The semantics of name resolution in Grace

Vergu, Vlad; Haisma, Michiel; Visser, Eelco

DOI
10.1145/3133841.3133847

Publication date
2017

Document Version
Final published version

Published in
DLS Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017

Citation (APA)
Vergu, V., Haisma, M., & Visser, E. (2017). The semantics of name resolution in Grace. In D. Ancona (Ed.), *DLS Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017* (pp. 63-74). (ACM Sigplan Notices- DLS'17; Vol. 52, No. 11). Association for Computing Machinery (ACM). https://doi.org/10.1145/3133841.3133847

The Semantics of Name Resolution in Grace

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

Michiel Haisma
TU Delft
The Netherlands
michielhaisma@gmail.com

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

ACM Reference Format:
Vlad Vergu, Michiel Haisma, and Eelco Visser. 2017. The Semantics of Name Resolution in Grace. In *Proceedings of 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3133841.3133847

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 [1], Self [31], Newspeak [7] and Java [15]. 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, 17, 18]. 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 [7, 31], (2) Smalltak-80 has an operational semantics [37] but Smalltalk lacks nested objects, and (3) Java's semantics [30] in K [5] 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, available at https://github.com/MetaBorgCube/metaborg-grace/tree/dls17, 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 [26, 32] and to operationalize this model as a specification for Grace in the Spoofax language workbench [20, 35].

Figure 1 shows the architecture of our implementation. A syntax definition in SDF [36] derives a parser with error

DLS'17, October 24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5526-1/17/10.
https://doi.org/10.1145/3133841.3133847

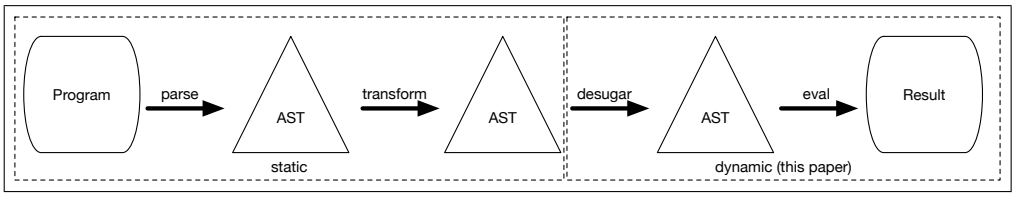


Figure 1. Architecture of the Grace language artifact.

