Appendix A Dynamic Semantics

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

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

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

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

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

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

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

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

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

A.1.1 Objects and Object Identities

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

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

oid: $Object \rightarrow ObjectId$
$oid(id, \iota, is, as) = id$

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

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

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

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

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

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

A.1.2 Direct Inheritance, Inheritance Relations, and Ancestry

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

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

```
Larch_obj inherits SpecLang_obj
```

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

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

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

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

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

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

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

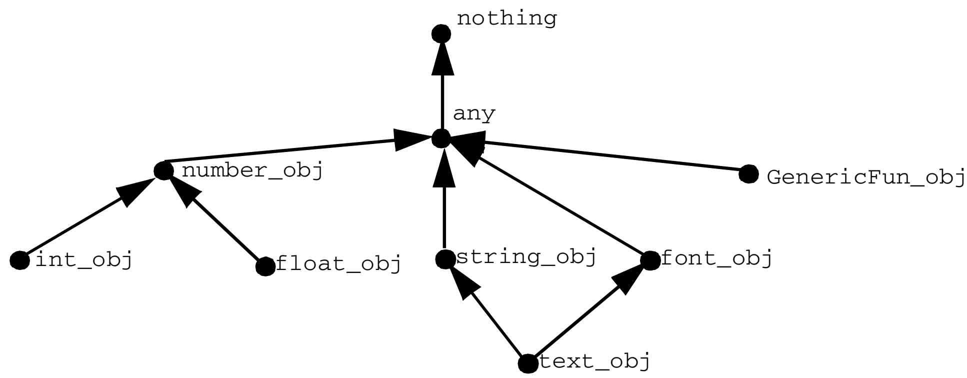

Figure A-5: A picture of an inheritance relation.

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

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

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

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

A.1.4 Closures and Storage Tables

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

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

[invoke closure]	$$\frac{(\eta',\iota) / \sigma \vdash B \Rightarrow v/\sigma'}{\sigma \vdash \mathit{invoke}(\mathbf{closure}((I_1,\ldots,I_n), B, (\eta,\iota)), (v_1,\ldots,v_n)) \Rightarrow v/\sigma'}$$	where $n \geq 0$, $\eta' =_s \eta \uplus \mathit{bind}((I_1,\ldots,I_n), (\mathbf{arg}(v_1),\ldots,\mathbf{arg}(v_n))$
[invoke storage 1]	$\sigma \vdash \mathit{invoke}(\mathbf{storage}(l), (v_1,\ldots,v_n)) \Rightarrow s/\sigma$	where $n \geq 0$, $\mathit{lookup}(\sigma, l) =_s \mathbf{table}(f, s_d)$, $((\mathit{oid}(v_1),\ldots,\mathit{oid}(v_n)){:}s) \in f$
[invoke storage 2]	$\sigma \vdash \mathit{invoke}(\mathbf{storage}(l), (v_1,\ldots,v_n)) \Rightarrow s_d/\sigma$	where $n \geq 0$, $\mathit{lookup}(\sigma, l) =_s \mathbf{table}(f, s_d)$, $(\mathit{oid}(v_1),\ldots,\mathit{oid}(v_n)) \notin \mathit{dom}(f)$

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

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

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

[assign storage]	$\sigma \vdash \mathit{assign}(\mathbf{storage}(l), (v_1,\ldots,v_n), s) \Rightarrow \sigma'$	where $n \geq 0$, $\mathit{lookup}(\sigma, l) =_s \mathbf{table}(f, s_d)$ $t = \mathbf{table}(f[(\mathit{oid}(v_1),\ldots,\mathit{oid}(v_n)) := s], s_d)$, $\sigma' = \sigma[l := t]$
[assign acceptor]	$$\frac{(\eta',\iota) / \sigma \vdash B \Rightarrow v/\sigma'}{\sigma \vdash \mathit{assign}(\mathbf{closure}((I_1,\ldots,I_n,I_{n+1}),B,(\eta,\iota)), (v_1,\ldots,v_n), s) \Rightarrow \sigma'}$$	where $n \geq 0$, $\eta' =_s \eta \uplus \mathit{bind}((I_1,\ldots,I_n,I_{n+1}), (\mathbf{arg}(v_1),\ldots,\mathbf{arg}(v_n),\mathbf{arg}(s))$

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

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

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

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

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

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

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

A.1.5 The Model of Acceptors

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

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

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

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

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

A.1.6 Orderings on Cases and Case Sets

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

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

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

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

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

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

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

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

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

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

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

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

$\cdot \cup\!\!\!\cdot\ \cdot : Powerset(Case) \times Powerset(Case) \to Powerset(Case)_\perp$

$cs_1 \cup\!\!\!\cdot\ cs_2 =$ **if** $nonclashing(cs_1, cs_2)$ **then** $cs_1 \cup cs_2$ **else** $\perp$

$\cdot \uplus \cdot : Powerset(Case) \times Powerset(Case) \to Powerset(Case)$

$cs_1 \uplus cs_2 = cs_2 \cup \{c \mid c \in cs_1, specializers(c) \notin specializers(cs_2)\}$

A.1.7 Actual Arguments and Applicability

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

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

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

A.1.8 Processing Applications and Assignments

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

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

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

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

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

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

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

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

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

A.1.9 Contexts, Environments, and Denotable Values

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

A.1.10 Elaborateds

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

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

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

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

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

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

A.1.11 Object Counters and the Store

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

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

A.2 Dynamic Semantics

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

A.2.1 Programs

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

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

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

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

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

A.2.2 Recursive Declaration Sequences

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

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

[rec-decl-sequence]

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

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

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

[block]

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

Figure A-18: Dynamic semantics for blocks.

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

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

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

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

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

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

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

A.2.3 Blocks

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

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

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

Figure A-19: Dynamic semantics of declaration sequences.

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

A.2.4 Declaration Sequences

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

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

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

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

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

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

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

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

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

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

[object declaration]	$\kappa/\sigma \vdash$ **`object`** I ➤ $(\{I{:}id\}, \{\}, \{\})/\sigma'$	where $lookup(\sigma, l_{idcounter}) =_s$ **ocounter**(id), $\sigma' = \sigma[l_{idcounter} :=$ **ocounter**$(id+1)]$
[inherits declaration]	$$\frac{\eta \vdash CN_1 \Rightarrow_{class} o_1, \quad \eta \vdash CN_2 \Rightarrow_{class} o_2}{(\eta,\iota)/\sigma \vdash CN_1\ \textbf{inherits}\ CN_2 \blacktriangleright (\{\},\{(id_1, id_2)\},\{\})/\sigma}$$	where $id_1 = oid(o_1)$, $id_2 = oid(o_2)$
[extension declaration]	$$\frac{\eta \vdash CN \Rightarrow_{class} o, \quad (\eta,\iota)/\sigma \vdash GF \Rightarrow (is,ags)/\sigma',}{(\eta,\iota)/\sigma \vdash CN\ \textbf{has}\ GF \blacktriangleright (\{\}, \{\}, \{id{:}(is,ags)\})/\sigma'}$$	where $id = oid(o)$
[hide declaration]	$$\frac{(\eta_0,\iota_0)/\sigma_0 \vdash RDS \blacktriangleright (\eta',\iota')/\sigma, \quad (\eta',\iota')/\sigma \vdash D^* \blacktriangleright (\omega'',\iota'',h'')/\sigma'}{(\eta_0,\iota_0)/\sigma_0 \vdash \textbf{hide}\ RDS\ \textbf{in}\ D^*\ \textbf{end} \blacktriangleright (\omega'',\iota'',h'')/\sigma'}$$	where $(\forall I\ .\ I \in dom(\omega'') \Rightarrow \eta_0(I) = \eta'(I))$

Figure A-20: Dynamic semantics of declarations.

A.2.5 Declarations

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

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

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

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

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

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

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

Figure A-21: Dynamic semantics of class names.

