Delft University of Technology
Software Engineering Research Group
Technical Report Series

The Semantics of Name Resolution in Grace

Vlad Vergu, Michiel Haisma, Eelco Visser

Report TUD-SERG-2017-011

TUD-SERG-2017-011

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

The Semantics of Name Resolution in Grace

Vlad Vergu
TU Delft
The Netherlands
v.a.vergu@tudelft.nl

Michiel Haisma
TU Delft
The Netherlands
m.a.haisma@student.tudelft.nl

Eelco Visser
TU Delft
The Netherlands
visser@acm.org

Abstract

Grace is a dynamic object oriented programming language designed to aid programming education. We present a formal model of and give an operational semantics for its object model and name resolution algorithm. Our main contributions are a systematic model of Grace's name resolution using scope graphs, relating linguistic features to other languages, and an operationalization of this model in the form of an operational semantics which is readable and executable. The semantics are extensively tested against a reference Grace implementation.

CCS Concepts •**Software and its engineering → Classes and objects; Semantics;**

Keywords object orientation, name resolution, dynamic semantics

1 Introduction

Grace is a dynamic object-oriented programming language designed to aid learning the art of programming. It is designed to be a small and simple programming language to learn. Its designers and maintainers are experienced researchers and educators in the field of programming languages. Grace embodies findings from decades of research in the field, drawing inspiration from reference languages such as Smalltalk [3], Self [52], Newspeak [11] and Java [28]. At its core Grace is lean: it consists of nested object literal expressions with multiple inheritance and anonymous functions. Other language features (e.g. classes, traits and modules) are defined in terms of these concepts [4, 32, 33]. There are two mainstream implementations: a Grace to C compiler and a Grace to JavaScript transpiler. The latter is available as a web-based development environment.

Various implementations and documentations of Grace exist, each implementing and documenting slightly different semantics. Lengthy discussions around the language's object model and name resolution are common within the design team. We posit (and our conversations with the Grace design team support this basis) that the fuel for discussion is the effect on name resolution of combining nested object expressions, multiple inheritance using traits, overriding and shadowing. The use of arbitrary expressions as ancestor objects (in the style of Newspeak) further hinders understanding. It becomes difficult to explain intended behavior to people both within and outside of the project.

The Grace community puts significant effort in creating and maintaining the Grace documentation which comes in two forms: an interactive tutorial and a language specification. Both are in prose with concrete code examples. The intended audience of the documentation is the user of the language. But a common feature of prosaic documentation is that it cannot afford the verbosity to describe all special and interesting cases of the language. Details of object construction and name resolution take a back seat in favor of explanations of how the language can be used. There is also no formal definition of the core linguistic features. Languages that inspired the design of Grace also either (1) lack formalizations themselves, (2) are conceptually distant from Grace, or (3) are statically typed: (1) Self and Newspeak have prose specifications [11, 52], (2) Smalltak-80 has an operational semantics [59] but Smalltalk lacks nested objects, and (3) Java's semantics [48] in K [7] defines static name resolution. As a consequence it is hard to fully grasp the underlying concepts in Grace.

In this paper we propose that a concise definition of Grace's object model and name resolution algorithm resolves this problem. The definition serves as readable documentation and as executable specification that can be used for experimental validation and as a reference implementation.

Our approach is to model Grace's name resolution using scope graphs [44, 53] and to operationalize this model as a specification for Grace in the Spoofax language workbench [36, 56].

Figure 1 shows the architecture of our implementation. High-level Grace features are desugared to lower level concepts using source-to-source transformations. We use scope graph notation to model the key aspects of name resolution in desugared programs. An operational semantics in DynSem [55], a domain-specific language for dynamic semantics specifications, serves as a concise and executable definition for object construction and name resolution semantics.

The contributions of this paper are:

- We model run-time name resolution using the scope graph paradigm.
- We give a concise and executable definition of Grace's object model and name resolution semantics in the DynSem dynamic semantics specification language.
- We separate name resolution from naming and confidential access policies. Policies are configurable by the language designer.

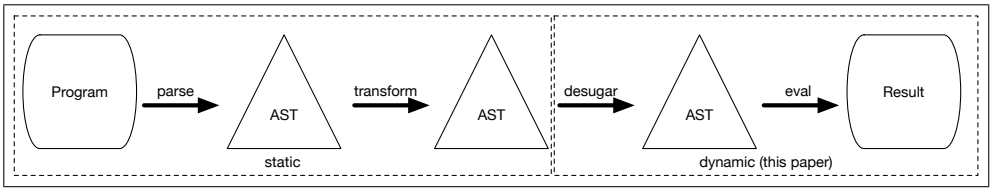


Figure 1. Architecture of the Grace language artifact.

- We have validated our specification through extensive testing against a reference implementation of Grace.

Outline The remainder of this paper is structured as follows. We begin with an overview of desugaring source-to-source transformation in Section 2. In Section 3 we model Grace name resolution using in the scope graph paradigm and give a systematic account of key aspects of name resolution. Section 4 defines the operational semantics of the object model, the name resolution algorithm and enforcement of policies. We evaluate our approach in Section 5, discuss related work in Section 6 and conclude with Section 7.

2 Desugaring

We desugar high-level Grace features of in terms of lower level concepts using a transformation implemented in Stratego [13]. The transformation is local (does not require global knowledge of the program) and is performed statically. Desugaring reduces the feature set which we must formally define. It applies the following transformations which are relevant to the object model and name resolution: (1) explicate the implicit enclosing object, (2) rewrite classes and traits as factory methods and (3) canonicalize method names.

(1) All Grace programs live in an implicit object — the module object. Desugaring rewrites a program `P` to `object{P}`.

(2) Class and trait declarations are rewritten as factory methods. Factory methods are regular methods which contain an object expression in their bodies. For example, the class declaration of Figure 2a desugars to the factory method of Figure 2b. Trait declarations are treated similarly. Classes and traits are always public, hence the `public` annotation of the factory methods.

(3) Method names are canonicalized such that all name parts are concatenated. The canonical method name encodes the arity of the method. For example, the method and call of Figure 2c desugar to the method and call of Figure 2d. The canonicalized method name is `if(_)then(_)`. The number of `_` symbols encodes the arity.

Throughout the remainder of the paper we assume that programs have been desugared as described above.

3 Name Resolution

Much of Grace's semantics revolves around name resolution, a concern which is strongly related to object orientation. Grace has lexical scoping of declarations and allows arbitrarily deep nesting of object expressions. Expressions may

```
class A {
  method m {...}
}
```
(a)

```
method A is public {
  object {
    method m {...}
  }
}
```
(b)

```
method
  if(e) then(b1){
  ...
}
if(1==1) then {
  ...
}
```
(c)

```
method if(_)then(_)(e, b1) {
  ...
}
if(_)then(_)(1==1, {...})
```
(d)

Figure 2. (a) class declaration before desugaring to (b) factory method. Multi-part method before (c) and after (d) name canonicalization.

explicitly refer to lexically surrounding objects and objects may have multiple ancestors. The use of expressions to determine ancestors means that meaningful name resolution can only be performed at run time. Method aliasing and exclusion combined with shadowing and overriding policies complicates name resolution. When lexical nesting and inheritance combine, name resolution becomes a complex comb-like search [10]. Some of the design decisions taken to aid learning Grace introduce additional name resolution concerns.

In this section we discuss the key aspects of Grace's name resolution by means of scope graphs [44, 53]. A scope graph is the result of distilling the abstract syntax tree of a program to information about names and scoping in the program. A scope graph is a directed graph consisting of the following ingredients. A scope represents a region in a program that behaves uniformly with respect to name binding. In scope graph diagrams, scopes are represented by circles. A declaration is the introduction of a name in a program. In diagrams, declarations are represented by a box with an incoming edge from the scope they are declared in. A reference is a use of a name in a program. In diagrams, references are represented by boxes with an outgoing edge to the scope in which they reside. Edges between scopes determine visibility inclusion. Name resolution consists of finding a path from each reference to a declaration with the same name, following the edges in the graph. As originally introduced, scope graphs represent purely static information about a program. However, in a dynamic language such as Grace, the scope graph partially emerges at run time. In diagrams we represent such dynamically constructed connections using red edges.

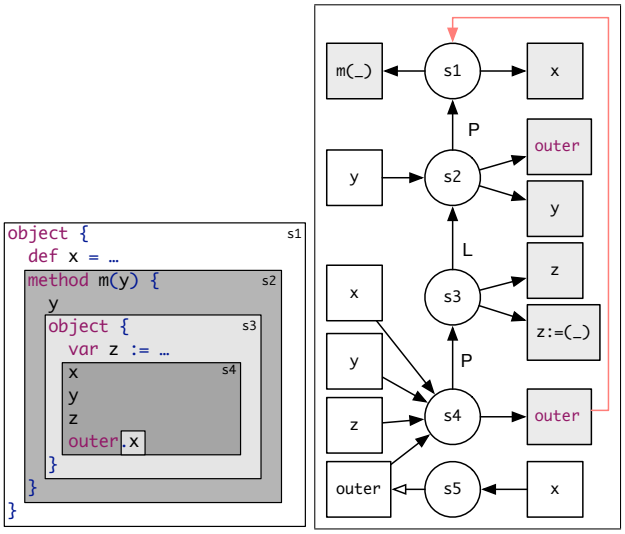


Figure 3. Program illustrating nested scopes and its scope graph.

As an introductory example, consider the scope graph and program of Figure 3. It identifies five scopes, of which scopes `s1`, `s2`, `s3` and `s4` are in a lexical structure. Scope `s1` corresponds to the top-level object. Scope `s1` has two declarations: one for field `x` and one for method `m(_)`. Method `m(_)`) has its own scope named `s2`. The `P` edge from `s2` to `s1` corresponds to the nesting of method `m(_)`) in the top-level object. The declaration of `y` in scope `s2` corresponds to parameter `y` of method `m(_)`. Reference `y` in scope `s2` refers to parameter `y`. Scope `s3` is the scope of the object created by method `m(_)`. The `L` edge from `s4` to `s3` corresponds to the nesting of the object expression in the method scope.

Name resolution comes down to two tasks: maintaining the scope graph for the program and calculating paths in the scope graph. In a statically typed language much of these tasks can be performed statically [47]. In a dynamic language such as Grace the two tasks are interleaved at run time.

We systematically describe Grace's name resolution in terms of scope graphs. Details of the intuition behind Grace's name resolution be found in the extended version [54].