recovery and an IDE with syntactic editor services. Source-to-source transformations implemented in Stratego[8] desugar high-level Grace features to lower level concepts. We use scope graph notation to model the key aspects of name resolution in desugared programs. An operational semantics in DynSem [34], 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.
- 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 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 features of Grace in terms of lower level concepts using a transformation implemented in Stratego [8]. 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) 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

```
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.

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 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 [6]. 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 [26, 32]. 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.

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. It has two declarations: one for field `x` and one for method `m(_)`. Method `m(_)` has its own scope `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 this parameter `y`. Scope `s3` is the scope of the object created by method `m(_)`. The `L` edge from `s3` to `s2` corresponds to the nesting of the object expression in the method scope. Distinguishing `P` and `L` edges allows object and method scopes to be distinguished, a requirement for policy enforcement.

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 [29]. 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 can be found in the extended version [33].

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 with scope `s4`. This 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

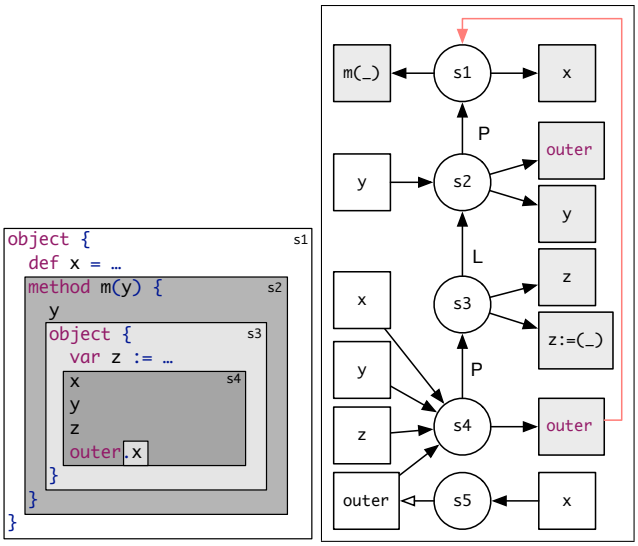


Figure 3. Program illustrating nested scopes and its scope graph.

to the declaration scope and this resolution path contains only `P` and `L` edges. In Figure 3, declarations in scope `s2` of method `m(_)` are reachable only from `s2`, `s3` and `s4`.

`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).

Every method implicitly declares an `outer`. The `outer` pseudo-variable can be used to qualify references to members in surrounding objects. Enclosing objects can be reached 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`.

Fields are slots with getters and setters. As Figure 3 shows, fields and method declarations 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.

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.

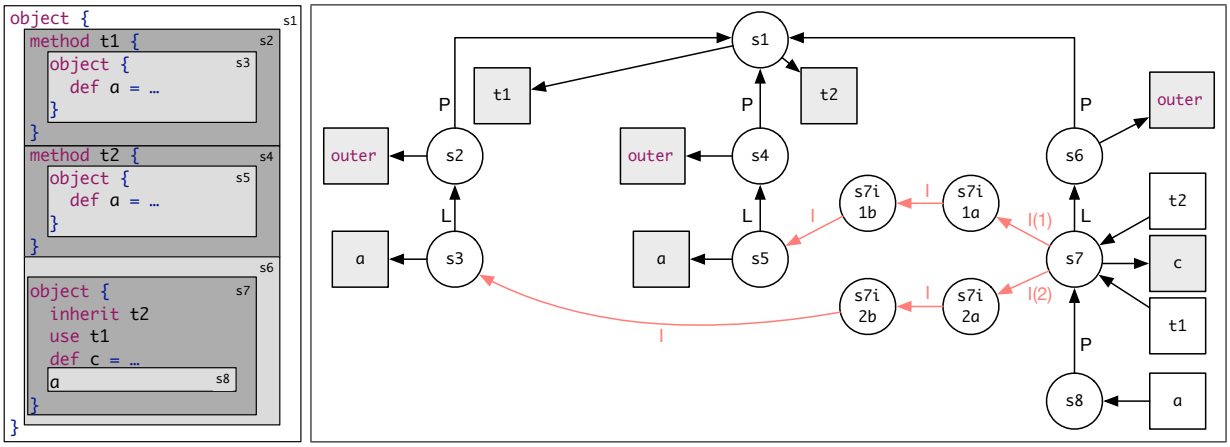


Figure 4. Scope graph (right) for a program with objects having multiple inheritance (left).

Consider the scope graph of Figure 4. The inheritance expression `t2` is resolved in the inheriting scope `s7`. 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 at run time after evaluating the inheritance expression. (We discuss the auxiliary import 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, resolving reference `a` in scope `s8` of the constructor method of object with scope `s7` of Figure 4 yields the resolution path `[P,I(1),I,I]` to `s5`.

Allowing arbitrary inheritance expressions increases the expressivity of the language but complicates name resolution. Meaningful static name resolution requires intra- and inter-procedural data-flow analyses.

Objects have multiple ancestors. Objects in Grace can inherit from multiple traits. Traits are just objects without state, a restriction enforced syntactically. A desugaring rewrites trait declarations to factory methods. Consider the object with scope `s7` of Figure 4 which inherits as a trait the object scope `s3` identified by `t1`. 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.

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.

Lexical scope may not be imported. Objects may not be used as proxies for their lexical scopes. It is a design decision that maintains encapsulation and ensures privacy of an object's internal details. The restriction has a natural parallel in the scope graph model: `I` edges may only be followed by `I` edges.

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 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`. If a path without any `I` edge exists, the new declaration for `x` would be in lexical range of another declaration and must be rejected. The declaration violates restriction (1) if the last path edge is `L`, or restriction (2) otherwise.

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.

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

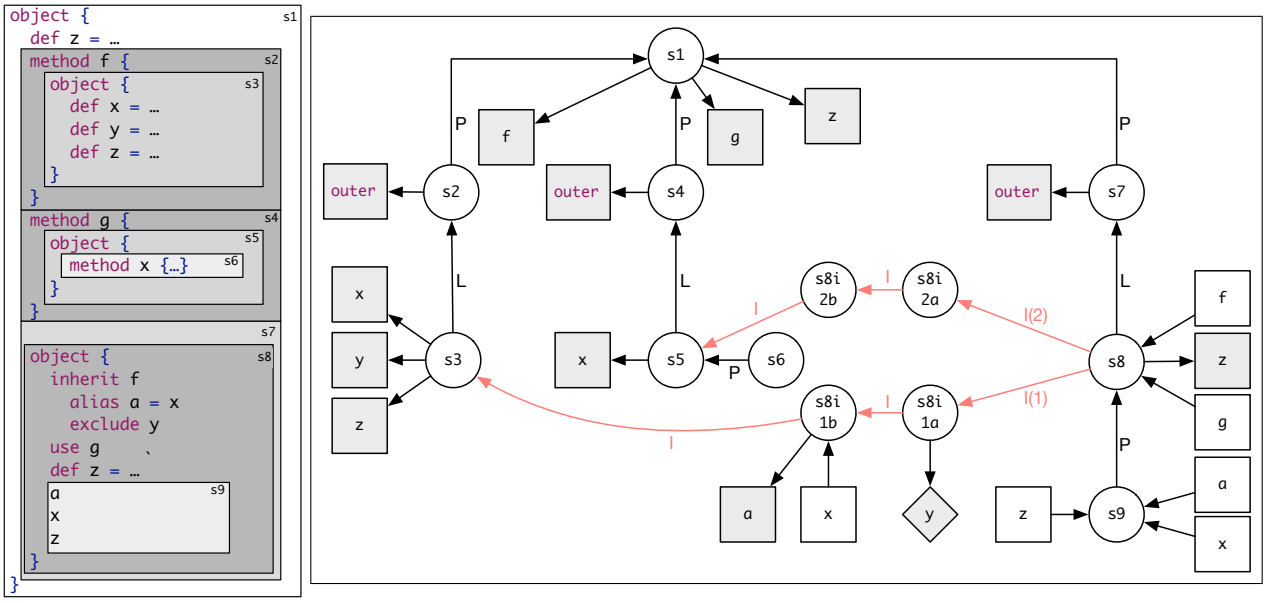


Figure 5. Multiple inheritance with aliases and excludes, shadowing and overriding (left) and corresponding scope graph (right).

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.

A local declaration overrides inherited declarations. For example, resolution prefers path `[P]` over path `[P,I(1),I,I]` when resolving reference `z` in `s9`.

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, and enforcement of naming policies.

We use DynSem [34] 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 ⊢ symbol) and from read-write components. A rule can omit semantic components which it does not use; these are implicitly propagated. Read-only components propagate downwards (environment semantics), read-write components thread through the rules (store semantics). The DynSem compiler derives an interpreter for the object language from the specification.

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