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

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

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

A.2.6 Class Names

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

A.2.7 Generic Function Attributes

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

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

$copyStorage\colon CaseSet \times Store \to (CaseSet \times Store)_\perp$

$copyStorage(\{c_1,\ldots,c_n\}, \sigma) =$

$\quad \mathbf{let}\ ((c'_1,\ldots,c'_n),\sigma_n) = copylist((c_1,\ldots,c_n), \sigma)\ \mathbf{in}\ (\{c'_1,\ldots,c'_n\}, \sigma_n)$

$copylist\colon Case^* \times Store \to (Case^* \times Store)_\perp$

$copylist((),\sigma) = ((),\sigma)$

$copylist((c_1,c_2,\ldots,c_n),\sigma) =$

$\quad \mathbf{let}\ ((c'_1,\sigma_1) = copy1(c_1,\sigma)\ \mathbf{in\ let}\ ((c'_2,\ldots,c'_n),\sigma_n) = copylist((c_2,\ldots,c_n),\sigma_1)\ \mathbf{in}\ ((c'_1,c'_2,\ldots,c'_n),\sigma_n)$

$copy1\colon Case \times Store \to (Case \times Store)_\perp$

$copy1((idv,\mathbf{closure}((I_1,\ldots,I_m),B,\kappa)), \sigma) = ((idv,\mathbf{closure}((I_1,\ldots,I_m),B,\kappa)), \sigma)$

$copy1((idv,\mathbf{storage}(l)), \sigma) =$

$\quad \mathbf{let}\ (l',\sigma') = allocate(\sigma, lookup(\sigma,l))\ \mathbf{in}\ ((idv,\mathbf{storage}(l')), \sigma')$

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

[identifier attribute]
$$\frac{\eta \vdash I \Rightarrow (id, \iota, is, ags)}{(\eta,\iota)/\sigma \vdash I \Rightarrow (is',ags')/\sigma'}$$
where $(is',\sigma') =_s copyStorage(is,\sigma)$, $ags' = \{(idv,b) \mid (idv,b) \in is', b = \mathbf{storage}(l)\} \cup \{(idv,b) \mid (idv,b) \in ags, b = \mathbf{closure}((I_1,...,I_n),B,(\eta,\iota))\}$

[method attribute]
$$\frac{\eta \vdash CN_1 \Rightarrow_{class} o_1, ..., \eta \vdash CN_n \Rightarrow_{class} o_n}{(\eta,\iota)/\sigma \vdash \texttt{method(}I_1\texttt{@}CN_1,...,I_n\texttt{@}CN_n\texttt{) \{}B\texttt{\}} \Rightarrow (is,\{\})/\sigma}$$
where $n \geq 0$, $is = \{((oid(o_1),...,oid(o_n)), \mathbf{closure}((I_1,...,I_n), B, (\eta,\iota)))\}$

[storage attribute]
$$\frac{\eta \vdash CN_1 \Rightarrow_{class} o_1, ..., \eta \vdash CN_n \Rightarrow_{class} o_n, \quad (\eta,\iota)/\sigma \vdash E \Rightarrow v/\sigma'}{(\eta,\iota)/\sigma \vdash \texttt{storage(}I_1\texttt{@}CN_1,...,I_n\texttt{@}CN_n\texttt{) :=} E \Rightarrow (cs,cs)/\sigma''}$$
where $n \geq 0$, $(l,\sigma'') = allocate(\sigma', \mathbf{table}(\{\}, v))$, $cs = \{((oid(o_1),...,oid(o_n)), \mathbf{storage}(l))\}$

[acceptor attribute]
$$\frac{\eta \vdash CN_1 \Rightarrow_{class} o_1, ..., \eta \vdash CN_n \Rightarrow_{class} o_n}{(\eta,\iota)/\sigma \vdash \texttt{acceptor(}I_1\texttt{@}CN_1,...,I_n\texttt{@}CN_n\texttt{) :=} I \texttt{ \{}B\texttt{\}} \Rightarrow (\{\},ags)/\sigma}$$
where $n \geq 0$, $ags = \{((oid(o_1),...,oid(o_n)), \mathbf{closure}((I_1,...,I_n,I), B, (\eta,\iota)))\}$

[combined-attribute]
$$\frac{\kappa/\sigma \vdash GF_1 \Rightarrow (is_1,ags_2)/\sigma_1, \quad \kappa/\sigma_1 \vdash GF_2 \Rightarrow (is_2,ags_2)/\sigma_2}{\kappa/\sigma \vdash GF_1 \texttt{ \& } GF_2 \Rightarrow (is,ags)/\sigma_2}$$
where $is = is_1 \uplus is_2$, $ags = ags_1 \uplus ags_2$

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

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

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

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

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

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

[identifier expression]

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

[application expression]

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

[assignment expression]

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

[sequence expression]

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

Figure A-23: Dynamic semantics of expressions.

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

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

A.2.8 Expressions

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

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

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

For example, the following block returns 27.

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

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

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

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

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

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

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

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

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

A.2.9 Identifiers

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

A.2.10 Actual Arguments

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

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

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

[identifier 1]	$\eta \vdash I \Rightarrow o$	where $(I{:}\mathbf{class}(o)) \in \eta$
[identifier 2]	$\eta \vdash I \Rightarrow o$	where $(I{:}\mathbf{arg}(o)) \in \eta$

Figure A-24: Dynamic semantics of identifiers.

[undirected actual]

$$\frac{\kappa/\sigma \vdash E \Rightarrow v/\sigma'}{\kappa/\sigma \vdash E \Rrightarrow (v, \{\})\ /\ \sigma'}$$

[directed actual]

$$\frac{(\eta,\iota)/\sigma \vdash E \Rightarrow v/\sigma',\quad \eta \vdash CN_1 \Rightarrow_{class} o_1, \ldots, \eta \vdash CN_n \Rightarrow_{class} o_n}{(\eta,\iota)/\sigma \vdash E@CN_1,\ldots,CN_n \Rrightarrow (v, \{id_1,\ldots,id_n\})\ /\ \sigma'}$$

where $n \geq 0$, $id_1 = oid(o_1), \ldots, id_n = oid(o_n)$

Figure A-25: Dynamic semantics of actual arguments.

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

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

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

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

B.2 Generic Functions are Named, not Methods

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

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

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

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

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

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

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

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

B.4 Use of Declarations instead of Expressions

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

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

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

B.5 Distinction between Class Names and Identifiers

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

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

B.6 Mutual Recursion in Declarations

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

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

```
object complex_obj inherits any
object cartesian_obj inherits complex_obj
object polar_obj inherits complex_obj
object addc inherits GenericFun_obj
addc has method(x@cartesian_obj, y@complex_obj) {...}
addc has method(x@complex_obj, y@polar_obj) {...}
-- the above is not consistent by itself
addc has method(x@complex_obj, y@complex_obj) {...}
addc has method(x@polar_obj, y@polar_obj) {...}
```

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

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

B.7 The Model of Objects

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

A fundamentally different semantics would have been to have two different kinds of values: objects and generic functions. In this case objects would not have any value at all, aside from their identity. However, unless generic functions were treated differently than objects, there would be no point in making the distinction. For example, one could say that generic functions did not inherit from any objects, which would prohibit them from being stored or passed around. But if that were the case, the language would be rigidly first-order, and other constructs would have to be added to handle control structures, etc.

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

B.8 Inheritance

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

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

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

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

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

B.9 Extension vs. Inheritance

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

B.10 Storage Tables

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

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

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

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

$$\ulcorner \textbf{var } I \textbf{ := } E \urcorner \equiv$$
$$\quad \ulcorner \textbf{gf } I \urcorner$$
$$\quad I_1 \textbf{ has method()\{} \ulcorner E \urcorner \textbf{\}}$$

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

$$\ulcorner E_{var} \textbf{ () := } E_{val} \urcorner \equiv$$
$$\quad \ulcorner \textbf{let } I_{val} = E_{val} \textbf{ in}$$
$$\quad E_{var} \textbf{ \&:= method()\{} I_{val} \textbf{\}} \urcorner, \qquad \text{where } I_{val} \text{ is a fresh identifier.}$$

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

$\lceil$ **field** I **of** CN **:=** $E \rceil \equiv$

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

  I **has method(**x**@**CN**)** **{**$\lceil E \rceil$**}**, where x is a fresh identifier.

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

$\lceil E_{fld}$ **(**E_{obj}**@**CN_{obj}**) :=** $E_{val} \rceil \equiv$

  $\lceil$ **let** I_{obj} **=** E_{obj}**;** I_{val} **=** E_{val}

  in {

    gf old_fld

    old_fld **has** E_{fld}

    fun new_fld**(**x**@**CN_{obj}**)** **{** if(**eq**(x,I_{obj}), **[**I_{val}**]**, **[**old_fld(x)**]**) **}**

    E_{fld} **&:=** new_fld**;**

    E_{fld}

  } $\rceil$, where x, I_{obj}, and I_{val} are fresh identifiers.

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

B.11 Initialization of Storage

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

B.12 Copying of Storage Tables during Extensions

Consider the following example declarations.

