EGO: Controlling the Power of Simplicity

Andi Bejleri
Università di Pisa
Lugarno Pacinotti
43 - 56126 Pisa, Italy
bejleri @ cli.di.unipi.it

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, PA 15213, USA
jonathan.aldrich @ cs.cmu.edu

Kevin Bierhoff
Carnegie Mellon University
Pittsburgh, PA 15213, USA
kevin.bierhoff @ cs.cmu.edu

Abstract

The SELF programming language provides powerful dynamic features, allowing programmers to add and remove methods from objects and to change the inheritance hierarchy at run time. These facilities are useful for modeling objects that behave in different ways at different points in the object's lifecycle. Unstructured use of these techniques, however, can result in arbitrary changes to the interface of the object, and thus is incompatible with statically type-checked languages such as C++, C# and Java.

This paper proposes a structural type system for tracking changes to the interface of an object as methods are added and removed, and inheritance is changed at run time. The type system tracks the linearity of object and method references in order to ensure that objects whose interfaces change are not aliased. We show how our type system can express and enforce interesting protocol specifications. We then define a formal model of the language and type system, and prove that the type system is sound. Thus, our system is a foundation for languages that combine much of the power of dynamic languages like SELF with the benefits of static typechecking.

Keywords Prototype-based languages, dynamic inheritance, Self, Type safety

1. Introduction

Objects, by their nature, often have different behavior in different stages of their lifecycle. SELF [25] is a prototype-based object-oriented language that allows programmers to dynamically change the inheritance hierarchy and the set of methods that each object understands. Thus SELF objects can have different behavior at different points in program execution. This model is appealing for implementing a large variety of software systems (see figure 1).

SELF is dynamically typed, allowing arbitrary changes to objects. Unfortunately, this leads to runtime errors when objects receive messages they don't understand. Static typing can eliminate these errors at compile-time. However, this traditionally comes at a cost: In existing statically typed object-oriented languages the class of an object and the messages it understands are fixed at object creation time and cannot be changed later. We propose a new language, EGO, that has the power of SELF to change the behavior of objects but controls this power with a static type system. The following two subsections explain in more detail what changes to objects are allowed in SELF and how EGO guarantees the validity of those changes.

```
| socket <− () | "a new empty object"
socket AddSlots: (|bind = (< code > ...
  socket AddSlots: (| port <− Nil |)
        "adding a new data slot"
  socket AddSlots: (|listen = (< code > ...
    socket AddSlots: (|accept = (< code > ...
      socket AddSlots: (|read = (< code >)|)
      socket AddSlots: (|write = (| : data...| < code >)|)
      socket AddSlots: (|close = (< code >)|)
)|))|))|)
```

Figure 1. TCP socket example illustrating the expressiveness of SELF for ensuring that method protocols are respected

1.1 Expressing Method Protocols in SELF

SELF's dynamism can be used to express constraints on the ordering of calls to an object's methods. For example, Figure 1 demonstrates how a Berkeley TCP socket might be implemented in SELF; the *addSlots:* message is used to add new methods to the socket object. A constraint on the design of sockets is that methods must be called in a particular order: first *bind*, then *listen*, then *accept*, and finally any series of *read* and *write* before calling *close*. In SELF, we can ensure that *listen* is not called before *bind* by not even adding the *listen* method to the socket object until the *bind* method has been called. Similarly, when the *listen* method is called, the *accept* method is added to the object, and when that is called, the *read*, *write*, and *close* methods are added. We can use the same technique for fields: the *port* field is not meaningful until *bind* is called, and so in SELF we can simply avoid adding it to the object until the *bind* method executes.

If a client of the socket object tries to invoke *read* before any of the earlier methods, the system will raise a message not understood error, since this method has not yet been defined for the socket object. Thus, adding (and removing, although this is not shown in the examples) methods to an object dynamically is an elegant way to ensure that methods are not called inappropriately, because the method simply does not exist.

In contrast, in a more conventional object-oriented language such as Java, clients could call methods in an arbitrary order. The developer of a Java socket library must either manually implement run-time checks that throw an exception if the methods are called in the wrong order, or risk corruption of the socket's internal data structures if clients invoke operations in the wrong order.