```
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.

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`.

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. Given a store address the $\xrightarrow{\texttt{exist}}$ arrow checks whether the address is allocated in the store and the $\xrightarrow{\texttt{outer}}$ arrow accesses the address of the enclosing object.

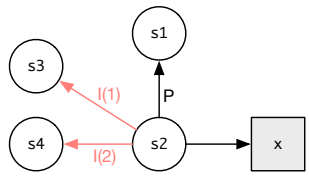
This representation of objects in the store encodes the scope graph information. Consider the scope graph fragment of Figure 8a. If we regard scope names (`s1`, `s2`, `s3` and `s4`) as store addresses then import edges from scope `s2` denote the addresses of the lexical parent and of objects inherited 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

```
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`.

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.

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

```
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 ⇒ LInh(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.

referring to the new object. Object initialization is concerned with evaluating the initialization statements of objects.

An object expression may be evaluated to either the root of an object hierarchy (the youngest descendant) or to an ancestor node. Run-time context determines in which mode to evaluate an object expression. The same object expression may be evaluated in different modes throughout the lifetime of a program. We must distinguish between the two modes in order to correctly initialize object hierarchies. Figure 10a introduces component EB which will propagate in read-only fashion to object construction rules. The component has a dual purpose: (1) to maintain the current evaluation mode and (2) to hold the address of the hierarchy root (the `self` of a hierarchy) when evaluating ancestors. EB will be `E()` when evaluating code that is part of a hierarchy root and conversely EB is `B(S)` when evaluating an ancestor of the hierarchy rooted at the object with address `S`.

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, creates slots and records method declarations. The `P` edge in the scope graph between nested objects materializes in the store as a link from the fresh object to its enclosing object identified by component `S`.