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

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

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

B.13 No Linearization

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

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

Appendix C Static Semantics

C.1 Domains for the Static Semantics

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

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

$$
\begin{array}{lll}
cl \in \mathit{Class} & = \mathbf{class}(\mathit{ObjectId}) \\
id \in \mathit{ObjectId} & = \mathit{Nat} \\
ids \in \mathit{ObjectIdSet} & = \mathit{Powerset}(\mathit{ObjectId}) \\
idv \in \mathit{ObjectIdVec} & = \mathit{ObjectId}^* \\
\mu \in \mathit{NamedType} & = \mathbf{namedtype}(\mathit{TypeId}) \\
tid \in \mathit{TypeId} & = \mathit{Nat} \\
tids \in \mathit{TypeIdSet} & = \mathit{Powerset}(\mathit{TypeId}) \\
\pi \in \mathit{TypeEnv} & = \mathit{FiniteFun}[\mathit{Identifier}, \mathit{TypeDenotable}] \\
\theta \in \mathit{TypeDenotable} & = \mathit{Class} + \mathbf{formal}(\mathit{TypeAttribute}) + \mathit{NamedType} \\
\tau \in \mathit{TypeAttribute} & = \mathit{NamedType} + \mathit{InvocableType} + \mathit{AssignableType} + \mathit{ExactType} \\
 & \qquad + \mathit{ConjunctiveType} + \mathit{DisjunctiveType} \\
\tau v \in \mathit{TypeAttributeVec} & = \mathit{TypeAttribute}^* \\
it \in \mathit{InvocableType} & = \mathbf{arrow}(\mathit{TypeAttribute}^* \times \mathit{TypeAttribute}) \\
at \in \mathit{AssignableType} & = \mathbf{assignable}(\mathit{TypeAttribute}^* \times \mathit{TypeAttribute}) \\
et \in \mathit{ExactType} & = \mathbf{exact}(\mathit{InvokeCaseSet} \times \mathit{AssignCaseSet}) \\
ics \in \mathit{InvokeCaseSet} & = \mathit{Powerset}(\mathit{ExactCase}) \\
acs \in \mathit{AssignCaseSet} & = \mathit{Powerset}(\mathit{ExactCase}) \\
ec \in \mathit{ExactCase} & = \mathit{ObjectId}^* \times \mathit{ExactBranch} \\
ecs \in \mathit{ExactCaseSet} & = \mathit{Powerset}(\mathit{ExactCase}) \\
eb \in \mathit{ExactBranch} & = \mathit{InvocableType} + \mathit{AssignableType} \\
\alpha \in \mathit{ConjunctiveType} & = \mathbf{andtype}(\mathit{PowerSet}(\mathit{TypeAttribute})) \\
\delta \in \mathit{DisjunctiveType} & = \mathbf{ortype}(\mathit{PowerSet}(\mathit{TypeAttribute})) \\
\zeta \in \mathit{Subtypes} & = \mathit{FiniteRel}[\mathit{TypeId}, \mathit{TypeAttribute}] \\
\iota \in \mathit{Inherits} & = \mathit{FiniteRel}[\mathit{ObjectId}, \mathit{ObjectId}] \\
\chi \in \mathit{Conforms} & = \mathit{FiniteRel}[\mathit{ObjectId}, \mathit{TypeAttribute}] \\
K \in \mathit{TypeContext} & = \mathit{TypeEnv} \times \mathit{Inherits} \times \mathit{ClassInfo} \times \mathit{Conforms} \times \mathit{Subtypes} \times \mathit{TypeInfo} \\
\phi \in \mathit{ClassInfo} & = \mathit{FiniteFun}[\mathit{ObjectId}, \mathit{Inherits}] \\
\psi \in \mathit{TypeInfo} & = \mathit{FiniteFun}[\mathit{TypeId}, \mathit{Inherits} \times \mathit{Powerset}(\mathit{ObjectId})] \\
\xi \in \mathit{TypeElaborated} & = \mathit{TypeEnv} \times \mathit{Inherits} \times \mathit{HasTypes} \times \mathit{Conforms} \times \mathit{Subtypes} \\
H \in \mathit{HasTypes} & = \mathit{FiniteFun}[\mathit{ObjectId}, \mathit{InvokeCaseSet} \times \mathit{AssignCaseSet}] \\
ta \in \mathit{TypedActual} & = \mathit{TypeAttribute} \times \mathit{Powerset}(\mathit{ObjectId}) \\
aa \in \mathit{ActualArgPair} & = \mathit{ObjectId} \times \mathit{Powerset}(\mathit{ObjectId}) \\
aav \in \mathit{ActualArgPairVec} & = \mathit{ActualArgPair}^*
\end{array}
$$

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

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

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

C.2 Static Semantics Domains and Auxiliary Functions

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

C.2.1 Type Denotables, Class and Type Identities

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

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

Syntax (T)	Semantics (τ)
TN	$\mathbf{namedtype}(tid)$
$(T_1,\ldots,T_n)$ `->` T_{n+1}	$\mathbf{arrow}((\tau_1,\ldots,\tau_n),\ \tau_{n+1})$
$(T_1,\ldots,T_n)$ `:=` T_{n+1}	$\mathbf{assignable}((\tau_1,\ldots,\tau_n),\ \tau_{n+1})$
exact{(CN**:**T**)->**T**, (**CN**:**T**):=**T**}**	$\mathbf{exact}(\{((id),\mathbf{arrow}((\tau),\tau))\},\{((id),\mathbf{assignable}((\tau),\tau))\})$
T_1 **&** T_2	$\mathbf{andtype}(\{\tau_1,\ \tau_2\})$
T_1 **|** T_2	$\mathbf{ortype}(\{\tau_1,\ \tau_2\})$
(CN_1**:**$T_1,\ldots,CN_n$**:**T_n**)** `->` T_{n+1}	$((id_1,\ldots,id_n),\ \mathbf{arrow}((\tau_1,\ldots,\tau_n),\ \tau_{n+1}))$
(CN_1**:**$T_1,\ldots,CN_n$**:**T_n**)** `:=` T_{n+1}	$((id_1,\ldots,id_n),\ \mathbf{assignable}((\tau_1,\ldots,\tau_n),\ \tau_{n+1}))$

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

C.2.2 Type Environments

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

$classes$: $TypeEnv \to FiniteFun[Identifier,\ ObjectId]$
$classes(\pi) = \{(I{:}id) \mid (I{:}\mathbf{class}(id)) \in \pi\}$

$types$: $TypeEnv \to FiniteFun[Identifier,\ TypeId]$
$types(\pi) = \{(I{:}tid) \mid (I{:}\mathbf{namedtype}(tid)) \in \pi\}$

C.2.3 Models of Types

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

The identities in a type attribute cannot be random. Instead, they must be object and type identities that are known in a given type environment. The rules that check this well-formedness condition for type attributes are given in Figures C-4 and C-5. These rules are used to check that type of a block is well-formed with respect to the type environment in which the block is checked (see Section C.3.4).

C.2.4 Direct Subtype and Subtype Relations

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

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

```
Larch subtypes SpecLang
```

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

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

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

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

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

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

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

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

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

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

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

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

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

[sub-base]	$\zeta \vdash \mathbf{namedtype}(tid) \leq_{sub} \tau$	where $(tid,\tau) \in \zeta$
[sub-refl]	$\zeta \vdash \tau \leq_{sub} \tau$	
[sub-trans]	$\dfrac{\zeta \vdash \tau_1 \leq_{sub} \tau_2, \quad \zeta \vdash \tau_2 \leq_{sub} \tau_3}{\zeta \vdash \tau_1 \leq_{sub} \tau_3}$	
[sub-and1]	$\dfrac{\zeta \vdash \tau \leq_{sub} \tau_1, \ldots, \zeta \vdash \tau \leq_{sub} \tau_n}{\zeta \vdash \tau \leq_{sub} \mathbf{andtype}(\{\tau_1,\ldots, \tau_n\})}$	where $n \geq 0$
[sub-and2]	$\zeta \vdash \mathbf{andtype}(\{\tau_1,\ldots, \tau_n\}) \leq_{sub} \tau_i$	where $n \geq 0$, $\tau_i \in \{\tau_1,\ldots, \tau_n\}$
[sub-or1]	$\dfrac{\zeta \vdash \tau_1 \leq_{sub} \tau, \ldots, \zeta \vdash \tau_n \leq_{sub} \tau}{\zeta \vdash \mathbf{ortype}(\{\tau_1,\ldots, \tau_n\}) \leq_{sub} \tau}$	where $n \geq 0$
[sub-or2]	$\zeta \vdash \tau_i \leq_{sub} \mathbf{ortype}(\{\tau_1,\ldots, \tau_n\})$	where $n \geq 0$, $\tau_i \in \{\tau_1,\ldots, \tau_n\}$
[sub-arrow]	$\dfrac{\zeta \vdash \tau'_1 \leq_{sub} \tau_1,\ldots, \zeta \vdash \tau'_n \leq_{sub} \tau_n, \quad \zeta \vdash \tau_r \leq_{sub} \tau'_r}{\zeta \vdash \mathbf{arrow}((\tau_1,\ldots,\tau_n),\tau_r) \leq_{sub} \mathbf{arrow}((\tau'_1,\ldots,\tau'_n),\tau'_r)}$	where $n \geq 0$
[sub-assignable]	$\dfrac{\zeta \vdash \tau'_1 \leq_{sub} \tau_1,\ldots, \zeta \vdash \tau'_n \leq_{sub} \tau_n, \quad \zeta \vdash \tau'_r \leq_{sub} \tau_r}{\zeta \vdash \mathbf{assignable}((\tau_1,\ldots,\tau_n), \tau_r) \leq_{sub} \mathbf{assignable}((\tau'_1,\ldots,\tau'_n), \tau'_r)}$	where $n \geq 0$
[sub-exact-translate]	$\zeta \vdash \mathbf{exact}(\{(idv_1,it_1),\ldots,(idv_n,it_n)\}, \{(idv'_1,at_1),\ldots,(idv'_m,at_m)\}) \leq_{sub} \mathbf{andtype}(\{it_1,\ldots,it_n, at_1,\ldots,at_m\})$	where $n \geq 0$, $m \geq 0$
[sub-gf]	$\dfrac{\zeta \vdash it_{f(1)} \leq_{sub} it'_1, \ldots, \zeta \vdash it_{f(n)} \leq_{sub} it'_n, \quad \zeta \vdash at_{g(1)} \leq_{sub} at'_1, \ldots, \zeta \vdash at_{g(o)} \leq_{sub} at'_o}{\zeta \vdash \mathbf{andtype}(\{it_1,\ldots,it_k, at_1,\ldots,at_l, \tau_1,\ldots,\tau_m\}) \leq_{sub} \mathbf{andtype}(\{it'_1,\ldots,it'_n, at'_1,\ldots,at'_o, \tau_1,\ldots,\tau_m\})}$	where $k \geq 0$, $l \geq 0$, $m \geq 0$, $n \geq 0$, $o \geq 0$, $(\forall i \in \{1,\ldots,n\} . (\exists f(i) \in \{1,\ldots,k\} .$ $it'_i = \mathbf{arrow}(\tau v_i,\tau_{ir})$ and $it_{f(i)} = \mathbf{arrow}(\tau v_{f(i)},\tau_{f(i)r}))$, $(\forall j \in \{1,\ldots,o\} . (\exists g(j) \in \{1,\ldots,l\} .$ $at'_j = \mathbf{assignable}(\tau v'_j,\tau'_{jr})$ and $at_{g(j)} = \mathbf{assignable}(\tau v'_{g(j)},\tau'_{g(j)r})))$