Initialization statements make a constructor. Statements directly within the body of an object which are not declarations are initialization statements. Figure 3 illustrates an object with scope `s3`. The initialization expression of field `z` and the expressions below are initialization statements which reside in an implicit constructor method scope `s4`. The additional method scope `s4` simplifies reasoning about initialization statements and allows us to model all statements in an object uniformly.

Names are lexically scoped. Grace declarations are lexically scoped. Blocks (delimited by `{}`) scope declarations within. Lexical scoping implies that the declarations in a scope are only reachable from the scope itself or from scopes with a path to the declaration scope.

A reference is in lexical range of its declaration if there is a resolution path in the scope graph from the reference scope to the declaration scope and this resolution path contains only `P` and `L` edges. For example, in Figure 3, declarations in scope `s2` of method `m(_)` are only reachable from within `s2` and from nested scopes `s3` and `s4` which directly and indirectly import `s2`.

`Outer` *identifies surrounding object.* A distinctive feature of Grace is the `outer` pseudo-variable: the `outer` of an object o_2 is o_1 if the declaration of o_2 is enclosed by the declaration of o_1. The `outer` of the top-level object is undefined. (An exception are programs with *dialects*, which are outside of the scope of this paper).

The `outer` pseudo-variable can be used to qualify references to members in surrounding objects. Enclosing objects can be peeled off with successive `outer` references, e.g. `outer.outer.outer.x`. Consider the qualified reference `outer.x` in the implicit constructor scope `s4` of Figure 3. The qualified reference to `x` is a reference in anonymous scope `s5`, which imports `outer`. Reference `outer` in `s4` resolves to a declaration in same scope which is dynamically bound to the object scope `s1`.

We call `outer` a *pseudo*-variable because it lacks an explicit declaration and because it is constant. Instead, every method scope implicitly declares an `outer`, such that at run time the lexically surrounding object is reachable even when the method executes in derived objects.

The intention behind the `outer` feature is to give the programmer explicit resolution control when relying on nested objects. This can become useful when members are shadowed by declarations in nested objects. A similar feature in Java allows references qualified by the name of the surrounding class but the syntax is more involved. Self and JavaScript do not support explicit referencing of surrounding objects.

Fields are slots with getters and setters. As Figure 3 shows, fields and method declaration live in the same namespace. The declaration of field `x` in the object with scope `s1` induces a declaration for a method `x` which reads the value of the slot in the object corresponding to the field. Mutable fields, such as `z` in the object with scope `s3`, also induce a declaration for a setter method (e.g. `z:=(_)`) which writes the value of the parameter into the slot for the field.

Common namespaces for fields and methods allow programmers to replace fields with methods and vice versa without having to modify other parts of the program. A use pattern of this feature is to override getters and setters of an inherited field with methods that perform more complex computations. From a linguistic perspective the distinction between slots and named members conceptually separates an object's logic from the object's data.

Treatment of fields and methods in Grace is identical to that of Newspeak, and strongly dissimilar to that of Java where fields and methods reside in different namespaces. JavaScript fields and methods also share a namespace but fields are not slots with getters and setters, instead the object has named attributes and methods are function values assigned to named attributes.

Ancestor is determined dynamically. Grace objects can inherit from other objects, an unsurprising feature for an object oriented language. What sets Grace aside from many other languages is that an ancestor object is determined by evaluating an inheritance expression. The expression can perform arbitrary computation as long as it evaluates to a fresh object.

Consider the scope graph of Figure 4. The inheritance expression `t2` lives in scope `s7` the inheriting object scope. The value of the inheritance expression identifies the scope of the ancestor — `s5` from within method `t2` — and induces an import edge — `I(1)`.

The target of this import edge can only be computed after evaluation of the inheritance expression, thus the edge itself can only be computed at run time, at object construction time. (This is illustrated in scope graphs by colored edges.) We discuss the auxiliary scopes — namely `s7i1a` and `s7i1b` in the context of method aliasing and exclusion.

Method resolution entails finding a path in the scope graph from the scope of the youngest descendant object to a declaring scope. For example, consider resolving the reference `a` in scope `s8` of the constructor method from the object with scope `s7` in Figure 4. The scope graph yields the path `[P,I(1),I,I]` which traverses from the method scope to the object scope and to inherited scope via the auxiliary import scopes. The presence of an indexed `I` edge in the path indicates the fact that declaration is inherited.

The designers of Grace chose to support arbitrary inheritance expressions with the intuition that having such a feature increases the expressivity of the language. While it can be desired to have control flow determine the inheritance hierarchy of an object, it complicates name resolution. The complication is that static name analysis becomes dependent on intra- and inter-procedural data-flow analysis.

Identifying inheritance through arbitrary expression is similar to Newspeak. In Newspeak the expression is restricted, but only for parsing reasons [11]. Linguistically, the feature is similar to assigning the result of an arbitrary computation as the prototype of a JavaScript object. The fundamental difference with Self/JavaScript is that the ancestor of a Grace object cannot be changed after the object's creation. In Java ancestor classes can be identified by name only, and classes are resolved in a distinct class namespace.

Objects have multiple ancestors. Objects in Grace can inherit from multiple traits. Traits are just objects without state. Consider the object with scope `s7` of Figure 4 which inherits as a trait the object scope `s3` identified by `t1`. Firstly, the scope graph shows that the inheritance expression for the trait is evaluated in the object scope `s7`. This is a conscious decision by the Grace designers to allow the programmer to inherit form a trait provided by an ancestor class. For example, it is valid to inherit a trait which is inherited over the `I(1)` edge of the object scope. Import edges for traits induce additional `I` edges. `I` edges are indexed so that a path uniquely identifies a resolution.

Descendants may `alias` ***and*** `exclude` ***methods.*** The programmer may choose to alias and exclude methods when inheriting from objects and traits. The scope graph of Figure 5 illustrates how method aliasing and exclusion affects name resolution. For example, the alias `a = x` introduces the declaration for `a` as an alias to the inherited method `x`, in an auxiliary alias scope `s8i1b`. Exclusions introduce declaration filters in a separate auxiliary scope, `s8i1a`, directly importing from the alias scope.

The scope graph shows that a programmer may introduce an alias which is immediately excluded in the same inherit clause. Such a pattern is valid Grace although it would be of little use in practice. As long as there is a path from the reference part of the alias or from the reference part of an exclusion, to a declaration in an imported scope, it is valid to alias or exclude that method, respectively. Resolving an alias to the actual method declaration takes place in two steps. Firstly, the alias reference, e.g. `a` in scope `s9` of the constructor method, is resolved to a declaration in the auxiliary scope `s8i1b` via path `[P,I(1),I]`. Secondly, having found an alias declaration, a new resolution is performed for reference `x` starting from the alias scope `s8i1b`, yielding path `[I]` to scope `s3`.

If resolution of a reference, say `y` in scope `s9`, reaches a filter declaration for that name, for example `y` from auxiliary scope `s8i1a`, that resolution path is abandoned and resolution must backtrack.

The ability to inherit a method under a different name has two main uses: to preserve access to an inherited method if another ancestor would override it; and to avoid ambiguities when a lexically bound declaration has the same name as an inherited method. If inheritance is seen as an implementation method, then excluding a method from the interface of an object can be useful [50]. For example, if a particular operation is not supported by the composition of two interfaces, that feature can be excluded from the final object. Programmers be warned that excluding methods in descendants violates the Liskov Substitution Principle. One will eventually notice that an excluded method is not just excluded from the object's interface but is actually absent from the final object. Invoking any method (inherited or locally defined) that calls an excluded method will raise a run time error when the call is encountered.

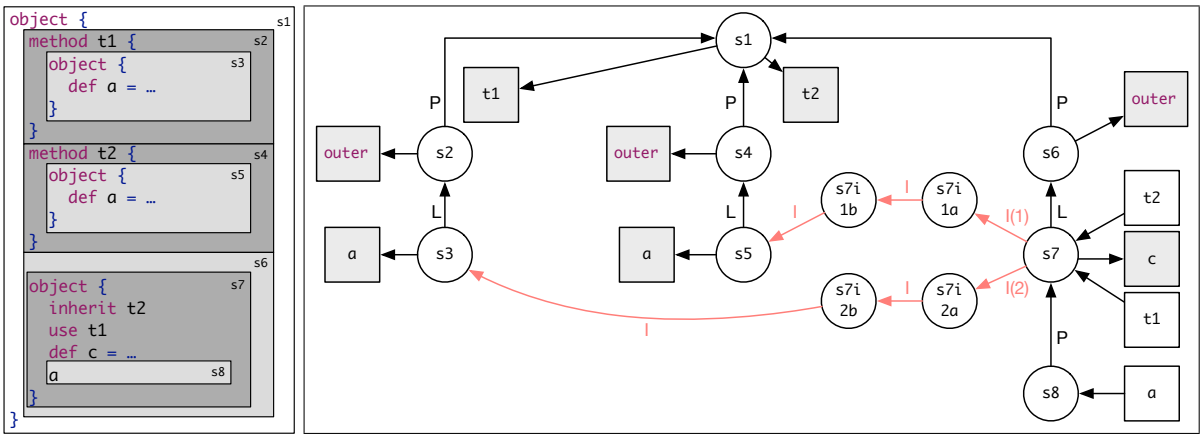


Figure 4. Scope graph (right) for a program with objects having multiple inheritance (left).