After arrow $\xrightarrow{\text{bld}}$ completes, the state of the store reflects the structure of the scope graph and 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 happens after an object's structure is created but before it is initialized. There is insufficient information about inherited methods prior to construction. We discuss naming policies 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. Note that rule `(1)` evaluates both hierarchy roots and ancestor objects. To prevent initialization of incomplete hierarchies the `init-obj` meta-function is a `noop` when EB is `B(_)`.

Ancestor Evaluation. We discuss the semantics of evaluating ancestors and linking descendants to them. Rule `(2)` of Figure 10a evaluates each ancestor using rule `(3)`. It determines the `self` of the hierarchy root and passes it to rule `(3)`. Rule `(3)` evaluates the ancestor expression `e` with EB set to `B(S)` to flag that ancestor initialization should be deferred. Rule `(4)` defines the special semantics of method calls when EB is `B(_)`. It evaluates the receiver and parameters as normal but flags the invoked code (using the `call-qualified` meta-function) as ancestor code.

As an illustration consider constructing the object with scope `s3` from Figure 11. The context of rule `(2)` of Figure 10a

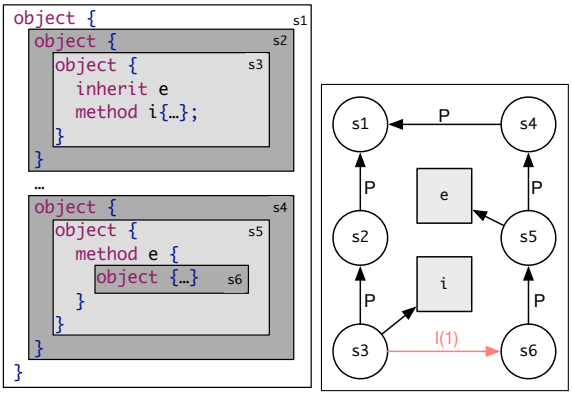


Figure 11. Fragment of program (left) and scope graph fragment after construction of `s3` (right).

when applied to object `s3` is `S=s2` and `EB=E()`. It first allocates object `s3` linking `s2` as its enclosing object, and then evaluates its ancestor expression. Rule `(3)` evaluates the ancestor expression `e` in `B(s3)` mode eventually entering rule `(2)` for the object with scope `s6`. At this second invocation of rule `(2)` the context is `S=s5` and `EB=B(s3)`. Had `s6` itself had an ancestor expression, rule `(2)` would evaluate it in a context where `self` is the hierarchy root (`S=s3`) and `outer` is the object surrounding the object expression (`O=s5`).

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. This allows us to model them outside of the name resolution algorithm. We choose to define their behavior with traditional environment-passing semantics with closures. The environment maps local variable names to locations in a separate variable store. The store threads through the semantics.

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.

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.

```
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.

```
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 syntactic placeholder for a rule which halts evaluation. Arrow $\xrightarrow{\text{read}}$ access an object in the store.

```
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
  S --outer--> 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 14 shows the semantics of name resolution in inherited scope. Rules over arrow $\xrightarrow{\texttt{res-inh}}$ resolve a reference `x` in the inherited scope `S'` subject to aliases (`als`) and exclusions (`exs`). The inherited scope is abandoned if the method was excluded (rule `(1)`). If the method is an alias, rule `(2)` starts a new resolution from the alias declaration to the target declaration. As an optimization it concatenates the two resolution paths.

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 $\xrightarrow{\texttt{disamb}}$ to choose the

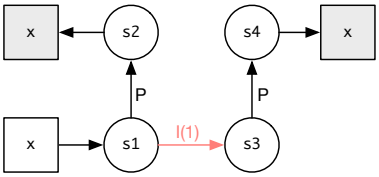


Figure 16. Scope graph with multiple resolution paths for a reference.

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

Looking up a declaration entails following a resolution path. The algorithm simply walks the resolution path to return a declaration and its scope. It is noteworthy that the returned scope is (1) always the receiver object in the case of a qualified call and (2) always the `self` of a hierarchy of objects. A definition of $\xrightarrow{\texttt{lookup}}$ semantics can be found in the extended version [33].

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 17 shows the local variable policy for Grace. It specifies whether a local variable name is legal in certain conditions. Rule `(1)` of Figure 10a applies this policy after it has constructed an object.

Figure 17 illustrates the rejection mechanism for a local variable that shadows a method. The rules decide whether to reject local variable `x` using the name resolution path `p`. A local path ($\texttt{p} \xrightarrow{\texttt{local}} \textbf{true}$) indicates that `x` shadows a local method. A strictly lexical path ($\texttt{p} \xrightarrow{\texttt{s-lex}} \textbf{true}$) indicates that `x` shadows a method in enclosing scope.

Method naming policy. We create a similar policy for method names, as shown in Figure 18. Rule `(2)` of Figure 10b enforces this policy before recording a method declaration.

Figure 18 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

```
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 17. 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 18. Method name policy and example of enforcement semantics.

enables this behavior by setting `member-allow-shadow-lex()` to **`false`**.