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

C.2.5 Direct Conformance and Conformance Relations

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

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

Figure C-8:Inference rule for conformance.

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

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

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

```
Larch_obj conforms Larch
```

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

C.2.6 Exact Types for Generic Functions and Exact Cases

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

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

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

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

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

$specializers: ExactCase \rightarrow ObjectId^*$
$specializers((id_1,...,id_m), eb)) = (id_1,...,id_m)$

$argtypes: ExactCase \rightarrow TypeAttribute^*$
$argtypes((id_1,...,id_m), \textbf{arrow}(\tau v,\tau_r))) = \tau v$
$argtypes((id_1,...,id_m), \textbf{assignable}(\tau v,\tau_r))) = \tau v$

$restype: ExactCase \rightarrow TypeAttribute$
$restype((id_1,...,id_m), \textbf{arrow}(\tau v,\tau_r))) = \tau_r$
$restype((id_1,...,id_m), \textbf{assignable}(\tau v,\tau_r))) = \tau_r$

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

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

$$\text{[overrides-eb]} \qquad \frac{\iota \vdash ec_1 \leq_{inh} ec_2}{\iota \vdash ec_1 \; overrides \; ec_2} \qquad \text{where } specializers(ec_1) \neq specializers(ec_2)$$

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

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

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

$nonclashing: Powerset(ExactCase) \times Powerset(ExactCase) \rightarrow Boolean$
$nonclashing(ecs_1, ecs_2) = ((specializers(ecs_1) \cap specializers(ecs_2)) = \{\})$

$\cdot \uplus \cdot: Powerset(ExactCase) \times Powerset(ExactCase) \rightarrow Powerset(ExactCase)_\perp$
$ecs_1 \uplus ecs_2 = \textbf{if } nonclashing(ecs_1, ecs_2) \textbf{ then } ecs_1 \cup ecs_2 \textbf{ else } \perp$

$\cdot \ominus \cdot: Powerset(ExactCase) \times Powerset(ExactCase) \rightarrow Powerset(ExactCase)$
$ecs_1 \ominus ecs_2 = ecs_2 \cup \{ec \mid ec \in ecs_1, specializers(ec) \notin specializers(ecs_2)\}$

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

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

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

$tenv: TypeContext \rightarrow TypeEnv$
$tenv(\pi, \iota, \phi, \chi, \zeta, \psi) = \pi$
$directSubtypes: TypeContext \rightarrow Subtypes$
$directSubtypes(\pi, \iota, \phi, \chi, \zeta, \psi) = \zeta$

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

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

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

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

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

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

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

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

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

[type-info-is]
$$(\chi,\zeta,\iota) \vdash \{tid_1,\ldots,tid_n\}\ type\text{-}info\text{-}is\ \psi \quad \text{where } n \geq 0,\ (\forall i\ .\ i \in \{1,\ldots,n\} \Rightarrow ids_i = \{id_c \mid (\chi,\zeta) \vdash id_c <: \textbf{namedtype}(tid_i)\}),\ \psi = \cup_{i \in \{1,\ldots,n\}} \{(tid_i{:}(\iota, ids_i))\}$$

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

C.2.8 Type Elaborateds

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

$tenv$: $TypeElaborated \rightarrow TypeEnv$
$tenv(\pi, \iota, H, \chi, \zeta) = \pi$

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

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

C.3 Type Checking Rules

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

C.3.1 Implementation-Side Type Checks

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

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

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

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

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

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

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

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

[impl-type-checks]

$$\frac{\begin{array}{c}(\pi,\iota,\chi,\zeta) \vdash id\ impls\ it_1,\ \ldots,\\ (\pi,\iota,\chi,\zeta) \vdash id\ impls\ it_n,\\ (\pi,\iota,\chi,\zeta) \vdash id\ impls\ at_1,\ \ldots,\\ (\pi,\iota,\chi,\zeta) \vdash id\ impls\ at_m\end{array}}{(\pi,\iota,\chi,\zeta) \vdash id\ impl\text{-}type\text{-}checks}$$

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

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

[impls-arrow]

$$\frac{\begin{array}{c}(\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{arrow}(\tau v,\tau_r)\ for\ idv_1,\\ \ldots,\\ (\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{arrow}(\tau v,\tau_r)\ for\ idv_m\end{array}}{(\pi,\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{arrow}(\tau v,\tau_r)}$$

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

[impls-assignable]

$$\frac{\begin{array}{c}(\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{assignable}(\tau v,\tau_r)\ for\ idv_1,\\ \ldots,\\ (\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{assignable}(\tau v,\tau_r)\ for\ idv_m\end{array}}{(\pi,\iota,\chi,\zeta) \vdash id\ impls\ \mathbf{assignable}(\tau v,\tau_r)}$$

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

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

[impls-for-arrow 1]

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

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

[impls-for-assignable]

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

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

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

[tc program]

$$\frac{\begin{array}{c}(\{\},\ \{\},\ \{\},\ \{\},\ \{\},\ \{\}) \vdash RDS \succ \mathrm{K},\\ \mathrm{K} \vdash B : \tau\end{array}}{\vdash RDS\ ;\ B\ :\ \tau}$$

Figure C-22: Static semantics for programs.

C.3.2 Type Checking Programs

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

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

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

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

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

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

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

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

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

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

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

C.3.3 Type Checking Recursive Declaration Sequences

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

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

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

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

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

Figure C-27: Static semantics for blocks.

$$\begin{aligned}&\mathit{exactConforms}: \mathit{TypeEnv} \times \mathit{HasTypes} \to \mathit{Conforms}\\ &\mathit{exactConforms}(\pi, H) = \{(id{:}\mathbf{exact}(ics,acs)) \mid (id{:}(ics,acs)) \in H\}\\ &\qquad \cup \{(id{:}\mathbf{exact}(\{\},\{\})) \mid id \notin dom(H),\ id \in range(classes(\pi))\}\end{aligned}$$

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

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

$$\begin{aligned}&\mathit{hasOrphans}: \mathit{TypeEnv} \times \mathit{HasTypes} \to \mathit{Boolean}\\ &\mathit{hasOrphans}(\pi, H) = (\exists id\ .\ id \in dom(H) \wedge id \notin range(classes(\pi)))\end{aligned}$$

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

C.3.4 Type Checking Blocks

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

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

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

[nests-ok]

$$\frac{\begin{array}{c}(\iota',\psi_0) \vdash (id_1,tid_1)\ \mathit{inherits\text{-}above},\\ \ldots,\\ (\iota',\psi_0) \vdash (id_n,tid_n)\ \mathit{inherits\text{-}above}\end{array}}{(\pi_0,\iota_0,\phi_0,\chi_0,\zeta_0,\psi_0) \vdash (\pi,\iota,\chi,\zeta)\ \mathit{nests\text{-}ok}}$$

where $n \geq 0$,
$\iota' =_s \iota_0 \circledast \iota$,
$\{(id_1,tid_1),\ldots,(id_n,tid_n)\} =$
$\{(id,tid) \mid (I{:}\mathbf{namedtype}(tid)) \in \pi_0,$
$id \in \mathit{conformers}(\pi_0 \uplus \pi, \chi_0 \cup \chi, \zeta_0 \cup \zeta,$
$\mathbf{namedtype}(tid))\}$

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

[inherits-above]

$$\frac{\begin{array}{c}\iota \vdash id \leq_{inh} id_c,\\ \iota' \vdash (id_{1,1},id_{1,2})\ \mathit{ordered},\\ \ldots,\\ \iota' \vdash (id_{n,1},id_{n,2})\ \mathit{ordered}\end{array}}{(\iota,\psi) \vdash (id,tid)\ \mathit{inherits\text{-}above}}$$

where $n \geq 0$,
$(tid{:}(\iota',ids)) \in \psi$,
$id_c \in ids$,
$\{(id_{1,1},id_{1,2}),\ldots,(id_{n,1},id_{n,2})\} =$
$\{(id_a,id_b) \mid id_a \in ids,\ id_b \in ids,$
$\iota \vdash id \leq_{inh} id_a,\ \iota \vdash id \leq_{inh} id_b\}$