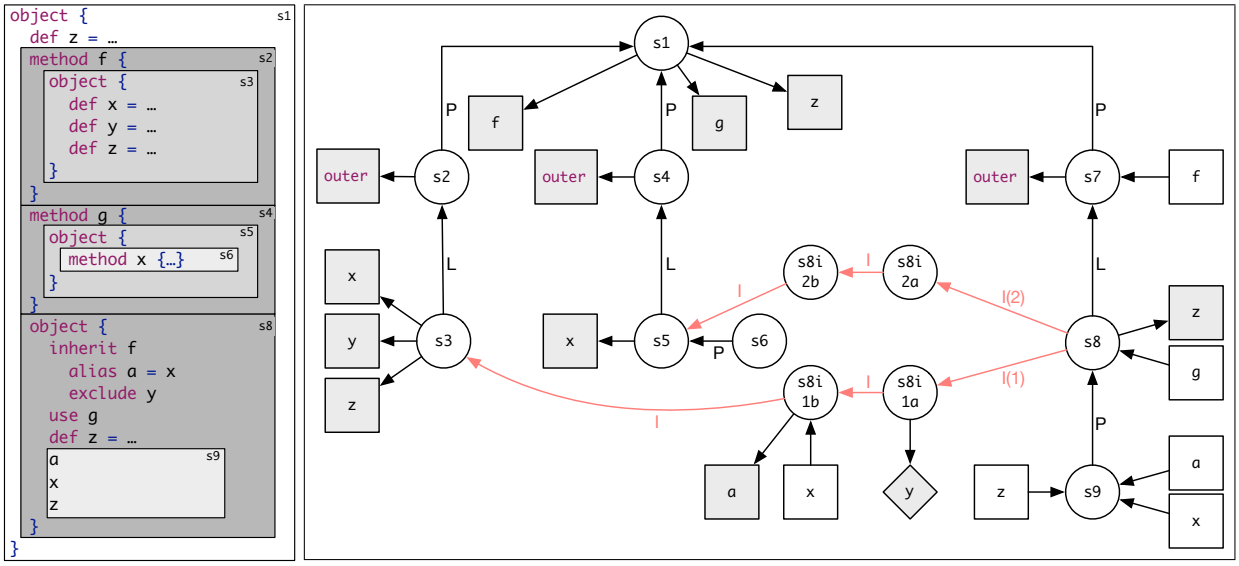


Figure 5. Multiple inheritance with aliases and excludes, shadowing and overriding (left) and corresponding scope graph (right).

Lexical scope may not be imported. Objects may not be used as proxies for their lexical scopes. It as design decision that maintains encapsulation and ensures privacy of an object's internal details. The context which constructed an object should at most be relevant to the features of the object itself, not the user of those features. This restriction has a natural parallel in the scope graph model: a path from a reference scope to a declaration scope may not contain `P` or `L` edges after `I` edges. We can rephrase this path restriction more intuitively: only `I` edges may follow `I` edges. This ensures that the receiver of qualified calls and that inherited objects may not be asked to resolve declarations in their lexical scopes.

Methods and local variables have different names. Grace enforces two restrictions on local variables: (1) two local variables in lexical range may not have the same name and (2) a local variable may not have the same name as a method in lexical range.

We enforce both restrictions at run time by examining resolution paths. Prior to recording a declaration for `x` in a method scope `s`, we resolve a fictive reference `x` from scope `s`. Note that if a path exists, its first edge must be `P`. If a path exists and it contains no `I` edges then the new declaration for `x` would be in lexical range of another declaration. We can further distinguish between restriction (1) and (2) by examining the last path edge: `L` in case (1) and `P` in case (2).

We find that discouraging multiple locals with the same name is good practice regardless of a language's policy. However we cannot determine a reason why locals and members should not be able to share the same name.

Ambiguous references are illegal. A reference may have two resolution paths in the scope graph, one strictly lexical and one via an import edge. Such references are illegal in Grace and raise run-time exceptions. The designers of Grace deemed that ambiguous references introduce unnecessary opacity in programs and that rejecting these should be a feature of the language. It is expected that programmers rename the inherited method using an alias, rename the declaration in a lexical scope, or qualify the reference to the lexically bound declaration using one or more `outer` qualifications.

Latest overriding method wins. Resolving a reference may yield multiple resolution paths. For example, resolving `x` from scope `s9` of the constructor method in Figure 5 yields two paths: `[P,I(1),I,I]` to `s3` and `[P,I(2),I,I]` to `s5`; two declarations are inherited from two separate scopes. The Grace-specific disambiguation policy for this situation is to choose the path containing the import edge with the highest index. This translates to a method overriding semantics where a later inherited declaration overrides earlier ones.

If a reference has both local and import path candidates, the local path is always preferred. For example, the reference `z` in `s9` has two candidate paths: `[P]` and `[P,I(1),I,I]`. The former path is preferred over the latter path. This translates to an intuitive semantics where a local definition of a method overrides all of the imported declarations for that name.

Members shadow `outer`'s members. A reference may have multiple resolution paths, all lexical. Reference `z` in constructor method scope `s9` of Figure 5 has two potential resolution paths, both lexical: `[P]` and `[P,L,P]`. The disambiguation policy is to prioritize the shortest path. This translates to a shadowing semantics: member declarations shadow declarations in outer object scopes.

Confidentiality requires name resolution. We conclude this section by claiming that confidentiality of an object's members is not part of name binding. Whether or not a reference to a particular member declaration is allowed from outside of the object, a resolution path will exist for that reference. More so, deciding whether access should be granted or rejected requires information about the reference and declaration scopes, the latter being the result of name resolution.

Suppose for instance that a qualified reference `o.x` from some scope s_{ref} resolves to a confidential declaration in some object scope s_{obj}. Access should be granted if $s_{ref} = s_{obj}$. If $s_{ref} \neq s_{obj}$, access should only be granted if the path from s_{ref} to s_{obj} has a `P` edge, i.e. that s_{ref} reaches s_{obj} through a lexically enclosing scope.

4 Operational Semantics

The dynamic semantics of a programming language defines the run-time behavior of its constructs. In this section we give an operational semantics for Grace's object model, name resolution algorithm, lookup, and enforcement of naming policies.

We use DynSem [55] as a specification language for the semantics. DynSem is a domain-specific language for specifying the dynamic semantics of programming languages. Specifications are given in terms of syntax-oriented rules over named arrows from program terms to values. Rules can access contextual evaluation information from read-only components (mentioned left of the $\vdash$ symbol) and from read-write components (mentioned right of the `::` symbol). A rule can omit semantic components which it does not use; a feature borrowed from I-MSOS [42]. Omitted components are implicitly propagated. Read-only components propagate downwards (environment semantics), read-write components thread through the rules (store semantics). Implicit propagation of components makes for more concise and modular specifications. DynSem statically type checks rules with respect to arrow, component and term signatures. Invalid term construction and impossible pattern matching are detected statically. A DynSem specification derives an interpreter for the object language.

4.1 Object Model

The object model is responsible for constructing objects from object expressions and for evaluating and linking objects in an inheritance hierarchy. We first describe the representation of object expressions and object values, and then give an operational semantics for their construction.

Desugaring object expressions. At evaluation time we further desugar object expressions by means of two transformations. (1) We replace each field declaration by a slot with a getter and an optional setter method. (2) We lift the object initialization statements in the object expression into a constructor method. This transformation helps to keep the semantic rules concise without changing the meaning of programs. In Section 3 we have discussed name resolution for these desugared object expressions.

Desugared object expressions follow the signature defined in Figure 6. An object expressions (`LObj`) is a triple consisting of (1) a list of parent expressions (`LInh`) with aliases and exclusions, (2) a list of slot numbers (`LSlots`), and (3) a map (association list) mapping names to method declarations, including the derived getter and setter methods for fields and the object constructor method named `#ctr`. The transformation does not change the object hierarchy, does not evaluate inheritance expressions, is local to the object expression (i.e. does not require any global knowledge), and is stateless (i.e. is not dependent on program state).

```
module obj-desugar
imports grace-sig
signature
  sorts LInh
  sort aliases
    LAliases  = Map(String, String)
    LExcludes = Map(String, String)
    LSlots    = List(Int)
    LMethods  = List((String * Declaration))
  constructors
    LObj: List(LInh) * LSlots * LMethods → Exp
    LInh: Exp * LAliases * LExcludes     → LInh
```

Figure 6. Signature of desugared objects.

Representing objects. The signature in Figure 7 defines the structure of object instances which result from evaluation of `LObj` terms. The value visible in a Grace program is `RefV(a)`, a reference value to a store address `a` (objects are passed by reference rather than by value). The store associates addresses to object data.

An object (`Obj`) is a quadruple of (1) the store address of the lexically surrounding object, (2) an ordered list of parents (ancestors), (3) the object's data, a map of slot numbers to values, and (4) its operations, a map from method names to closures.

This representation of objects in the store encodes the scope graph information. Consider the scope graph fragment of Figure 8a. If we regard scope names (`s1`, `s2`, `s3` and `s4`) as store addresses then import edges from scope `s2` denote the addresses of the lexical parent and of objects inherited into the object at store location `s2`, as shown in Figure 8b. In the store, scope graph information is augmented with the objects' data and operations.

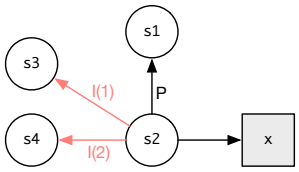
At any point during run time the store contains sufficient information to inspect the scope graph of a running program. We revisit this claim in Section 4.2 when we formalize the semantics of name resolution.

Binding `self` and `outer`. A method declaration closes over the address of the object which encloses its owner object. In Section 3 we modeled this as a declaration for the `outer` pseudo-variable in the method scope. It is the responsibility of the method call mechanism to bind the correct value for `outer`. The same mechanism also binds the `self` pseudo-variable which identifies the address of the object handling the method call. In our semantics we treat `self` and `outer` as semantic components, rather than explicit variables. Instead of the method call mechanism binding them as variables, it makes them available to method code as read-only evaluation contexts.

Figure 9 shows the declaration of components `S` (Self) and `O` (Outer). Resolving program references to `self` or `outer` becomes a matter of wrapping the contextual information as a value as we show in the rules of Figure 9. We believe this treatment of `self` and `outer` to lead to a more elegant semantics.

```
module obj-repr
imports store functions values
signature
  constructors
    RefV: Addr → V
  sorts Obj
  sort aliases
    Slots    = Map(Int, V)
    Methods  = Map(String, AnnotatedClosure)
    Aliases  = Map(String, String)
    Excludes = Map(String, String)
    Parent   = (Addr * Aliases * Excludes)
    Parents  = List(Parent)
  constructors
    Obj: Addr * Parents * Slots * Methods → Obj
  arrows
    Addr --outer--> Addr
    Addr --exist--> Bool
```

Figure 7. Representation of objects.



(a)

```
{s2 ↦ Obj(s1,[s3,s4],...,{x ↦ ..., #ctr ↦ ...})
 s1 ↦ ... s3 ↦ ... s4 ↦ ...}
```

(b)

Figure 8. (a) Scope graph fragment and (b) corresponding store after construction of `s2`

```
module self-outer
imports obj-repr store
signature components
  S : Addr
  O : Addr
rules
  S ⊢ Self()  ⟶ RefV(S)
  O ⊢ Outer() ⟶ RefV(O)
```

Figure 9. Semantics of `self` and `outer`.