Confidential access. 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 $\xrightarrow{\text{lookup}}$. Given a reference `x` in scope s_{ref}, its resolution path `p`, declaration scope s_{dec} and visibility annotation `a`, we take an access decision as follows. If `a` is `confidential` then access is granted if (1) the declaration is local ($\texttt{p} \xrightarrow{\text{local}} \textbf{true}$) or (2) if the declaration is reachable through lexical scope ($\texttt{p} \xrightarrow{\text{lex}} \textbf{true}$). A method alias is regarded `confidential`. If `a` is `public` and the method is not an alias then access is permitted.

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. On average each test covers 66% of semantic rules and 65% of rule premises. The most comprehensive test covers 78% of rules and 80% of premises. We use these tests to validate that our executable specification has the same behavior as the mainstream Grace implementation.

There is an ongoing discussion within the Grace community about how dynamic Grace should be. Proponents of static name resolution aim to limit the freedom of inheritance expressions to allow static checking. Proponents of more dynamic behavior enforce no restrictions on inheritance expressions. Various implementations with various degrees of dynamism therefore exist. We have chosen to model the most dynamic and generic behavior possible by not enforcing any restrictions on the inheritance expressions. Our aim is to provide a specification which acts as a baseline for experimentation and validation. Principled restrictions can be applied to the specification to limit the dynamism of the language.

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. Name resolution from outside of an object with n ancestors has an $O(n)$ complexity. Name resolution from within an object with n ancestors, m enclosing objects each with n ancestors has an

$O(m*n)$ complexity. For an object with m methods each with v variables, checking local variable names requires $m * v$ name resolutions. Enforcing the method name policy requires m name resolutions. Although name resolution if the semantics does not flatten objects is expensive, it gives a principled definition which serves as a baseline for principled optimizations.

6 Related Work

Object oriented languages. Grace is inspired by Smalltalk, Self, Newspeak and Java. Smalltalk [14] is a dynamic object oriented language with single inheritance and no nested class declarations. Names are resolved upwards in the inheritance chain. Self [31] is the first programming language with prototype inheritance. Self unifies the representation of attributes, methods and local variables. Resolution of methods is indistinguishable from resolution of local variables. Newspeak [7] has nested classes and single inheritance identified by expressions. Each object maintains a link to its outer object. Method references can reach declarations in lexical range; inheritance chains of outer objects are never searched. Self has no explicit `outer` pseudo-variable. Java [15] has classes with single inheritance. Classes may be nested arbitrarily deep and references are lexically resolved. Name resolution is static. Similarly to Grace, Python allows arbitrary class inheritance expressions.

Formal definitions of object-oriented languages Wolczko [37] gives a denotational semantics for a significant subset of Smalltalk. The semantics does not distinguish resolution from lookup and does not enforce name policies. The operational semantics of Mäki-Turja et al. [25] separates resolution from lookup. K-Java [5] is a complete specification of Java 1.4. Name resolution is static and emits equivalent programs with fully qualified names. Maffeis et al. [23] give an operational semantics of JavaScript [10]. Objects are not flattened, both inheritance and scope nesting are encoded as object hierarchies. λ_{JS} [16] is a core of JavaScript with a reduction semantics. JavaScript programs are compiled to λ_{JS}. The compilation phase changes name binding to allow for a simpler definition. Politz et al. [28] take a similar approach for Python and give reduction semantics for a core language λ_{π}.

Dynamic semantics formalisms. DynSem is most related to I-MSOS [24], where it borrows the concepts of semantic components and implicit propagation from. Typical DynSem specifications give semantics in a big-step style [19]. Rules resemble SOS [27] rules if meta-functions and implicit reductions are used.