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

[ordered1]

$$\frac{\iota \vdash id_1 \leq_{inh} id_2}{\iota \vdash (id_1,id_2)\ \mathit{ordered}}$$

[ordered2]

$$\frac{\iota \vdash id_2 \leq_{inh} id_1}{\iota \vdash (id_1,id_2)\ \mathit{ordered}}$$

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

[tc declaration sequence]

$$\frac{\begin{array}{c}\mathrm{K}_0 \vdash D_1 \succ \xi_1,\\ \ldots,\\ \mathrm{K}_0 \vdash D_n \succ \xi_n\end{array}}{\mathrm{K}_0 \vdash (D_1 \ldots D_n) \succ \xi}$$

where $n \geq 0$,
$(\forall i,j\ .\ 1 \leq i < j \leq n$
$\Rightarrow ((\mathit{range}(\mathit{types}(\mathit{tenv}(\xi_i))) \cap \mathit{range}(\mathit{types}(\mathit{tenv}(\xi_j))) = \{\,\})$
$\wedge\ (\mathit{range}(\mathit{classes}(\mathit{tenv}(\xi_i))) \cap \mathit{range}(\mathit{classes}(\mathit{tenv}(\xi_j))) = \{\,\}))),$
$\xi =_s \dot\cup_{i \in \{1,\ldots,n\}}\ \xi_i$

Figure C-28: Static semantics of declaration sequences.

C.3.5 Type Checking Declaration Sequences

The checking of a declaration sequence in an assumed type context produces a type elaborated. The type checking rule is given in Figure C-28. Note that the object and type identities used are required to be unique.

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

C.3.6 Type Checking Declarations

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

[tc object declaration]
$$K \vdash \texttt{object}\ I \succ (\{I{:}\mathbf{class}(id)\}, \{\}, \{\}, \{\}, \{\})$$
where $(I{:}\mathbf{class}(id)) \in tenv(K)$

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

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

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

[tc type declaration]
$$K \vdash \texttt{type}\ I \succ (\{I{:}\mathbf{namedtype}(tid)\}, \{\}, \{\}, \{\}, \{\})$$
where $(I{:}\mathbf{namedtype}(tid)) \in tenv(K)$

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

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

Figure C-29: Static semantics of declarations.

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

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

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

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

C.3.7 Static Semantics of Class Names

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

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

Figure C-30: Static semantics of class names.

[tc identifier attribute 1]

$$\frac{\begin{array}{c}\pi \vdash I \Rightarrow \mathbf{class}(id),\\ (\chi,\zeta) \vdash id <: \mathbf{exact}(ics,acs)\end{array}}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I \Rightarrow (ics,acs)}$$

[tc identifier attribute 2]

$$\frac{\begin{array}{c}\pi \vdash I \Rightarrow \mathbf{formal}(\tau),\\ \zeta \vdash \tau \leq_{sub} \mathbf{exact}(ics,acs)\end{array}}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I \Rightarrow (ics,acs)}$$

[tc method attribute]

$$\frac{\begin{array}{c}\pi \vdash CN_1 \Rightarrow_{class} id_1, \ldots, \pi \vdash CN_n \Rightarrow_{class} id_n,\\ \pi \vdash T_1 \Rightarrow \tau_1, \ldots, \pi \vdash T_n \Rightarrow \tau_n, \quad \pi \vdash T_r \Rightarrow \tau_r,\\ (\pi'',\iota,\phi,\chi,\zeta,\psi) \vdash B : \tau_r\end{array}}{\begin{array}{c}(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash \texttt{method (}I_1\texttt{@}CN_1\texttt{:}T_1,\ldots,I_n\texttt{@}CN_n\texttt{:}T_n\texttt{) :}T_r\ \texttt{\{}B\texttt{\}} \Rightarrow\\ (\{((id_1,\ldots,id_n),\mathbf{arrow}((\tau_1,\ldots,\tau_n),\tau_r))\},\{\})\end{array}}$$

where $n \geq 0$,
$\pi' =_s \{I_1{:}\mathbf{formal}(\tau_1)\} \cup \ldots \cup \{I_n{:}\mathbf{formal}(\tau_n)\}$,
$\pi'' = \pi \uplus \pi'$

[tc storage attribute]

$$\frac{\begin{array}{c}\pi \vdash CN_1 \Rightarrow_{class} id_1, \ldots, \pi \vdash CN_n \Rightarrow_{class} id_n,\\ \pi \vdash T_1 \Rightarrow \tau_1, \ldots, \pi \vdash T_n \Rightarrow \tau_n, \quad \pi \vdash T_r \Rightarrow \tau_r,\\ (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash E : \tau_r\end{array}}{\begin{array}{c}(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash \texttt{storage (}I_1\texttt{@}CN_1\texttt{:}T_1,\ldots,I_n\texttt{@}CN_n\texttt{:}T_n\texttt{) :=}E\texttt{:}T_r \Rightarrow\\ (\{((id_1,\ldots,id_n),\mathbf{arrow}((\tau_1,\ldots,\tau_n),\tau_r))\},\\ \{((id_1,\ldots,id_n),\mathbf{assignable}((\tau_1,\ldots,\tau_n),\tau_r))\})\end{array}}$$

where $n \geq 0$,
the $I_1,\ldots,I_n$ are distinct

[tc acceptor attribute]

$$\frac{\begin{array}{c}\pi \vdash CN_1 \Rightarrow_{class} id_1, \ldots, \pi \vdash CN_n \Rightarrow_{class} id_n,\\ \pi \vdash T_1 \Rightarrow \tau_1, \ldots, \pi \vdash T_n \Rightarrow \tau_n, \quad \pi \vdash T_r \Rightarrow \tau_r,\\ (\pi'',\iota,\phi,\chi,\zeta,\psi) \vdash B : \tau_r\end{array}}{\begin{array}{c}(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash \texttt{acceptor (}I_1\texttt{@}CN_1\texttt{:}T_1,\ldots,I_n\texttt{@}CN_n\texttt{:}T_n\texttt{) :=}\ I\texttt{:}T_r\\ \texttt{\{}B\texttt{\}} \Rightarrow (\{\},\{((id_1,\ldots,id_n),\mathbf{assignable}((\tau_1,\ldots,\tau_n),\tau_r))\})\end{array}}$$

where $n \geq 0$,
$\pi' =_s \{I_1{:}\mathbf{formal}(\tau_1)\} \cup \ldots \cup \{I_n{:}\mathbf{formal}(\tau_n)\} \cup \{I{:}\mathbf{formal}(\tau_r)\}$,
$\pi'' = \pi \uplus \pi'$

[tc combination attribute]

$$\frac{K \vdash GF_1 \Rightarrow (ics_1,acs_1), \quad K \vdash GF_2 \Rightarrow (ics_2,acs_2)}{K \vdash GF_1\ \texttt{\&}\ GF_2 \Rightarrow (ics,acs)}$$

where
$ics = ics_1 \uplus ics_2$
$acs = acs_1 \uplus acs_2$

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

C.3.8 Type Checking Generic Function Attributes

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

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

Figure C-34: Static semantics of type names.

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

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

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

C.3.9 Static Semantics of Type Expressions

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

C.3.10 Static Semantics of Type Names

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

C.3.11 Type Checking Expressions

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

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

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

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

[arrow type]
$$\frac{\pi \vdash T_1 \Rightarrow \tau_1, \ldots, \pi \vdash T_n \Rightarrow \tau_n, \quad \pi \vdash T_n \Rightarrow \tau_r}{\pi \vdash \mathtt{(}T_1,\ldots,T_n\mathtt{)} \texttt{->} T_r \Rightarrow \mathbf{arrow}((\tau_1,\ldots,\tau_n),\tau_r)} \qquad \text{where } n \geq 0$$

[assignable type]
$$\frac{\pi \vdash T_1 \Rightarrow \tau_1, \ldots, \pi \vdash T_n \Rightarrow \tau_n, \quad \pi \vdash T_n \Rightarrow \tau_r}{\pi \vdash \mathtt{(}T_1,\ldots,T_n\mathtt{)} \texttt{:=} T_r \Rightarrow \mathbf{assignable}((\tau_1,\ldots,\tau_n),\tau_r)} \qquad \text{where } n \geq 0$$

[and type]
$$\frac{\pi \vdash T_1 \Rightarrow \tau_1, \quad \pi \vdash T_2 \Rightarrow \tau_2}{\pi \vdash T_1 \;\mathtt{\&}\; T_2 \Rightarrow \mathbf{andtype}(\{\tau_1, \tau_2\})}$$

[or type]
$$\frac{\pi \vdash T_1 \Rightarrow \tau_1, \quad \pi \vdash T_2 \Rightarrow \tau_2}{\pi \vdash T_1 \;\mathtt{|}\; T_2 \Rightarrow \mathbf{ortype}(\{\tau_1, \tau_2\})}$$