Object construction and initialization. Object construction is concerned with evaluating an object expression to create a fresh object value in the store. It encodes the structure of the scope graph into the store, and produces a value referring to the new object. Object initialization is concerned with evaluating the initialization statements of objects.

Object construction as defined in rule `(1)` of Figure 10a consists of three stages: first the object is built using the $\xrightarrow{\text{bld}}$ arrow, second the policy governing local variable naming is enforced, and third the object is initialized using the `init-obj` meta-function.

Rule `(2)` builds the object structure in the store. It allocates a fresh (and empty) object in the store, evaluates parents (ancestors) and adds links to them, and creates slots and records method declarations. The `P` edge in the scope

```
module obj-constr
imports obj-desugar obj-repr obj-init
        obj-self-outer store
signature
  sorts EvalMode
  constructors
    E : EvalMode
    B : Addr → EvalMode
  components
    EB : EvalMode
  arrows
    Exp --bld--> Addr
    add-parents(Addr, List(LInh)) ⟶ U
    add-parent(Addr, LInh) ⟶ U
rules
  o --bld--> S';
  enforce-locals-policy(S') ⟶ _;
  S' ⊢ init-obj(S') ⟶ _
  ---------------------------------                    (1)
  o@LObj(_, _, _) ⟶ RefV(S')

  Obj(S, [], {}, {}) --store--> S';
  EB ⇒ B(S'') or S' ⇒ S'';
  S S'', O S ⊢ add-parents(S', ps) ⟶ _;
  O S ⊢ add-slots(S', ss) ⟶ _;
  O S ⊢ add-methods(S', ms) ⟶ _
  ---------------------------------------              (2)
  S, EB ⊢ LObj(ps, ss, ms) --bld--> S'

  par ⇒ LInherit(e, als, exs);
  EB B(S) ⊢ e ⟶ RefV(S'');
  record-parent(S', (S'', als, exs)) ⟶ _
  ---------------------------------------              (3)
  S ⊢ add-parent(S', par) ⟶ U()

  EB E() ⊢ e ⟶ recv; EB E() ⊢ es ⟶ vs;
  EB B(S) ⊢ call-qualified(recv, x, vs) ⟶ v
  ---------------------------------------------        (4)
  EB B(S) ⊢ MCallRecvL(e, ID(x), es) ⟶ v
```

(a)

```
signature arrows
  add-slots(Addr, LSlots) ⟶ U
  add-slot(Addr, LSlot) ⟶ U
  add-methods(Addr, LMethods) ⟶ U
  add-method(Addr, LMethod) ⟶ U
rules
  record-slot(S', {s ↦ def-val()}) ⟶ _
  ------------------------------------                 (1)
  add-slot(S', s) ⟶ U()

  enforce-method-policy(S', m) ⟶ _;
  method-closure(decl) ⟶ clos;
  record-method(S', {m ↦ clos}) ⟶ _
  ---------------------------------                    (2)
  add-method(S', (m, decl)) ⟶ U()
```

(b)

Figure 10. (a) semantics of object construction and (b) slot and method installation.

graph between nested objects materializes in the store as a link from the fresh object to its enclosing object identified by component `S`.

At this phase of object construction (i.e. after arrow $\xrightarrow{\text{bld}}$ completes), the state of the store reflects the structure of the scope graph. It contains sufficient information to perform name resolution. This information is used at this stage to enforce the policy governing local variable names (using the `enforce-locals-policy` meta-function). Policy enforcement must happen after building the object structure but before the object is initialized. There is insufficient information about inherited methods prior to construction. We discuss naming policies in more detail in Section 4.4, for now it suffices to know that enforcement will halt evaluation if any of the object's methods violates the policy.

As the final step rule `(1)` initializes the new object by invoking the constructor method (using the `init-obj` meta-function) in ancestor-first order. This order is customary in object-oriented languages.

Ancestor Evaluation. We discuss the semantics of computing and adding links to inherited objects, as defined in rule `(3)` of Figure 10a. During construction of a hierarchy the value of component `EB` keeps track of the youngest descendant in the hierarchy — the `self` of the new hierarchy of objects. `EB` is either `E()` if construction is just starting, or has value `B(S)` when evaluating the hierarchy of object `S`. Rule `(2)` triggers evaluation of inheritance expressions in `S` bound to the `self` of the hierarchy and in `O` bound to the `outer` of their owning object. Rule `(3)` evaluates an inheritance expression `e` to an ancestor `S''` and records it. Recording a parent is equivalent to adding an `I(i)` edge in the scope graph.

An ancestor object may not be initialized independently. Its initialization behavior will be modified by method overriding in descendants. The `init-obj` meta-function invoked by rule `(1)` skips initialization if `EB=B(_)`. Special handling of method calls allows regular computation to proceed until the object expression of the ancestor is reached. Rule `(4)` defines the semantics of a method call during ancestor evaluation (`EB=B(_)`). It evaluates the receiver `e` and parameters `es` in regular context (`EB=E()`) and the call in construction context (`EB=EB(_)`).

As an illustration of object construction consider constructing the object with scope `s3` and corresponding scope graph from Figure 11. When rule `(2)` of Figure 10a evaluates the object expression, the context is `S=s2` and `O=s1`. It first allocates the new object `s3` and then evaluates ancestors in a changed context: `S=s3` and `O=s2`. Rule `(3)` evaluates the inheritance expression `e` which eventually reaches the object expression for `s6`. Had `s6` itself had an inheritance expression `e2`, rule `(2)` would trigger its evaluation in context `S=s3` and `O=s5`.

We observe an interesting aspect of evaluating inheritance expressions: they evaluate in a context where `self` is bound to the youngest descendant in the inheritance hierarchy. Why should this be the case? Since programmers write inheritance expressions within an object block, they can reasonably expect that `self` and `outer` refer to the local object and its lexical parent, respectively. A problem occurs

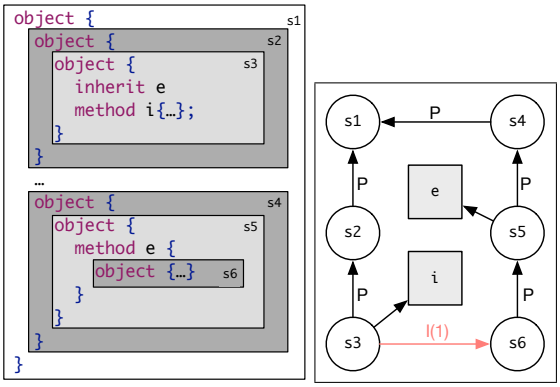


Figure 11. Fragment of program (left) and scope graph fragment after construction of `s3` (right).

though if an inheritance expression invokes a local method. For example, suppose that inheritance expression `e` from object `s3` of Figure 11 actually involves a call to method `i` in the same object. Since, by the definition in rule `(2)` of Figure 10a, ancestor expressions are evaluated before recording methods, method `i` does not yet exist in object `s3` and the program halts with an error. In rule `(3)` we model the object construction order of the official Grace implementation. It may actually confuse beginner programmers that a method which they can see and touch is reported by the Grace interpreter as unknown. Our semantics does not flatten object hierarchies. This implies that resolving overridden methods must happen during name resolution. The same holds for method aliasing and exclusion.

4.2 Name Resolution

Name resolution is the task of computing a path in the scope graph from a reference scope to a declaration scope. We first describe the structure of resolution paths, the outcome of name resolution, and then give a semantics for the name resolution algorithm of Grace.

Local variables. Grace has a strict no-shadowing policy for local variables. Two local variables in lexical range of each other must have different names. Grace has a strict no-shadowing policy for local variables. Each local variable must be unique in a chain of lexical scopes. We see this restriction as a simplification of name resolution. We choose to stray away from the scope graph paradigm and instead model local variables with traditional environment passing semantics. Rules propagate a read-only environment which associates variable names with to addresses in a variable store. The variable store is propagated as read-write component throughout the specification. Method declarations close over their declaration environment and the method call mechanism extends it parameters and local variables.

```
module path
signature
  sorts PathPart
  sort aliases
    Path = List(PathPart)
  constructors
    L: String        → PathPart
    I: Int * String → PathPart
    P: PathPart
  arrows
    Path -found-> Bool   Path -local-> Bool
    Path -lex->   Bool   Path -s-lex-> Bool
    Path -inh->   Bool   Path -s-inh-> Bool
```

Figure 12. Definition of resolution paths.

Using environments for local variables means that method resolution paths become shorter. There is possibly a performance benefit since a variable dereference requires fewer store operations.

Resolution paths. A path is the list of edges traversed between the reference scope and the declaration scope. We build on the paths of Section 3 and enrich them with more resolution information as defined in Figure 12. An empty path indicates failure to resolve. Arrow $\xrightarrow{\text{found}}$ reduces a path to a success value. The path resulted from successful resolution always ends with an `L(x)` segment, where `x` is the declaration name. It is easier to use a path later if it includes the declaration name. Paths into lexical scope have a `P()` edge. Paths into inherited scopes have an `I(i,x)` edge, where `i` is the index of the import edge and `x` is the name to resolve in the inherited scope. Suppose, for example, that resolving a reference `x1` results in a path `[P(),I(1,x2),L(x2)]`. This path indicates that the declaration is reachable in the lexical scope, where it was imported as an alias to the declaration of `x2`. Rich paths contain sufficient information to lookup the declaration and to enforce naming policies.

Resolution semantics. Name resolution is the task of finding a path in the scope graph from a reference scope to a declaration scope. The recursive name resolution algorithm is shown in Figure 13. Rule `(3)` over $\xrightarrow{\text{res}}$ encodes the base case of finding a local declaration named `x` in scope `S''`.

Two resolution directions are possible if the declaration is not local: in inherited scopes and in lexical scope.

Resolution in inherited scopes. The algorithm maps over inherited scopes (using the $\xrightarrow{\text{res-inhs}}$ arrow) in reverse order until it finds a matching declaration. Resolution in reverse inheritance order describes Grace's semantics of method overriding. Later inherited declarations should override earlier ones, so starting resolution at the last import ensures that overriding methods will be resolved instead of the overridden ones. Stopping after the first success guarantees that only the latest declaration is reached.