FOOL/WOOD '06 14 January 2006, Charleston, SC

Thus, compared to languages like Java, SELF's dynamic mechanisms can be used to express and enforce constraints on the ordering of method calls in an elegant way. However, because SELF is dynamically typed, a violation of these constraints will not be detected until the message not understood exception is raised at run time.

It is easier to identify the cause of this error than it would be if the method call succeeded but corrupted the socket's data structures (as might happen without the use of dynamic method addition), but nevertheless it would be nice to detect the possibility of the error statically. Static detection of errors is challenging, however, due to the changes in an object's interface when the set of methods in an object is modified, and due to the possibility that there might be aliases to the object being modified. Because of the lack of static checking, the potential benefits of dynamic inheritance and method change at run time are underutilized in practice.

1.2 Contributions

The contribution of this paper is a type system that statically ensures that all accesses to object slots will succeed at runtime, even in the presence of method changes and dynamic inheritance. We formally define an imperative, object-oriented language, EGO, which is a core language modeled after Fisher et al. [15], with additional SELF-style primitives and typing restrictions sufficient to ensure type safety. In particular we control dynamic changes to aliased objects. We designed EGO in such a way that a static type checker can guarantee that a well typed program will lack "message not understood" errors at run time. The type safety proof for EGO directly implies this property. A consequence of type safety is that the technique of adding and removing methods to an object dynamically can be used to statically enforce message protocols in EGO.

The type system of EGO blends the features of several previous type systems in order to achieve soundness. For each object it keeps track of all methods a client can invoke. The type system distinguishes between linear (non-aliased) and non-linear (aliased) objects [17]. It statically ensures that linear variables are used at most once, and that linear functions are called at most once, while allowing aliasing of non-linear variables and multiple calls to non-linear functions.

The use of linearity in typing objects solves crucial aliasing and typing issues. Dynamic changes to the type of the object (e.g. by adding a method) are only permitted on linear objects. A new object has a linear type when it is created and the type system guarantees its linearity during the program unless the client explicitly makes it an aliased object (on which fewer changes are allowed).

To our knowledge, our system is the first sound, static type system that supports imperative method addition, removal, and dynamic changes of an object's inheritance. Previous systems have been limited to adding methods to an object (without supporting method removal), or determining an object's inheritance at run time (but not allowing it to be changed), or supporting only functional changes to an object's interface (where a new object is created and the original object is left unchanged, as opposed to our imperative object updates). Our system is also the first (of which we know) to integrate first-class linear functions into an object-oriented language.

Our system can be considered a foundation for research into more flexible typestate systems for objects [9, 10]. As a foundational system, it may not be as succinct or easy to use as a source-level language, but instead is designed to further understanding into the core mechanisms of typestate and to explore more flexible implementation strategies for typestate, such as dynamic changes to the methods and superclass of an object. Incorporating this additional flexibility into easy-to-use source-level languages is an important area of future work.

Organization. The remainder of this paper is organized as follows. Section 2 gives an intuitive presentation of EGO illustrated with a number of examples. Section 3 introduces the core language, its dynamic semantics, static semantics, and a brief presentation of the type safety proof. Section 4 summarizes related work, and the last section concludes.

2. Overview of EGO

This section gives an informal introduction to our language. After giving a brief intuition of its constructs, we show how to encode some common object-oriented programming idioms. We then discuss how EGO tackles the important problem of aliasing. That forms the basis for a detailed description of how methods are defined. Finally we demonstrate EGO's expressive power with a number of examples. Throughout the section we highlight the challenges that static typechecking must confront.

2.1 Language Intuition

A program in EGO is a pair of an expression and a mutable store. An expression can be anything from a simple value to a complex object manipulation. Some kinds of expressions can contain other nested expressions. The store keeps track of the current objects in the system, and allows us to model imperative updates to objects. We use lambda abstractions to define a function and bind a variable in its body expression. Moreover, we use the notation of Fisher et al.'s calculus [15] and introduce also a number of primitives for object manipulation that are inspired by the work on SELF [25].

- *clone* duplicates an object.
- *delegate* imperatively changes the super field of an object, thus determining from whom the object inherits.
- $\leftrightarrow$ imperatively adds a method to an object (or changes the implementation of an existing method).
- *change_linearity* is a technical primitive used for dealing with aliasing, as we shall see later.
- Finally $e \Leftarrow m$ invokes a method on an object.