[exact type]
$$\frac{\begin{array}{c}\pi \vdash ET_{f(1)} \Rightarrow (idv_1, \mathbf{arrow}(\tau v_1, \tau_1)),\\ \ldots,\\ \pi \vdash ET_{f(n)} \Rightarrow (idv_n, \mathbf{arrow}(\tau v_n, \tau_n)),\\ \pi \vdash ET_{f(n+1)} \Rightarrow (idv_{n+1}, \mathbf{assignable}(\tau v_{n+1}, \tau_{n+1})),\\ \ldots,\\ \pi \vdash ET_{f(n+k)} \Rightarrow (idv_{n+k}, \mathbf{assignable}(\tau v_{n+k}, \tau_{n+k}))\end{array}}{\begin{array}{c}\pi \vdash \texttt{exact\{}ET_1 \texttt{,} \ldots \texttt{,} ET_{n+k}\texttt{\}} \Rightarrow\\ \mathbf{exact}(\{(idv_1,\mathbf{arrow}(\tau v_1,\tau_1)),\ldots,(idv_n,\mathbf{arrow}(\tau v_n,\tau_n))\},\\ \{(idv_{n+1},\mathbf{assignable}(\tau v_{n+1},\tau_{n+1})),\\ \ldots,(idv_{n+k},\mathbf{assignable}(\tau v_{n+k},\tau_{n+k}))\})\end{array}}$$

where $n \geq 0$, $k \geq 0$, f is a bijection on $\{1,\ldots,n+k\}$, all of the ET_i are distinct.

Figure C-32: Static semantics of type expressions.

[exact arrow]
$$\frac{\begin{array}{c}\pi \vdash CN_1 \Rightarrow_{class} id_1, \ldots, \pi \vdash CN_n \Rightarrow_{class} id_n,\\ \pi \vdash T_1 \Rightarrow \tau_1, \ldots, \pi \vdash T_n \Rightarrow \tau_n, \quad \pi \vdash T_r \Rightarrow \tau_r\end{array}}{\pi \vdash \mathtt{(}CN_1{:}T_1\mathtt{,} \ldots \mathtt{,} CN_n{:}T_n\mathtt{)} \texttt{->} T_r \Rightarrow ((id_1,\ldots,id_n), \mathbf{arrow}((\tau_1,\ldots,\tau_n),\tau_r))} \qquad \text{where } n \geq 0$$

[storage type]
$$\frac{\begin{array}{c}\pi \vdash CN_1 \Rightarrow_{class} id_1, \ldots, \pi \vdash CN_n \Rightarrow_{class} id_n,\\ \pi \vdash T_1 \Rightarrow \tau_1, \ldots, \pi \vdash T_n \Rightarrow \tau_n, \quad \pi \vdash T_r \Rightarrow \tau_r\end{array}}{\pi \vdash \mathtt{(}CN_1{:}T_1\mathtt{,} \ldots \mathtt{,} CN_n{:}T_n\mathtt{)} \texttt{:=} T_r \Rightarrow ((id_1,\ldots,id_n), \mathbf{assignable}((\tau_1,\ldots,\tau_n),\tau_r))} \qquad \text{where } n \geq 0$$

Figure C-33: Static semantics of exact types.

a most specific applicable exact case. From the set of all these selected cases, the least upper bound of their result types is returned as the result type of the invocation.

The auxiliary function zip, used in the rules for applications and assignments with directed actuals, is defined below.

zip: $ObjectId^* \times Powerset(ObjectId)^* \rightarrow ActualArgPair^*$
$zip((id_1,\ldots,id_n),(ids_1,\ldots,ids_n)) = ((id_1,ids_1),\ldots,(id_n,ids_n))$

[tc identifier expression 1]

$$\frac{\pi \vdash I \Rightarrow \mathbf{class}(id)}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I : \tau}$$

where $(id,\tau) \in \chi$

[tc identifier expression 2]

$$\frac{\pi \vdash I \Rightarrow \mathbf{formal}(\tau)}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I : \tau}$$

[tc application expression 1]

$$\frac{\begin{array}{c} K \vdash E_0 : \mathbf{arrow}((\tau_1,\ldots,\tau_n),\tau_r), \\ K \vdash AA_1 \blacktriangleright (\tau_1, \{\}) \text{ , ..., } K \vdash AA_n \blacktriangleright (\tau_n, \{\}) \end{array}}{K \vdash E_0(AA_1,\ldots,AA_n) : \tau_r}$$

where $n \geq 0$

[tc application expression 2]

$$\frac{\begin{array}{c} (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I \Rightarrow_{class} id_0 \\ (\chi,\zeta) \vdash id_0 <: \mathbf{exact}(ics,acs), \\ (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash AA_1 \blacktriangleright (\tau_1, ids_1) \text{ ,} \\ \ldots, \\ (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash AA_n \blacktriangleright (\tau_n, ids_n), \\ (\iota_0, aav_1) \vdash ec_1 \textit{ is-best-for ics}, \\ \ldots, \\ (\iota_0, aav_m) \vdash ec_m \textit{ is-best-for ics}, \\ \zeta \vdash (\tau_1,\ldots,\tau_n) \leq_{sub} \tau v_1, \ldots, \zeta \vdash (\tau_1,\ldots,\tau_n) \leq_{sub} \tau v_m \end{array}}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I\ (AA_1,\ldots,AA_n) : \tau_r}$$

where $n \geq 0$, $m \geq 0$,
$\iota_0 = \phi(id_0)$
$\{idv_1,\ldots,idv_m\} =$
$\quad conformers(\pi,\chi,\zeta,(\tau_1,\ldots,\tau_n))$,
$(\forall i \,.\, i \in \{1,\ldots,m\} \Rightarrow$
$\quad ec_i \in ics$ and
$\quad \tau v_i = argtypes(ec_i)$ and
$\quad aav_i = zip(idv_i, (ids_1,\ldots,ids_n)))$,
$\tau_r = \mathbf{ortype}(\{restype(ec_1), \ldots,$
$\qquad restype(ec_m)\})$

[tc assignment expression 1]

$$\frac{\begin{array}{c} \kappa \vdash E_0 : \tau, \\ \zeta \vdash \tau \leq_{sub} \mathbf{assignable}((\tau_1,\ldots,\tau_n),\tau_{n+1}), \\ K \vdash AA_1 \blacktriangleright (\tau_1, \{\}) \text{ , ..., } \kappa \vdash AA_n \blacktriangleright (\tau_n, \{\}), \\ K \vdash E_{n+1} : \tau_{n+1} \end{array}}{K \vdash E_0(AA_1,\ldots,AA_n)\mathbf{:=}E_{n+1} : \tau}$$

where $n \geq 0$,
$\zeta = directSubtypes(\kappa)$

[tc assignment expression 2]

$$\frac{\begin{array}{c} (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I \Rightarrow_{class} id_0 \\ (\chi,\zeta) \vdash id_0 <: \mathbf{exact}(ics,acs), \\ (\chi,\zeta) \vdash id_0 <: \tau, \\ (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash AA_1 \blacktriangleright (\tau_1, ids_1) \text{ ,} \\ \ldots, \\ (\pi,\iota,\phi,\chi,\zeta,\psi) \vdash AA_n \blacktriangleright (\tau_n, ids_n), \\ (\pi,\iota,\phi,\xi,\zeta,\psi) \vdash E_{n+1} : \tau_{n+1} \\ (\iota_0, aav_1) \vdash ec_1 \textit{ is-best-for acs}, \\ \ldots, \\ (\iota_0, aav_m) \vdash ec_m \textit{ is-best-for acs}, \\ \zeta \vdash (\tau_1,\ldots,\tau_n) \leq_{sub} \tau v_1, \ldots, \zeta \vdash (\tau_1,\ldots,\tau_n) \leq_{sub} \tau v_m, \\ \zeta \vdash \tau_{n+1} \leq_{sub} \tau_{r,1}, \ldots, \zeta \vdash \tau_{n+1} \leq_{sub} \tau_{r,m} \end{array}}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash I\ (AA_1,\ldots,AA_n)\mathbf{:=}E_{n+1} : \tau}$$

where $n \geq 0$, $m \geq 0$,
$\iota_0 = \phi(id_0)$
$\{idv_1,\ldots,idv_m\} =$
$\quad conformers(\pi,\chi,\zeta,(\tau_1,\ldots,\tau_n))$,
$(\forall i \,.\, i \in \{1,\ldots,m\} \Rightarrow$
$\quad ec_i \in acs$ and
$\quad \tau v_i = argtypes(ec_i)$ and
$\quad \tau_{r,i} = restype(ec_i)$ and
$\quad aav_i = zip(idv_i, (ids_1,\ldots,ids_n)))$

[tc sequence expression]

$$\frac{K \vdash E_1 : \tau_1, \quad K \vdash E_2 : \tau_2}{K \vdash E_1 \mathbf{;} E_2 : \tau_2}$$

Figure C-35: Static semantics of expressions.

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

Figure C-36: Subsumption rule for expressions.