Figure 14 shows the semantics of name resolution in inherited scope. Rules over arrow $\xrightarrow{\text{res-inh}}$ resolve a reference `x` in

```
module method-resolution
imports path obj-repr
signature arrows
    resolve(String, Addr) ⟶ Path
    (String, Addr) -res-> Path
rules
  (x, S') -res-> p; p -found-> false;
  err("Unknown method " ++ x)
  ------------------------------                (1)
  resolve(x, S') ⟶ ???

  (x, S') -res-> p; p -found-> true;
  ------------------------------                (2)
  resolve(x, S') ⟶ p

  S' -read-> Obj(_, _, _, methods);
  methods[x?] ≡ true
  ------------------------------                (3)
  (x, S') -res-> [L(x)]

  S' -read-> Obj(O', parents, _, methods);
  methods[x?] ≡ false; O ≢ O';
  O, P 0 ⊢ (x, parents) -res-inhs-> p
  ------------------------------------          (4)
  O ⊢ (x, S') -res-> p

  S' -read-> Obj(O', ps, _, methods);
  methods[x?] ≡ false; O ≡ O';
  O ⊢ (x, ps) -res-inhs-> p-inh;
  O, S'  ⊢ (x, S') -res-lex-> p-lex;
  (x, p-inh, p-lex) -disamb-> p
  ------------------------------                (5)
  O ⊢ (x, S') -res-> p
```

Figure 13. Semantics of method resolution. ??? is a term placeholder for a rule which halts.

```
signature arrows
  (String, Parent) -res-inh-> Path
  (String, Parents) -res-inhs-> Path
rules
  exs[x?] ≡ true
  --------------------------                    (1)
  (x, (_, _, exs)) -res-inh-> []

  exs[x?] ≡ false;
  als[x?] ≡ true; als[x] ⇒ x';
  O ⊢ resolve(x', S') ⟶ p
  ------------------------------------------    (2)
  O, P ⊢ (x,(S', als, exs)) -res-inh-> [I(P,x')|p]

  exs[x?] ≡ false; als[x?] ≡ false;
  O ⊢ (x, S') -res-> p; p -found-> false
  ----------------------------------            (3)
  O, P ⊢ (x,(S', als, exs)) -res-inh-> []

  exs[x?] ≡ false; als[x?] ≡ false;
  O ⊢ (x, S') -res-> p; p -found-> true
  ------------------------------------------    (4)
  O, P ⊢ (x,(S', als, exs)) -res-inh-> [I(P, x)|p]
```

Figure 14. Method resolution in inherited scope

```
signature arrows
  (String, Addr) -res-lex-> Path
rules
  outer(S) ⟶ O; O -exist-> false
  ------------------------------                (1)
  (_, S) -res-lex-> []

  S -outer-> O; O -outer-> O'; O' -exist-> true
  O' ⊢ (x, O) -res-> p; p -found-> false
  --------------------------------------        (2)
  (x, S) -res-lex-> []

  S -outer-> O; O -outer-> O'; O' -exist-> true
  O' ⊢ (x, O) -res-> p; p -exist-> true
  --------------------------------------        (3)
  (x, S) -res-lex-> [P()|p]
```

Figure 15. Method resolution in lexical scope

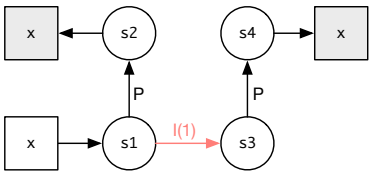


Figure 16. Scope graph with multiple resolution paths for a reference.

the inherited scope `S'` subject to aliases (`als`) and exclusions (`exs`). Rule `(1)` stops exploring the current inherited scope if the method was explicitly excluded on the inheritance expressions. If the reference was introduced as an alias, rule `(2)` will start a new resolution from the alias declaration to the target declaration. Rule `(2)` optimizes this aspect by concatenating the two resolution paths. Note that there is a error checking component here, since the resolution algorithm is reentered through the `resolve(x',S')` meta-function (rule `(1)` of Figure 13).

Resolution in lexical scope. Resolution in lexical scope (Figure 15) moves the resolution outwards by one lexical scope and sets the lexical context `O` accordingly. Objects may not act as proxies to their lexical scope, so lexical resolution is only permitted if it originated from nested scopes. Rules `(4)` and `(5)` of Figure 13 enforce this by checking equality of enclosing scopes. Rule `(5)` uses arrow $\xrightarrow{\text{disamb}}$ to choose the successful path or to raise an error if both an inherited and lexical declaration are reachable.

For example, consider the scope graph of Figure 16 and suppose we invoke name resolution for `x` in `s1`. Resolution starts with `O=s2`, and searches inherited scope `s3` after searching local scope `s1`. It cannot proceed lexically because `outer(s1)≠outer(s3)`, and resolution returns to scope `s1`. It may search in lexical scope and finds the declaration of `x` in scope `s2`.

4.3 Lookup Using Paths

Following a resolution path from the reference scope to the declaration itself is trivial. Figure 17 defines the path walking mechanism as rules over the $\xrightarrow{\texttt{lookup}}$ arrow. Given a name resolution path and a starting scope the algorithm walks the path and returns the pair of declaration scope and declaration. Since a path could not exist if name resolution failed no error handling code is necessary.

It is noteworthy that rule (3) of Figure 17 which looks up a declaration across an inherited scope edge `I(i, _)` does not actually return the scope containing the declaration. Instead it returns the inheriting scope. This gives the appearance that all inherited methods reside in the descendant object. This is consistent with semantics of object oriented languages in general.

4.4 Enforcing Name Policies

Grace has three policies regarding names: a confidential access policy, a local variable naming policy and a method naming policy. We briefly discuss each of them.

Local variable naming policy. We would like to have name policies which the language designer can easily modify. To achieve this we introduce a policy configuration which can be changed without modifying the enforcement mechanism. Figure 18 shows the local variable policy for Grace. It specifies whether a local variable name is legal in certain conditions. Rule (1) of Figure 10a for object construction enforces this policy by invoking the `enforce-locals-policy` rule after it has constructed the object. This rule iterates through every local variable of each method and checks its compliance with the configured policy.

Figure 18 illustrates the rejection mechanism for a local variable that shadows a method. The rules use the name resolution path `p` for a variable named `x` to decide whether its declaration as a local variable is legal. If the path is local ($p \xrightarrow{\texttt{local}} \textbf{true}$) then variable `x` shadows a method of the same object. If the path is strictly lexical ($p \xrightarrow{\texttt{s-lex}} \textbf{true}$), i.e. it contains at least one `P()` edge and no `I(_,_)` edges, then variable `x` shadows a method declared in an enclosing scope. If a resolution path does not exist then there is no member with that name that is shadowed. We observe that confidentiality of methods does not influence the enforcement of the policy, so there is no interplay between policies.

Method naming policy. We create a similar policy for method names, as shown in Figure 19. Rule (2) of Figure 10b enforces this policy before recording a method declaration.

Figure 19 illustrates how to forbid methods that shadow lexical methods. Rule (1) rejects a new method `m` if there is a strictly lexical path to a declaration. The language designer enables this behavior by setting `member-allow-shadow-lex()` to **`false`**. The effect obtained is that the naming policy for methods becomes symmetric to that of local variables.

```
module lookup
imports obj-repr path
signature arrows
  (Path, Addr) --lookup--> (Addr * AnnotatedClosure)
rules
  S --read--> Obj(_, _, _, methods)
  ---------------------------------                    (1)
  ([L(x)], S) --lookup--> (S, methods[x])

  S --outer--> S'
  --------------------------------                     (2)
  ([P()|p], S) --lookup--> lookup(p, S')

  S --read--> Obj(_, parents, _, _);
  parents[i] ⇒ (S', _, _);
  lookup(path, S') ⟶ (_, clos)
  ------------------------------------                 (3)
  ([I(i, _)|path], S) --lookup--> (S, clos)
```

Figure 17. Semantics of lookup.

```
module policy-locals
signature arrows
  local-allow-duplicates()        ⟶ false
  local-allow-shadow-local()      ⟶ false
  local-allow-shadow-method()     ⟶ false
  local-allow-shadow-inherited()  ⟶ true
rules
  local-allow-shadow-method() ⟶ false;
  (x, S) --res--> p; p --found--> true; p --local--> true;
  err("Variable '" ++ x ++ "' shadows
       member in surrounding object")
  ------------------------------------------           (1)
  S ⊢ enforce-local-shadow-method(x) ⟶ ???

  local-allow-shadow-method() ⟶ false;
  (x, S) --res--> p; p --found--> true; p --local--> false;
  p --s-lex--> true;
  err("Variable '" ++ x ++ "' shadows
       member in enclosing scope")
  ------------------------------------------           (2)
  S ⊢ enforce-local-shadow-method(x) ⟶ ???
```

Figure 18. Local variable name policy and example of enforcement semantics.

```
module policy-members
signature arrows
  member-allow-duplicates()      ⟶ false
  member-allow-override()        ⟶ true
  member-allow-shadow-local()    ⟶ false
  member-allow-shadow-lex()      ⟶ true
  member-allow-shadow-lex-inh()  ⟶ true
rules
  member-allow-shadow-lex() ⟶ false;
  (m, S) --res--> p; p --found--> true; p --s-lex--> true;
  err("Member '" ++ m ++ "' shadows
       member surrounding scope")
  ------------------------------------------           (1)
  S ⊢ enforce-member-shadow-lex(m) ⟶ ???
```

Figure 19. Method name policy and example of enforcement semantics.

Confidential access. In Section 3 we stated that enforcing confidentiality is a decision based on name resolution information and not a concern for the name resolution algorithm itself. Methods annotated `public` may be accessed from anywhere. `Confidential` methods may only be accessed from (1) within their declaration scope or (2) from descendant objects. We store each method together with its annotation so that it can be retrieved with $\xrightarrow{\texttt{lookup}}$. We take an access decision for a reference `x` given its scope s_{ref}, a resolution path `p`, a declaration scope s_{dec} and visibility annotation `a` as obtained from `-lookup->`.

If `a` is `private` then access should be granted in two cases: (1) if $s_{ref} = s_{dec}$ or (2) if `p` $\xrightarrow{\texttt{lex}}$**`true`**. Case (1) will hold when the reference scope is the same as the declaration scope. This is only the case when the reference scope is on the inheritance hierarchy. Case (2) occurs if `x` refers to a declaration in a lexically enclosing scope. The $\xrightarrow{\texttt{lex}}$ arrow checks that path `p` contains a `P()` edge.

If `a` is `public` access is permitted with one exception. Grace semantics state that methods introduced through aliases are `private` to the inheriting object. A path `p` that contains an `I(_, x')` edge where `x` $\neq$ `x'` indicates that method `x` is introduced through an alias. In that case we take an access decision using the mechanism described above for `private` methods.