Funcons [9] is a semantics formalism aiming to provide a definitive collection of reusable semantics. Redex [13] embeds meta-notation for Felleisen-Hieb reduction rules. K [30] is a semantics formalism that has been applied to production-sized languages (C [11] and Java [5]). XSemantics [3] is a DSL for the specification of type systems of languages implemented in Xtext [2], but it can also be applied to the implementation of interpreters. XSemantics allows only a single environment to be used per rule, which must be explicitly propagated.

Name Resolution The use of scope graphs [26] for name resolution results from a formalization of the declarative NaBL name binding language [22]. The distinguishing feature of NaBL and scope graphs with respect to other name binding approaches (see [26] for a discussion of related work) is the support for *imports*. In this paper, we have used scope graphs as conceptual model to explain name resolution in Grace. In the implementation we use a custom object model instead of frames derived from scopes. In future work, we would like to derive the object model automatically from the scope graph model according to the scopes and frames approach [29].

Language workbenches. We used the Spoofax language workbench [20, 35]. Spoofax provides a meta-DSL for each aspect of a language definition. Spoofax is part of the family of language workbenches. The Rascal [21] 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. The Redex [13] approach is to extend a single language with meta-notation for the various aspects of a definition. Specifications are given as reduction semantics. Erdweg et al. [12] thoroughly compare language workbenches.

7 Conclusion and Future Work

We 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 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 showed 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 envision the specification as a baseline for discussion and experimentation around Grace's semantics. It can be used in the Grace community to experiment with various flavors of Grace as well as to compare existing implementations. Principled optimizations, such as flattening of object hierarchies, can be specified on top of the specification. By comparing the behavior of the derived interpreter with that of an existing implementation it is possible to discover differences with or bugs in the existing 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 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.

Acknowledgments

We would like to thank the Grace community, and in particular Andrew Black, Kim Bruce, James Noble, and Michael Homer, for their help with our understanding of Grace and feedback on this paper. We would like to thank Peter Mosses and the anonymous reviewers for their comments on this paper. This research was supported by a gift from the Oracle Corporation and by the NWO VICI *Language Designer's Workbench* project (639.023.206).

References

[1] Hubert Baumeister, Harald Ganzinger, Georg Heeg, and Michael Rüger. 1987. Smalltalk-80. *it* 29, 4 (1987), 241–251.

[2] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend* (2nd ed.). Packt Publishing.

[3] Lorenzo Bettini. 2016. Implementing type systems for the IDE with Xsemantics. *jlp* 85, 5 (2016), 655–680.

[4] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. 2012. Grace: the absence of (inessential) difficulty. In *OOPSLA*. 85–98.

[5] Denis Bogdanas and Grigore Rosu. 2015. K-Java: A Complete Semantics of Java. In *POPL*. 445–456.

[6] 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)*.

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

[8] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. 2006. Program Transformation with Scoped Dynamic Rewrite Rules. *FUIN* 69, 1-2 (2006), 123–178.

[9] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. 2015. Reusable Components of Semantic Specifications. *TAOSD* 12 (2015), 132–179.

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

[11] Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. In *POPL*. 533–544.

[12] 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. *Comp. Lang., Syst. & Struct.* 44 (2015), 24–47.

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

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

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

[16] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP*. 126–150.

[17] 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).

[18] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. 2012. Patterns as objects in grace. In *DLS*. 17–28.

[19] Gilles Kahn. 1987. Natural Semantics. In *STACS*. 22–39.

[20] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*. 444–463.

[21] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM*. 168–177.

[22] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. 2012. Declarative Name Binding and Scope Rules. In *SLE*. 311–331.

[23] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *aplas*. 307–325.

[24] Peter D. Mosses and Mark J. New. 2009. Implicit Propagation in Structural Operational Semantics. *ENTCS* 229, 4 (2009), 49–66.

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

[26] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *ESOP*. 205–231.

[27] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *jlp* 60-61 (2004), 17–139.

[28] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: the full monty. In *OOPSLA*. 217–232.

[29] 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 *ECOOP*.

[30] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *jlp* 79, 6 (2010), 397–434.

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

[32] 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 *PEPM*. 49–60.

[33] 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.

[34] Vlad A. Vergu, Pierre Néron, and Eelco Visser. 2015. DynSem: A DSL for Dynamic Semantics Specification. In *RTA*. 365–378.

[35] 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 *OOPSLA*. 95–111.

[36] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. 2012. Declarative specification of template-based textual editors. In *LDTA*. 1–7.

[37] Mario Wolczko. 1987. Semantics of Smalltalk-80. In *ECOOPW*. 108–120.