[tc identifier 1] $\pi \vdash I \Rightarrow \textbf{class}(id)$ where $(I{:}\textbf{class}(id)) \in \pi$

[tc identifier 2] $\pi \vdash I \Rightarrow \textbf{formal}(\tau)$ where $(I{:}\textbf{formal}(\tau)) \in \pi$

Figure C-37: Static semantics of identifiers

[tc undirected actual]
$$\frac{K \vdash E : \tau}{K \vdash E \blacktriangleright (\tau, \{\})}$$

[tc directed actual]
$$\frac{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash E : \tau, \quad \pi \vdash CN_1 \Rightarrow_{class} id_1, \ldots, \pi \vdash CN_n \Rightarrow_{class} id_n}{(\pi,\iota,\phi,\chi,\zeta,\psi) \vdash E@CN_1,\ldots,CN_n \blacktriangleright (\tau, \{id_1,\ldots,id_n\})} \quad \text{where } n \geq 0$$

Figure C-38: Static semantics of actual arguments.

For type checking an assignment, the process is similar to type checking an invocation. If none of the actuals are directed, then the first form may be used, in which the operator must have an assignable type. The actuals and the value being assigned must be the same as (or subtypes of) the argument and result types of this assignable type. If some of the actuals are directed, then the operator must again be a class name, and each selected case must be a storage table with appropriate types for arguments and the result.

A sequence expression has the type of the second expression in the sequence.

There is also a subsumption rule for expressions, given in Figure C-36.

C.3.12 Static Semantics of Identifiers

An identifier is looked up in the assumed type environment, and a type denotable is returned. The identifier must denote either a class or a formal. The axioms are given in Figure C-37.

C.3.13 Static Semantics and Type Checking of Actual Arguments

The static semantics of an actual argument in an assumed type context produces a typed actual. A typed actual is a pair of a type attribute and a set of class identities. The type checking rules are given in Figure C-38. For both kinds of actuals, the expression must type check, and its type attribute, τ, is returned as part of the pair. For a directed actual, each named class must be a class, not just a formal parameter with an exact type, so that its identity is known.

Appendix D Design Decisions and Alternatives for the BeCecil Type System

In this section we describe some alternatives in the design of the BeCecil type system, and the reasons for our decisions.

D.1 Separation of types and classes

A fundamental design decision in the type system is the separation of the notions of type and class, and thus of subtype and subclass. Although it would simplify the type system if it relied on the identification of types and classes, there are well-known limitations in languages that identify the notions [Snyder 86]. However, it might be reasonable to only "half" separate the two notions, by making types be classes, but introducing two notions of inheritance (public and private). For example, in C++, every subtype must be a subclass, but not all subclasses are subtypes. This compromise also has well-known limitations. However, from the point of view of a theoretical study, there are two different kinds of relationships in either case (subtyping and subclassing, or public and private inheritance), and it seems to be cleaner to use the distinction between subtyping and subclassing. We use the notion of conformance to bridge the gap between the two worlds of objects and types.

D.2 Structural vs. By-Name Subtyping and Conformance

Another fundamental distinction is that between structural and "by name" type checking. This decision subsumes the finer distinction between structural and by name subtyping. In a language that distinguishes types and classes, one can also distinguish between structural and by name conformance; that is, does an object conform to a type simply by virtue of supporting the appropriate methods, or is an explicit declaration of conformance needed? The advantage of structural notions are that they work well with separate typechecking and with independent development of code; the disadvantage is that there sometimes there may be superficial matches (subtype relationships, conformance relationships) where the semantics of the types involved would say that the types should not be treated as matching. Languages like Modula-3 [Nelson 91, Harbison 92] allow the programmer to have it both ways, using structural relationships by default and allowing the programmer to explicitly "brand" types to prevent superficial matches.

In BeCecil, a fundamental problem with structural notions of typing, subtyping, and conformance is that named types and objects have no structure. Inexact types, that is invocable types and assignable types, do have structure, and there are structural rules that apply to them. There are also structural subtyping rules for conjunctive and disjunctive types. However, a named type has no such structure. Similarly, except in its capacity as a generic function, an object itself has no structure on which to base structural conformance rules. In the absence of such explicit structure, structural rules would have to be based on declared interfaces (protocols) for types. Such declarations are available in BeCecil, and thus structural subtyping could be based on the declared types of generic functions that are applicable to the types in question. However, such a notion is less fixed than similar notions for ADT or single-dispatch OO languages, because there is no clear distinction between "primitive" and "client" operations that apply to a type. Thus adding a new generic function that applied to a given type could change many subtype relationships in a program, possibly in subtle ways. Similarly, one could decide whether an object conforms to a type on the basis of what generic functions apply to it, but again this could change if new generic functions were defined. Furthermore, type names, since they correspond to ADTs, seem more likely to want "brands" to prevent superficial matches, but we wanted to avoid the extra complexity of dealing with both structural and by name relationships for type names in our theoretical language. For these reasons, we have only investigated by name subtyping and conformance for type names. We leave the investigation of structural relationships, and ways to explicitly declare type interfaces as future work.

D.3 Dealing with Nested Scopes

The trickiest aspect of the type system is in dealing with nested scopes. Recall from the discussion in Section 3 that the problem of types acquiring new conformers can lead to incompleteness, and the problem of relating unrelated conformers can lead to inconsistency. These problems have two different origins in our type system. The problem of types acquiring new conformers is caused by our separation of types and classes (i.e., by our separation of the notions of subtyping and inheritance). If every subtype had to be implemented by a class that inherited from a class that implemented the supertype in question, then this problem would disappear. Indeed this would also be a problem in single dispatch languages that had block structure and separated subtyping and inheritance.* The problem of relating unrelated conformers is due to multiple dispatch and our refusal to linearize the type hierarchy. With multiple dispatch and without linearization, the possibility of inconsistency in handling applications arises.

It may be instructive to briefly consider some ideas that we rejected for making the type system sound in the face of these problems. An early idea was to recheck that each generic function of a surrounding contour was properly implemented, in the context of each nested contour; the idea was that if the new object and inheritance declarations in the nested recursive declaration sequence did not invalidate the assumptions used in type checking the outer contour, then when objects created in the inner contour were passed out to surrounding contours, they would not cause problems. This is similar to what is the current type system does, but it is less conservative, and also less constructive. It is less constructive in the sense that it does not tell the programmer what situations to avoid in as much detail as our present rule.

Another idea was to prevent the “escape” from nested contours of all objects that did not inherit from classes known in outer contours. This would be sound, but was too restrictive, as it prevented almost all uses of inheritance in nested contours. The current rule is less restrictive, because it only prevents the uses of inheritance in nested contours that may cause the two problems mentioned above.

It should be noted, however, that nested contours are still not very useful. One basic feature of the dynamic semantics is that from within a nested contour, one cannot extend the generic functions or inheritance relationships of the surrounding contour in a way that will directly affect the surrounding contour. Thus, most programs will make most of their declarations at top-level.

D.4 Dealing with Directed Actual Arguments

In type checking applications with directed actual arguments, we chose to require that the operator be statically known (a class name). The reason for this is that to do the type checking for directed actuals, the inheritance relation closed with the class is needed to mimic the dynamic semantics.

We briefly considered putting information about the inheritance relation of a generic function into some form of exact type; the problem is that the classes that might be related by such inheritance relationships would not necessarily be statically known, which could happen if the generic function was defined in a contour other than a surrounding contour. Thus it is hard to envision how to write down such inheritance relations involving unknown class names; even class names that look as if they referred to classes in the surrounding contour might not be intended to do so. In any case the syntax would be messy and there seemed no great need for the capability of using directed actuals with unknown generic functions. Indeed it is somewhat against the spirit of OO programming to be able to invoke a particular method in an unknown generic function; it would be like extracting and calling a particular method of an unknown object in a single-dispatch language.

* Beta [Madsen et al. 93] is a block-structured OO language, but does not separate types and class.

D.5 Inference vs. Declaration for Generic Function Types

The typing rules for BeCecil could either infer the generic function types of objects or require them to be declared. The present type system infers the exact types of objects (i.e., the types of all generic function attributes). Thus programmers are not required to declare generic function types for objects, as they can always be inferred. However, programmers can still declare that generic functions must conform to a given type, and such declarations are checked against the declared exact types during implementation-side type checking. This has an advantage, in that one can specify that all objects that conform to a given type must implement a particular operation.

There is a small disadvantage to inferring inexact types from exact types, however. It is that programmers have to be careful to not make the result types of methods too specific, relative to their declared argument types, as described in Section 3.2.2. Language changes that would prevent such problems would be to require inexact type declarations instead of inferring them, or to allow the programmer to have some way to turn off the inference for particular methods. This second solution would complicate the language in a way that does not seem orthogonal, and so was rejected. The first solution seems workable, but would be farther away from a practical language, because most generic function type declarations would be redundant with generic function attributes. We leave investigation of this solution as an alternative for future work.

D.6 Object Roles (abstract vs. concrete objects)

In early versions of BeCecil (and in our previous work [Chambers & Leavens 94, Chambers & Leavens 95]), we distinguished two *roles* for objects. An *abstract* object could not be used in an expression, while a concrete object could be used in an expression. This distinction mimicked the distinction between abstract and concrete classes in languages such as Eiffel [Meyer 88, Meyer 92], C++, Dylan, and many others.