5 Evaluation

We evaluate our executable specification of Grace with respect to three criteria: (1) correctness, (2) specification size and readability, (3) time complexity of name resolution and policy enforcement.

Correctness. We have developed an extensive test suite of unit-style tests for corner cases of name resolution and object model. Each test is a small program paired with an output/error expectation recorded from the output produced by the Grace to JavaScript implementation. The test set includes 214 test programs directly testing aspects of the object model and name resolution. The most notable of these are: 51 tests of the object model (object creation, inheritance and initialization), 31 tests of traits, aliasing and exclusion, 71 tests for scoping and 24 tests of confidentiality enforcement. We use these tests to validate that our executable specification behaves the same as the mainstream Grace implementation.

Size and readability. We compare implementation size of our solution to that of the Grace to JavaScript implementation, as a weak measure of maintenance burden. Our desugaring transformation and semantics specification together account for $2.0K$ LOC ($44.8K$ characters). In comparison the Grace transpiler accounts for $3.5K$ LOC ($70.8K$ characters). The compiler is quite small since Grace and JavaScript have similar abstraction levels. While our specification is complete for object model and name resolution, it does not yet cover pattern matching, lineups and Grace's gradual type system. We conjecture that the maintenance burden of our specification is not higher than that of the mainstream Grace compiler.

Code readability is subjective. The transpiler encodes Grace semantics in JavaScript which hides the semantics of Grace. Conversely the DynSem specification is explicit. A semanticist will have no difficulties understanding the DynSem specification since rules are similar to natural semantics. Our target audience consists of people wanting to understand (or develop) Grace semantics. We think that our DynSem specification is accessible for them.

Time complexity. The number of objects visited during name resolution dominates its execution time. We distinguish two cases: (1) resolving a reference from outside of the object (a qualified call on an object) and (2) a resolving reference from within an object scope. In the first case the time complexity of resolution is $O(n + 1)$ if there are n objects. The worst case corresponds to resolving a method that is declared in the last queried ancestor. For complexity of name resolution in case (2) assume that the object has n ancestors and m enclosing objects each with n ancestors and that the method declaration is inherited into the n^{th} furthest away ancestor of the m^{th} enclosing object. The time complexity of resolving the method is $O(m * (n + 1))$. Each ancestor inherited into every object has to be visited before the declaration is reached.

Enforcing naming policies for method declarations and local variables dominates the time complexity of object construction. If an object has m methods each with v local variables, then enforcing the former policy requires m name resolution operations and enforcing the latter policy requires $m * v$ name resolution operations.

The high cost of name resolution comes from the fact that semantics does not flatten objects. However, it gives a principled definition with a clear specification that can serve as a baseline for principled optimization.

6 Related Work

Object oriented languages

Record classes In 1965 Tony Hoare proposed handling records of similar entities as *record classes* and subclasses [31]. Record access was restricted to access qualified by the class or subclass containing the record declaration. Attribute access used dot notation to separate the receiver part of the access from the identifier of the record being accessed.

Simula 67 Hoare's proposal inspired extension of the Simula [22] programming language into Simula 67 [21] including the concepts of classes and subclasses. Subclasses may redeclare attributes from superclasses. Attribute references are disambiguated statically by qualifying references with the class containing the intended declaration. Simula 67 extends Hoare's record classes with the concept of *virtual procedures*,

placeholders for procedures that subclasses must provide implementation for. There are no abstract classes, classes containing virtual procedures may be instantiated but invocation of a virtual procedure will raise a run-time error. Calls to virtual procedures are dynamically bound. Simula 67 is considered the first object-oriented programming language and inspired the design of many other object-oriented languages.

Smalltalk Smalltalk [27] builds on Simula 67's concepts of classes and objects. Smalltalk is a dynamic object oriented language and interactive programming environment. Each method in a Smalltalk class is a *message handler* of the class. A method invocation is equivalent to an object receiving a message. The message handler is looked up by name starting at the receiving object and extending the search upwards in the inheritance chain until a handler is found. If a message handler is found the expressions in its body are executed. The linear handler search allows subclasses to redefine message handlers of superclasses. A message is not understood if a message handler cannot be found. If a receiver object does not understand a message, it is sent a *doesNotUnderstand:* message containing the original message which was not understood. At least one *doesNotUnderstand:* handler exists in the root object of all hierarchies.

The programmer may define custom *doesNotUnderstand:* handlers for custom behavior of messages which are not understood. Method-local variables must be declared at the beginning of the method body. Explicit qualification to the current receiving object and to the superclass of the receiving object is possible through the *self* and *super* pseudo-variables, respectively. When the receiver of a message is *super*, the search for a message handler begins in the superclass of the object sending the message.

Nested class declarations are not allowed in Smalltalk, however passing control structures is possible by passing blocks. Blocks are delimited sequences of expressions, closures over the declaration contexts which can be returned from methods and assigned to variables. They have to be explicitly evaluated. A search for a message handler from within a block begins in the block and continues to the declaring object.

Wolczko [59] gives a formal semantics (in a denotational style) for a significant subset of Smalltalk. The semantics formalize the message receiver search algorithm. The semantics do not make a distinction between the resolution and lookup phases of the search, they resolve a name directly to the handling method. A more recent [43] formalization of Smalltalk semantics as an operational semantics introduces the distinction between the two phases, however only in the context of local variables, the semantics of message handler search directly returns the message handler.

Borning et al. [8] extended Smalltalk with multiple-inheritance. Their approach to message handler resolution is to reuse the standard Smalltalk search algorithm for classes with a single superclass. For classes with multiple superclasses, methods inherited from other than the first superclass are recompiled into the inheriting class at the time of class creation. At run time methods can be found with the standard search algorithm but they may appear to come from class lower in the inheritance chain than where they were declared. Methods inherited from multiple superclasses are compiled to special error methods which raise run-time errors when invoked. Manual disambiguation is needed by the programmer for such methods and Borning et al. introduce specific syntax with this purpose.

BETA BETA [39] is an object-oriented programming language built around *patterns* as the single abstraction mechanism. Objects and procedures are both patterns which can be nested arbitrarily deep. Single inheritance is possible at the level of each pattern. Optional pattern parameters specify restrictions on the types of pattern attributes. Unlike in Smalltalk, BETA subclasses can only extend inherited methods, they cannot be redefined in a subclass. Name resolution traverses lexical scopes upwards from the pattern nearest to the reference location. Because subpatterns cannot redefine inherited patterns the name resolution algorithm can be seen to work downwards towards the reference. Confidential pattern attributes can be declared by nesting attribute declarations. This ensures that confidential attributes cannot be accessed from outside a pattern, but all legal references to the confidential attributes must be qualified with the name of the attribute holding the grouping.

Self A dialect of Smalltalk, the Self programming language [52] was the first programming language to have prototype-based inheritance instead of classes. In Self new objects are created by cloning and adaptation of existing objects. Of particular interest in Self is the unification of prototype attributes, methods and local variables. Prototype attributes are accessed via accessor and assigner methods. Methods and blocks are represented as prototypes with a slot for each local variable and message handlers for their accessors and assigners. Method prototypes have parent links to their enclosing objects. Name resolution of unqualified message sends begins lookup at the object corresponding to the enclosing method nearest to the reference. Self inspired the design of the widely-used JavaScript programming language.

CLOS The Common Lisp Object System (CLOS) [5, 6] is a powerful object oriented model supporting multiple inheritance. Unlike Smalltalk where a class may have a single parent, a CLOS class may have many parent classes. A parent class may appear multiple times in a class hierarchy. The CLOS system linearizes hierarchies such that each ancestor appears only once. Subclasses can redefine inherited methods. The function *call-next-method* is used to invoke

redefined methods. The message sent to the ancestor is fixed in a *call-next-method*.

The order in which redefined methods are arranged depends on the result of the hierarchy linearization, which in turns depends on the order in which parent classes appear in class declarations. A small change in the class hierarchy, such as removing one of many declarations of the same parent class in a hierarchy, can result in a different linearized class hierarchy and therefore in a different invocation order by *call-next-method* [12].

CLOS provides mixins which allow abstract classes (without an ancestor) to be mixed into the class hierarchy. Methods defined in mixins can invoke *call-next-method* although their enclosing classes do not have a parent. An ancestor is determined by the result of linearization. CLOS method lookup is provided by the *find-method* function returning a method object or raising an error if the method cannot be found and no error handling was provided.

Newspeak Newspeak [11] is an object oriented language in the Smalltalk tradition. The object model revolves around classes and superclasses. Classes can be nested arbitrarily deep, similarly to BETA classes. Nested classes can be overridden in descendant classes. Each Newspeak object maintains a link to its outer object, the object enclosing the class declaration (or object literal expression). Methods and blocks close over the object enclosing their declaration. There is no explicit mechanism to identify the outer object as the receiver of a message. The name resolution algorithm searches for message handlers locally, upwards in the inheritance chain, and outwards in the lexical scope, giving precedence to inherited members in favor of members in the enclosing scope. Inheritance chains of outer enclosing objects are never searched [9, 11], avoiding a comb-like search [10]. Class members may be annotated with *public* or *protected*. The access policy is enforced by the name resolution algorithm.

C++ C++ [51] is an object-oriented programming language drawing from C and Simula. C++ has a class-based object model with multiple inheritance. Class on an inheritance trees are linearized (concatenated) but the order of linearization is not specified and is compiler-implementation specific. Classes appearing multiple times in an inheritance tree will appear multiple times in the linearization result, hence their members will also appear multiple times. This can be avoided by declaring ancestor classes as *virtual*. Members of virtual classes will only appear once in the concatenation. Descendant classes may hide fields of inherited classes and may override inherited functions. Class functions having different number of parameters may have the same name. C++ allows nested class declarations. Nested classes may declare fields which hide fields declared in enclosing classes, or variables in the global scope. Names can be resolved in the scope of enclosing classes by qualifying them with the name of the intended class. Hidden global variables can be accessed using the `::` prefix operator. Unqualified references occurring in the definition body of a member function are looked up locally, in the local class, in the ancestor classes and in the enclosing classes if any. Unqualified references appearing in class definitions outside of member function definition bodies are looked up in the class declarations above the reference, in the entire ancestors and in the enclosing classes' declarations above the declaration of the class containing the reference.

