obj: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_obj`. 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_obj
truncate has method(x@number_obj:number): int {...}      -- these do not type check
truncate has method(x@float_obj: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.

3.3.1 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_obj` and the corresponding type `X`, one often wishes, for example, to declare that an object inherits `X_obj` 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** $TN_1_obj, ..., TN_n_obj$ **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() :=** $\lceil E \rceil$**:**$\lceil T \rceil$

$\lceil$ **fun** *I*(*F**) : *T* {*B*} $\rceil \equiv$
    $\lceil$ **gf** *I* $\rceil$
    I **has method**(*F**) : *T* {*B*}
$\lceil$ **field** *I*:T_1 **of** *CN*:T_0 := *E* $\rceil \equiv$
    $\lceil$ **gf** *I* $\rceil$
    I **has storage**(*x*@*CN*:$\lceil T_0 \rceil$) := $\lceil E \rceil$:$\lceil T_1 \rceil$, where *x* is a fresh identifier.

3.3.2 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.

$\lceil$ *I* @ : *TN* $\rceil \equiv$
    I @ *TN_obj* : *TN*

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

$\lceil$ *I* : *T* $\rceil \equiv$
    I @ any : $\lceil T \rceil$

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

$\lceil$ *I* @ : (*T**) -> T_{n+1} $\rceil \equiv$
    I @ GenericFun_obj : ($\lceil T^* \rceil$) -> $\lceil T_{n+1} \rceil$

For example, we have the following expansion into BeCecil.

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

3.3.3 Typed Expression Sugars

The following sugar replaces the untyped version of the sugar for **new** given in Section 2.3.3.

$\lceil$ **new** *CN*:*T* $\rceil \equiv$
    $\lceil$ { **object** *I* **inherits** *CN* **conforms** *T*
    }^{I} $\rceil$ 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_obj*, one can use the following sugar.

$\lceil$ **new isa** *TN* $\rceil \equiv$
    $\lceil$ **new** *TN_obj*:*TN* $\rceil$

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 2.3.3.

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_obj inherits any

type number subtypes Top
type int subtypes number
type float subtypes number
object number_obj inherits any
object int_obj inherits number_obj
object float_obj inherits number_obj
object 1 isa int
object 2 isa int
object 3 isa int
   ...
object 4.7 isa float
   ...
```

3.4.1 Boolean

We add type declarations to the Booleans, from Section 3.4.1, 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_obj 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 }
```

3.4.2 collection

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

```
type collection subtypes Top
type collElem subtypes Top
object collection_obj 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()
}
```

3.4.3 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_obj inherits collection_obj   -- 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)])
}
```

3.4.4 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!
```

3.4.5 nonempty lists

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

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

3.4.6 cons

As with the empty list, we do not declare a subtype of `list` for lists implemented by `cons_obj`. 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_obj inherits nonempty_obj conforms list
hide
   field hd:listElem of cons_obj:list := default_list_elem
   field tl:list of cons_obj:list := nil
in
   initialize has method(c@cons_obj:list, x:listElem, l:list): list {
      hd(c) := x;
      tl(c) := l;
      c
   }
   tail has method(l@cons_obj:list): list {tl(l)}
   head has method(l@cons_obj:list): listElem {hd(l)}
   tail has acceptor(l@cons_obj:list) := new_tail:list {
      tl(l) := new_tail
   }
   head has acceptor(l@cons_obj:list) := new_head:listElem {
      hd(l) := new_head
   }
end
fun cons(x:listElem, l:list): list {
   initialize(new cons_obj: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_obj:list, y@nonempty_obj: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.

3.4.7 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_obj inherits any conforms link
object nil conforms link
hide
   field nxt:link of link_obj: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
```

3.4.8 alist

The following example, one way to implement association lists, shows some interesting uses of private inheritance and subtyping; in the hidden declarations, `alist_obj` is made to inherit from `link_obj` 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_obj inherits collection_obj conforms alist
hide
   alist_obj isa link
   field ky:key of alist_obj:alist := default_key
   field vl:value of alist_obj: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_obj, 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_obj:alist, k, v, next)
}
```

3.4.9 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_obj: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 descendants ("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_obj` conform to the type `T` is falsified in the body of the function `nested`, because `Another_T_obj` is a object that conforms to `T`, and it does not inherit from `First_T_obj`.

```
type T
gf equal conforms (T, T) -> boolean
object First_T_obj inherits any conforms T
equal has method(p1@First_T_obj:T, p2@First_T_obj:T): boolean { true }
fun nested():T {
   object Another_T_obj inherits any conforms T
   Another_T_obj
}
equal(nested(), First_T_obj) -- 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_obj` and `CircusPerf_obj`, each conforms to a different type, and the two types are unrelated. If however, `CircusPerf` were declared as a subtype of `Elephant` within `barnum`, then the nested contour would be illegal.

```
type Elephant subtypes Top
object Elephant_obj inherits any conforms Elephant
fun haul(e@:Elephant, thing:Top, distance@:int): void { ... }
type CircusPerf subtypes Top
object CircusPerf_obj inherits any conforms CircusPerf
fun space_for_act(cp@:CircusPerf): int { ... }
fun barnum(): void {
   type CircusElephant subtypes Elephant, CircusPerf
   object floppy inherits Elephant_obj, CircusPerf_obj
   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_obj
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_obj
     }⌉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. Since storage tables in BeCecil can also be specialized, one can also covariantly-specialize the equivalent of instance variables in BeCecil.)

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.)

$\lceil$ **[** l_1=ς $(x_1)\, b_1$, ..., l_n=ς $(x_n)\, b_n$ **]** $\rceil \equiv$

$\lceil$ **{ 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
} $\rceil$, where x and I are fresh.

$\lceil a.l \rceil \equiv$

$\lceil$ **let** I **=** a **in** $(I(l))(I)$ $\rceil$

$\lceil a.l \Leftarrow \varsigma\,(x)\, b \rceil \equiv$

$\lceil a(l) \rceil$ **:=** $\lceil$ **anon method**(x) **{** b **}** $\rceil$

$\lceil$ **clone**(a) $\rceil \equiv$

$\lceil$ **let** I_a **=** a
in { gf I
I **has** I_a
I
} $\rceil$, 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.

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.