In a language where classes are types and subclasses generate instances that are subtypes, such a distinction is necessary. The reason is so that the programmer does not have to implement all methods that are specialized on abstract classes, since such classes cannot be directly used to create instances. A method that is not implemented in an abstract class, and which must be overridden in all subclasses is called a "pure virtual method" in C++, and a "deferred method" in Eiffel. Examples of such methods include the `if` method in the classic coding of the Booleans with an abstract class `Boolean` and two subclasses, `True` and `False` [Goldberg & Robson 83]. Forcing the programmer to implement such methods in the abstract class would cause programmers to write methods that looped or gave errors when called; but such code is completely worthless. At best such code will never be called, and at worst a run-time error will occur, essentially reducing what should be a static error to a dynamic error. It is better to follow languages like C++, where one can declare that an abstract class has some pure virtual methods, and thus inform the type system that the method does not have to be implemented in that class.

However, a distinction between abstract and concrete objects turned out to be an unnecessary complication in BeCecil, because in BeCecil objects and types are completely separated. To declare that an object is an abstract class, one simply does not declare that it conforms to any type. Such an object cannot be used as an object in an expression (although it can be used as a generic function), and therefore it is abstract. This also simplifies the type system, which no longer has to deal with abstract objects that conform to various types.

D.7 Should Specializers Conform to Declared Argument Types?

It might seem natural to require that specializers of formal arguments should conform to their declared argument types. For example, in the following method, the first argument conforms in this way, but the second does not, because `GenericFun_obj` does not conform to `()->Top`.

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

Methods that use the specializer `any` would also not conform to their declared argument type. Such examples are common, and the use of `any` in particular is a pattern that makes an argument effectively "unspecialized" (if every class is assumed to inherit from `any`, as in Cecil).

The above discussion shows, however, only that it is a considerable programming convenience to permit specializers to not conform to their declared argument types. One could, less conveniently, declare a less restrictive type, or a more restrictive specializer class. In a language in which classes and types were identified (and subtyping is identified with subclassing), such inconveniences would seem more natural.

However, in BeCecil, there is also another reason for permitting specializers that do not conform to their declared argument types. The reason is that if one programs using an ADT pattern, then one will hide object names but make conformance relationships public. (See Section 2.6 for an example.) Doing this would prevent clients from inheriting certain operations, but that is a property of ADT languages (such as Ada 83 and CLU) as well. If one did that, then it would be necessary for client functions to use a specializer other than the class name, because the class name would be hidden. So to support this ADT pattern of completely hiding an implementation, and for reasons of convenience, the specializers of formal arguments need not conform to their declared types.

Appendix E Notations for Finite Functions, Stores, Environments, and Relations

E.1 Finite Functions

Finite functions are widely used in semantics to model environments and stores. In this section we describe notation for finite functions and finite, binary relations. There is an embedding of finite functions into relations, which we exploit to more easily describe finite functions. This embedding is just the traditional mathematical view that a function, f, is the same as its graph, the relation $\{(x, f(x)) \mid x \in dom(f)\}$.

We first describe finite functions and how to extract their graphs.

$f \;\in\; FiniteFun[A,B] \qquad = A \to B_\perp$
$dom\text{: } FiniteFun[A,B] \to Powerset(A)$
$dom(f) = \{a \mid f(a) \neq \perp\}$
$graph\text{: } FiniteFun[A,B] \to FiniteRel[A,B]$
$graph(f) = \{(a, b) \mid a \in dom(f), f(a) = b\}$

Finite relations are modeled as sets of ordered pairs. We consider the empty set, {}, singleton sets, and set union as generators for sets when writing inductive definitions over sets.

$R \;\in\; FiniteRel[A,B] \qquad = Powerset(A \times B)$
$dom\text{: } FiniteRel[A,B] \to Powerset(A)$
$dom(R) = \{a \mid (a, b) \in R\}$
$maps\text{: } FiniteRel[A,B] \times A \to Powerset(B)$
$maps(R, a) = \{b \mid (a, b) \in R\}$
$isFunction\text{: } FiniteRel[A,B] \to Boolean$
$isFunction(R) = (\forall a \,.\, a \in dom(R) \Rightarrow |maps(R, a)| = 1)$
$apply\text{: } FiniteRel[A,B] \times A \to B_\perp$
$apply(R, a) = \mathbf{if}\ isFunction(R) \wedge a \in dom(R)\ \mathbf{then}\ (\mathbf{let}\ \{b\} = maps(R, a)\ \mathbf{in}\ b)\ \mathbf{else}\ \perp$
$toFun\text{: } FiniteRel[A,B] \to FiniteFun[A,B]$
$toFun(R) = \lambda a \,.\, apply(R, a)$

It is well known that $toFun(graph(f)) = f$, and that if $isFunction(R)$ holds, then $graph(toFun(R)) = R$. Thus throughout the rest of this paper, we freely use this dual point of view, writing definitions from whichever point of view is suitable, omitting the various uses of *graph* and *toFun*.

Some of the following notation is from David A. Schmidt's book *The Structure of Typed Programming Languages* (MIT Press, 1994).

$\cdot[\cdot := \cdot]\text{: } FiniteFun[A,B] \times A \times B \to FiniteFun[A,B]$
$f[a_1 := b] = \lambda a_2 \,.\, \mathbf{if}\ a_2 = a_1\ \mathbf{then}\ b\ \mathbf{else}\ f(a_2)$
$disjoint\text{: } FiniteRel[A,B] \times FiniteRel[A,B] \to Boolean$
$disjoint(f_1, f_2) = ((dom(f_1) \cap dom(f_2)) = \{\})$
$\cdot \mathbin{\dot\cup} \cdot\text{: } FiniteFun[A,B] \times FiniteFun[A,B] \to FiniteFun[A,B]_\perp$
$f_1 \mathbin{\dot\cup} f_2 = \mathbf{if}\ disjoint(f_1, f_2)\ \mathbf{then}\ f_1 \cup f_2\ \mathbf{else}\ \perp$
$\cdot \uplus \cdot\text{: } FiniteFun[A,B] \times FiniteFun[A,B] \to FiniteFun[A,B]$
$f_1 \uplus f_2 = \lambda a \,.\, \mathbf{if}\ a \in dom(f_2)\ \mathbf{then}\ f_2(a)\ \mathbf{else}\ f_1(a)$
$bind\text{: } A^* \times B^* \to FiniteFun[A,B]_\perp$
$bind((a_1,\ldots,a_n), (b_1,\ldots,b_n)) = \{(a_1,b_1)\} \mathbin{\dot\cup} \ldots \mathbin{\dot\cup} \{(a_n,b_n)\}$

Following Schmidt, we also use the notation *a:b* for the ordered pair (a, b). Such a pair is called a *binding*. Thus we have the following abbreviations.

$(a{:}b) \in f \quad \equiv (a, b) \in f$
$\{a_1{:}b_1,\ldots,a_n{:}b_n\} \quad \equiv \{(a_1,b_1),\ldots,(a_n,b_n)\}.$

E.2 Stores

A *store* maps locations to stored values.

$\sigma \in Store = FiniteFun[Location, StoredValue]$
$l \in Location = Nat$

$emptystore: Store$
$emptystore = \{\}$

$allocate : Store \times StoredValue \rightarrow Location \times Store$
$allocate(\sigma, v) = (l, \sigma[l := v])$, **where** $l \notin dom(\sigma)$

$lookup: Store \times Location \rightarrow StoredValue_{\perp}$

$lookup(\sigma, l) = \sigma(l)$

E.3 Environments

An *environment* maps identifiers to denotable values.

$\eta \in Environment = FiniteFun[Identifier, DenotableValue]$

$emptyenv: Environment$
$emptyenv = \{\}$

$applyenv: Environment \times Identifier \rightarrow DenotableValue_{\perp}$

$applyenv(\eta, I) = \eta(I)$

E.4 Relations

The following additional operations on the type $FiniteRel[A,A]$ are also useful.

$\cdot^{+} : FiniteRel[A,A] \rightarrow FiniteRel[A,A]$

$R^{+} = R \cup \{(a_1, a_2) \mid a_1\ R\ b$ and $b\ R^{+}\ a_2$, for some $b \in A\}$

$\cdot{*} : FiniteRel[A,A] \rightarrow FiniteRel[A,A]$

$R{*} = R^{+} \cup \{(a, a) \mid a \in dom(R))\}$

$cyclic: FiniteRel[A,A] \rightarrow Boolean$

$cyclic(R) = (\exists a\ .\ a \in dom(R) \wedge a\ R^{+}\ a)$

$\cdot \overset{\otimes}{\cup} \cdot : FiniteRel[A,A] \times FiniteRel[A,A] \rightarrow FiniteRel[A,A]_{\perp}$

$R_1 \overset{\otimes}{\cup} R_2 =$ **if** $cyclic(R_1 \cup R_2)$ **then** $\perp$ **else** $R_1 \cup R_2$