Each class member may have one of the *private*, *protected* or *public* visibility annotations. Public members may be referenced from anywhere, protected members may be accessed from subclasses and from nested classes, private members may only be accessed from within the class defining them (implicitly also from the inner classes). An exception are *friend* functions - specially declared functions that reside outside the class declaration, which may access members of the class they contribute to as if residing in the class' definition body. Wallace [58] extends the algebraic semantics specification of C [30] with semantics of the object model.

C++ has a mechanism for explicitly scoping declarations in a named scope - a *namespace*, which may be nested arbitrarily deep. Declarations contained in namespaces can be imported into any local scope.

Java Java [28] is a popular class-based object oriented programming language. Java classes have single inheritance of functionality and multiple inheritance of type, i.e a class may have at most one ancestor class but may implement multiple interfaces. Classes can be nested arbitrarily deep. Subclasses may shadow inherited fields and may overload and override inherited methods. Inner classes and methods may shadow fields and methods in the enclosing scope. The same naming rules apply to anonymous classes (inline class declaration expressions). Nested and anonymous classes may refer to variables in the enclosing scope. Java provides visibility annotations with names and semantics similar to those from C++.

Name resolution is performed statically, at compile time. Class, field and method names reside in different namespaces and cannot conflict with one another. However, their references cannot be distinguished syntactically (with the exception of method reference). Reference resolution begins in the local scope, advances to the inherited classes and then searches outwards in enclosing scopes. Members inherited into enclosing scopes are considered during resolution. The rules of shadowing dictate how disambiguation of classes, fields and package names takes place. For example, a qualified reference to a field of a class in a package (`p.c.f`) may become a reference to a field of a field `f` in the type of field `c` if a class declaration for `p` is introduced at some point in the chain of enclosing scopes from the reference location to the root scope. The Java compiler compiles source code

to Java bytecode. References that appear in emitted bytecode are fully qualified by a path from the root scope to the declaration.

Various formalizations of Java exist. Featherweight Java (FJ) [34] is a minimal, but extendable, core calculus which mimics Java's object and name resolution models with various simplifications (no local variables, no interfaces, etc.). The goal of FJ is to provide a sufficiently small calculus such that reasoning about programs becomes humanly tractable. FJ is accompanied by formal definitions for its static and dynamic semantics. The semantics for field and method resolution directly reduce references to their declarations, i.e. no resolution path is explicitly computed. K-Java [7] is a complete specification of static and dynamic semantics of Java 1.4 in K [48]. The static semantics act as an elaboration phase which to emit a modified program AST containing, among others, type annotations on references. Each (part of a qualified) reference is wrapped in a type cast. For example, a reference `v` may be elaborated to `(int)((A) this).v`. Such type casts serve as name resolution information at run time. They can also be seen as a materialization of part of the resolution path for a qualified reference. The final part of the resolution path, the part indicating which parent in the class hierarchy contains the referred declaration, is not elaborated. The elaboration phase also annotates actual parameter expressions in method calls with their expected type, thereby keeping the dynamic semantics void of method overloading rules.

Obliq Obliq [18] is an untyped interpreted object-oriented programming with distributed computation. Obliq does not have classes, it only has object literal expressions, which can be nested arbitrarily deep. All declarations are lexically scoped. Each object resides at a single computation site, the site which created it but can be referenced and its members may be invoked from other sites. The result is a program state which is global but an object state which is local to each to site. Obliq provides four semantic operations on objects: select (invoke), update (assign), clone and delegate. Delegation on an object introduces new members, or overrides existing ones, as aliases to existing methods. Delegate members can be introduced at point in the lifecycle of an object. Inheritance of objects is obtained through cloning ancestor methods.

A method may modify the object that contains its declaration - a *self*-inflicted operation for the object, or it may modify another object - an external operation on the other object. External operations can come either from the same site or from a different site. Objects can be declared to be *protected*. Protected object reject operations which are not *self*-inflicted.

Name resolution starts in the scope closest to the reference and moves outwards towards the root scope. A reference to a remote object can be obtained from a name server. A reference to a remote object can be used identically to a local object. Name resolution is performed dynamically but the results are cached to reduce lookup time. Obliq objects are automatically garbage collected, reference counts to objects are maintained even across sites.

Formal definitions of object-oriented languages Besides the formal definitions of languages mentioned in the previous section, our work is related to formalizations of other object oriented languages.

Baby Modula-3 [2] is a subset of Modula-3 [19] restricted to focus on the object model. A formalized static and dynamic semantics is given for Baby Modula-3. The operational semantics are trivial with respect to object creation and name resolution. Object values are represented as a pair of symbol tables - named fields and named methods. Objects are constructed incrementally by inheriting from existing objects and updating the associative arrays of the inherited object. Since a name is always resolved locally in one of the two associative arrays, a distinction is not made between name resolution and lookup in either of the denotational and operational semantics.

C# [1] is a class-based object oriented programming language. A dynamic semantics following the language specification is proposed by Börger [17]. The definition is given in terms of orthogonal extensions of semantics. The definition assumes that the program has already been processed by the C# compiler. As such all class instantiations, field and method references are already resolved and no name resolution algorithm is needed.

Objective ML [49] is an extension of ML with class-based objects. The object model of Objective ML is at the base of the OCaml object model. Rémy et al. describe a static and dynamic semantics of Objective ML. Classes are reduced to structs with strict member ordering which simplifies the problem of class inheritance. Name resolution is encoded in the evaluation contexts by the reduction rules of class instantiation. The resolution of a reference remains a matter of looking up in name in the intended evaluation context.

TOOPLE [14] and PolyTOIL [15, 16] are statically typed object oriented languages designed to be provably type safe. PolyTOIL is a polymorphic extension to TOOPLE. Both languages have formal definitions of their operational semantics. In both cases the semantics model a flattened object representation. The semantic rules for class update and class extension encode the intended name resolution behavior in their result. The outcome is that a qualified reference is resolved by evaluating the receiver which must locally define the referred name.

JavaScript [23] is a popular dynamically typed object-oriented language inspired from Self, widely used in web browsers and increasingly on the server-side. Its semantics are lengthily and informally described in the language

specification. Maffeis et al. [40] give a small-step operational semantics of JavaScript. Object are stored on a heap, are not flattened and are referred to by reference values in the semantics. Each object has a reference to its prototype object. Functions are special types of objects with a prototype common to all functions. Static scopes [1] are encoded as hierarchies of scope objects. Resolution of an identifier involves looking up the identifier in the nearest scope object or upwards in its inheritance chain. The semantics given by Maffeis et al. use an assortment of meta-functions to specify the name resolution (`Scope`, `HasProperty`, etc.). The `Scope` meta-function returns the address of the (scope) object defining the identifier. The traversal path to that object is not returned.

Guha et al. [29] take a different approach to a formal semantics for JavaScript. Their approach is to define the semantics of JavaScript core, λ_{JS}, and to desugar whole JavaScript programs to λ_{JS}. The full system is able to pass the entire JavaScript test suite.

Objects are modeled in the core language. Each object has a `__proto__` attribute which contains a reference to the prototype object. Objects are not flattened. Functions are modeled as objects where local variables are object attributes and the code portion is encapsulated in a `code` attribute. The λ_{JS} core has simple lexical scoping. Lexical scoping alleviates the need to represent scopes as objects. The semantics of name resolution are to perform eager substitution. Much of the complex name resolution logic of JavaScript is embedded in the desugaring rules.

A similar approach is taken by Politz et al. [46] in defining the semantics of Python. Python is notorious for its complicated scoping rules, for its flexibility and for the fact that it has classes with multiple inheritance. Politz et al. introduce a desugaring of Python programs to a Python core - λ_{π}, and a semantics for this core language. Much of the scoping issues are resolved in the desugaring by using `let` statements. The desugaring is context-aware and therefore embeds name resolution logic. In λ_{π} semantics all `let`-bound variables are heap allocated and all unqualified variable references are resolved directly from the heap. Qualified variable reference, i.e. field accesses, are either resolved locally to the class or the class hierarchy is traversed. The class hierarchy is bound as the `__mro__` (*method resolution order*) attribute for each class. It contains the list of ancestor classes. The linearized class hierarchy is computed and bound to the `__mro__` attribute at class initialization time.

Dynamic semantics definition formalisms DynSem as a semantics definition language is most related to I-MSOS [42]. It borrows the notion of semantic components from MSOS [41] and propagates these implicitly as in I-MSOS. Specifications in DynSem are typically given in a big-step (*natural semantics*) style [35]. Use of meta-functions and relying on implicit reductions leads to specifications that resemble SOS [45] semantics, but the behavior is still big-step.

Other notable semantics formalisms are as follows. *funcons* [20] is a formalisms which aims to provide a definitive collection of reusable semantics. Redex [26] embeds meta-notation for semantics as Felleisen-Hieb reduction rules. K [48] is a language and toolchain for dynamic semantics specification. It has been applied to production-sized languages (C [24] and Java [7]). Semantics in K are given in terms of rewrite rules.

DynSem specification are interpreted. Most of the semantics definition languages mentioned above support execution: *funcons* generates an interpreter in Haskell, the Redex interpreter enacts reduction semantics, K generates an interpreter in Maude.

Language workbenches We used the Spoofax language workbench [36, 56] to specify Grace. The Spoofax approach is to provide a closely nit set of meta domain specific languages for each aspect of a language's definition: SDF3 [57] for syntax definition, NaBL [38] for name bind and type system rules, Stratego [13] for program transformation and DynSem [55] for dynamic semantics. Spoofax is part of the family of language workbenches. In Rascal [37] the approach is to use a single language for all aspects of a language. There is no specific formalism for dynamic semantics, but interpreters can be written using rewrite rules. In the Redex [26] approach a single language is extended with meta-notation for the various aspects of a programming language. Dynamic semantics are given as reduction semantics using specific meta-notation. Other language workbenches exist, Erdweg et al. [25] provide a thorough comparison.

7 Conclusion and Future Work

We have modeled the run-time name resolution of Grace using the scope graph paradigm. This served as a basis for discussion of the key aspects of name resolution in Grace. We defined operational semantics for the object model which encodes the name binding information from scope graphs into the object representation. We have defined the operational semantics of the name resolution algorithm in Grace. Separating name resolution from naming policies allowed us to keep the name resolution algorithm concise. We have shown how name resolution results are used to enforce naming policies. The specification as a whole serves as readable documentation and as executable specification that can be used for experimental validation and as reference implementation. We developed an extensive suite of unit-style tests and used it to validate the correct behavior of the specification with respect to the mainstream Grace implementation.

As future work, we plan to explore principled optimizations of the name resolution algorithm, in particular to find ways to reduce the number of objects that name resolution must search. One idea is to define evaluation contexts for

[1] JavaScript's scoping is arguably non-lexical

which caching of name resolution results is allowed. We also plan to define the semantics of the missing features: pattern matching, lineups and the gradual type system. Our hope is that by collaborating with the Grace community the specification becomes a reference implementation of the language.

References

[1] 2013. C# Language Specification 5.0. (June 2013). Copyright 1999-2012 Microsoft Corporation. All rights reserved.

[2] Martín Abadi. 1994. Baby Modula-3 and a Theory of Objects. *Journal of Functional Programming* 4, 2 (1994), 249–283.

[3] Hubert Baumeister, Harald Ganzinger, Georg Heeg, and Michael Rüger. 1987. Smalltalk-80. *it - Information Technology* 29, 4 (1987), 241–251. DOI: http://dx.doi.org/10.1524/itit.1987.29.4.241

[4] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. 2012. Grace: the absence of (inessential) difficulty. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012*, Gary T. Leavens and Jonathan Edwards (Eds.). ACM, 85–98. DOI: http://dx.doi.org/10.1145/2384592.2384601

[5] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. 1989. Common Lisp Object System Specification: 1. Programmer Interface Concepts. *Higher-Order and Symbolic Computation* 1, 3-4 (1989), 245–298.

[6] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. 1989. Common Lisp Object System Specification: 2. Functions in the Programmer Interface. *Higher-Order and Symbolic Computation* 1, 3-4 (1989), 299–394.

[7] Denis Bogdanas and Grigore Rosu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 445–456. DOI: http://dx.doi.org/10.1145/2676726.2676982

[8] Alan Borning and Daniel H. H. Ingalls. 1982. Multiple Inheritance in Smalltalk-8O. In *AAAI*. 234–237.

[9] Gilad Bracha. 2007. Executable Grammars in Newspeak. *Electronic Notes in Theoretical Computer Science* 193 (2007), 3–18. DOI: http://dx.doi.org/10.1016/j.entcs.2007.10.004

[10] Gilad Bracha. 2007. On the interaction of method lookup and scope with inheritance and nesting. In *In 3rd ECOOP Workshop on Dynamic Languages and Applications (DYLA)*.

[11] Gilad Bracha. 2017. Newspeak programming language draft specification version 0.1. http://newspeaklanguage.org/spec/newspeak-spec.pdf. (February 2017).

[12] Gilad Bracha and William R. Cook. 1990. Mixin-based Inheritance. In *OOPSLA/ECOOP*. 303–311.

[13] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. 2006. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae* 69, 1-2 (2006), 123–178. DOI: http://dx.doi.org/openurl.asp?genre=article&issn=0169-2968&volume=69&issue=1&spage=123

[14] Kim B. Bruce, Jonathan Crabtree, and Gerlad Kanapathy. 1993. An Operational Semantics for TOOPLE: A Statically-Typed Object-Oriented Programming Language. In *Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA, April 7-10, 1993, Proceedings (Lecture Notes in Computer Science)*, Stephen D. Brookes, Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt (Eds.), Vol. 802. Springer, 603–626.

[15] Kim B. Bruce, Angela Schuett, and Robert van Gent. 1995. PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language. In *ECOOP 95 - Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science)*, Walter G. Olthoff (Ed.), Vol. 952. Springer, 27–51. DOI: http://dx.doi.org/link/service/series/0558/bibs/0952/09520027.htm

[16] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. 2003. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems* 25, 2 (2003), 225–290. DOI: http://dx.doi.org/10.1145/641888.641891

[17] Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. 2005. A high-level modular definition of the semantics of C#. *Theoretical Computer Science* 336, 2-3 (2005), 235–284. DOI: http://dx.doi.org/10.1016/j.tcs.2004.11.008

[18] Luca Cardelli. 1995. A Language with Distributed Scope. In *POPL*. 286–297.

[19] Luca Cardelli, James E. Donahue, Lucille Glassman, Mick J. Jordan, Bill Kalsow, and Greg Nelson. 1992. Modula-3 language definition. *SIGPLAN Notices* 27, 8 (1992), 15–42. DOI: http://dx.doi.org/10.1145/142137.142141

[20] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. 2015. Reusable Components of Semantic Specifications. *Transactions on Aspect-Oriented Software Development* 12 (2015), 132–179. DOI: http://dx.doi.org/10.1007/978-3-662-46734-3_4

[21] Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. 1967. Simula 67 common base language. *Norwegian Computing Center* (1967).

[22] Ole-Johan Dahl and Kristen Nygaard. 1966. SIMULA - an ALGOL-based simulation language. *Commun. ACM* 9, 9 (1966), 671–678. DOI: http://dx.doi.org/10.1145/365813.365819

[23] ECMA. 2009. ECMA-262 ECMAScript Language Specification. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf. (December 2009).

[24] Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 533–544. DOI: http://dx.doi.org/10.1145/2103656.2103719

[25] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly 0001, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. DOI: http://dx.doi.org/10.1016/j.cl.2015.08.007

[26] Matthias Felleisen, Robby Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.

[27] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

[28] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification. Java SE 8 Edition*.

[29] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science)*, Theo D Hondt (Ed.), Vol. 6183. Springer, 126–150. DOI: http://dx.doi.org/10.1007/978-3-642-14107-2_7

[30] Yuri Gurevich and James K. Huggins. 1992. The Semantics of the C Programming Language. In *Computer Science Logic, 6th Workshop, CSL 92, San Miniato, Italy, September 28 - October 2, 1992, Selected Papers (Lecture Notes in Computer Science)*, Egon Börger, Gerhard Jäger, Hans Kleine Büning, Simone Martini, and Michael M. Richter (Eds.), Vol. 702. Springer, 274–308.

[31] C Hoare. 1965. Record handling. *Algol Bulletin* 21 (1965).

[32] Michael Homer, James Noble, Kim B. Bruce, and Andrew P. Black. 2013. Modules and dialects as objects in Grace. *School of Engineering and Computer Science, Victoria University of Wellington* (2013).

[33] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. 2012. Patterns as objects in grace. In *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, Tucson, AZ, USA, October 22, 2012*, Alessandro Warth (Ed.). ACM, 17–28. DOI: http://dx.doi.org/10.1145/2384577.2384581

[34] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001), 396–450. DOI: http://dx.doi.org/10.1145/503502.503505

[35] Gilles Kahn. 1987. Natural Semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings (Lecture Notes in Computer Science)*, Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.), Vol. 247. Springer, 22–39.

[36] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 444–463. DOI: http://dx.doi.org/10.1145/1869459.1869497

[37] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 168–177. DOI: http://dx.doi.org/10.1109/SCAM.2009.28

[38] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. 2012. Declarative Name Binding and Scope Rules. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science)*, Krzysztof Czarnecki and Görel Hedin (Eds.), Vol. 7745. Springer, 311–331. DOI: http://dx.doi.org/10.1007/978-3-642-36089-3_18

[39] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. 1983. Abstraction Mechanisms in the Beta Programming Language. In *POPL*. 285–298.

[40] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings (Lecture Notes in Computer Science)*, Ganesan Ramalingam (Ed.), Vol. 5356. Springer, 307–325. DOI: http://dx.doi.org/10.1007/978-3-540-89330-1_22

[41] Peter D. Mosses. 2004. Modular structural operational semantics. *Journal of Logic and Algebraic Programming* 60-61 (2004), 195–228. DOI: http://dx.doi.org/10.1016/j.jlap.2004.03.008

[42] Peter D. Mosses and Mark J. New. 2009. Implicit Propagation in Structural Operational Semantics. *Electronic Notes in Theoretical Computer Science* 229, 4 (2009), 49–66. DOI: http://dx.doi.org/10.1016/j.entcs.2009.07.073

[43] Jukka Mäki-Turja, K Post, and Jan Gustafsson. 1997. An operational semantics for Smalltalk. *Department of Computer Engineering, Mälardalen University, Sweden* (1997).

[44] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 205–231. DOI: http://dx.doi.org/10.1007/978-3-662-46669-8_9

[45] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming* 60-61 (2004), 17–139.

[46] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: the full monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 217–232. DOI: http://dx.doi.org/10.1145/2509136.2509536

[47] Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. 2016. Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. DOI: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20

[48] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. DOI: http://dx.doi.org/10.1016/j.jlap.2010.03.012

[49] Didier Rémy and Jerome Vouillon. 1998. Objective ML: An Effective Object-Oriented Extension to ML. *TAPOS* 4, 1 (1998), 27–50.

[50] Alan Snyder. 1986. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA*. 38–45.

[51] Bjarne Stroustrup. 1986. C++ Programming Language. *IEEE Software* 3, 1 (1986), 71–72.

[52] David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *OOPSLA*. 227–242.

[53] Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 49–60. DOI: http://dx.doi.org/10.1145/2847538.2847543

[54] Vlad Vergu, Michiel Haisma, and Eelco Visser. 2017. *The Semantics of Name Resolution in Grace.* Technical Report TUD-SERG-2017-011. Software Engineering Research Group, Delft University of Technology. Available at http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2017-011.pdf.

[55] Vlad A. Vergu, Pierre Néron, and Eelco Visser. 2015. DynSem: A DSL for Dynamic Semantics Specification. In *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland (LIPIcs)*, Maribel Fernández (Ed.), Vol. 36. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 365–378. DOI: http://dx.doi.org/10.4230/LIPIcs.RTA.2015.365

[56] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël D. P. Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SLASH '14, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz (Eds.). ACM, 95–111. DOI: http://dx.doi.org/10.1145/2661136.2661149

[57] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. 2012. Declarative specification of template-based textual editors. In *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, Anthony Sloane and Suzana Andova (Eds.). ACM, 8. DOI: http://dx.doi.org/10.1145/2427048.2427056

[58] Charles Wallace. 1993. *The semantics of the C++ programming language.* Citeseer.

[59] Mario Wolczko. 1987. Semantics of Smalltalk-80. In *ECOOP 87 European Conference on Object-Oriented Programming, Paris, France, June 15-17, 1987, Proceedings (Lecture Notes in Computer Science)*, Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman (Eds.), Vol. 276. Springer, 108–120. DOI: http://dx.doi.org/link/service/series/0558/bibs/0276/02760108.htm

TUD-SERG-2017-011
ISSN 1872-5392