The first four primitives yield the object created or manipulated to be used in the surrounding expression. The last one is used for method calls and thus yields the body of that method.

In the following sections we will develop a number of examples that show these primitives in action.

2.2 Elementary Programming Idioms

EGO is designed as a core language for expressing dynamic inheritance and method addition. We can define a number of derived forms for well-known and convenient idioms that will help us write more concise programs. That will also help us in presenting more advanced examples in the remainder of the section.

This section focuses on the notions of a *let* construct and instance fields for objects. We will also discuss how to create new objects and how to use them like traits in SELF (or equivalently, like classes in languages like Java).

The *let* variable binding construct can be simulated in the standard way, using a simple lambda expression as reflected in the following definition. It also allows us to define sequences of operations.

$$let\ x : \tau = e_1\ in\ e_2 \overset{\text{def}}{=} (\lambda x{:}\tau.e_2)\ e_1$$

$$e_1; e_2 \overset{\text{def}}{=} let\ \_ = e_1\ in\ e_2$$

We model instance fields as methods which take a self parameter but no others. Defining a field would look like the following:

$$e_1.f := e_2 \overset{\text{def}}{=} let\ x = e_2\ in\ (e_1 \leftarrow\!+ f = \lambda self{:}\tau.x)$$

This will also work for reassigning a field value. In this case, $\leftarrow\!+$ will just redefine the method body. Note that e_1 has to be an object and we use a let binding to evaluate e_2 to a value before the method body is created. Access of a field then becomes invoking a parameterless method (with $e \Leftarrow f$, where e is an object and f the name of a field).

In fact we can use the above derived form to add or change an arbitrary method on an object: If e_2 is itself a lambda expression then it simply defines a method body that relies on additional arguments as well as $self$. (We will discuss method definitions in detail below.)

How do we get an object in the first place? EGO is a prototype-based language that allows us to *clone* existing objects. We assume that a well-known variable *Object* is bound to the first object in the system. Thus creating a new object, adding two methods, and invoking the first one can be realized as follows.

$$clone(Object) \leftarrow\!+ m_1 = e_1\ \leftarrow\!+ m_2 = e_2 >\Leftarrow m_1$$

Expressions for a method body have to evaluate to a lambda abstraction with argument *self*. When a method is executed, the receiver object will be applied to this outermost lambda. Methods can refer to their receiver and its (other) methods by accessing the bound variable *self*.

We often want to use an object in a class-like manner, meaning that the object contains instance methods to be used by other objects. Such an object is called a *trait* in the SELF literature [25]. We can use the *let* construct in combination with delegation to realize traits as shown below.

```
$let\ Trait = clone(Object)$
        $\leftarrow\!+ succ = \lambda self{:}\tau.self \Leftarrow f + 1\ \ in$
   $(clone(Object).delegate(Trait)$
        $\leftarrow\!+ f = \lambda self{:}\tau.5) \Leftarrow succ$
```

The result of this expression would be 6. Obviously, an arbitrary number of objects can be defined that inherit their behavior from the *Trait* object above by delegation and define their own f field. Another option is to simply clone the trait object, which would result in simply duplicating all of the methods of *Trait* rather than sharing them through delegation. We will present an example of this more prototype-oriented approach in a later section.

2.3 The Challenge of Aliasing

So far we have ignored a major complication of our system: aliasing. An aliased object is (possibly) referred to by multiple names (references) in a program as opposed to *linear* objects that have only one name. Aliased objects are also called "non-linear", and linear ones are sometimes called "non-aliased".

In an object-oriented setting, aliasing is almost inevitable because of the state held in instance fields. A very common notion is that a server object s is used by multiple clients c_i that all hold a reference to s in their fields $c_i.r$. The object s is then heavily aliased as in the following definition.

```
// trait for s
$let\ b = change\_linearity(\ clone(Object)$
          $\leftarrow\!+ service = \lambda self{:}\tau_b.\ \lambda x{:}Nat.\ x + 1\ )\ in$
// now define s itself
$let\ s = change\_linearity(clone(Object).delegate(b))\ in$
// and finally the clients
  $let\ c1 = clone(Object) \leftarrow\!+ r = s\ \ in$
  $let\ c2 = clone(Object) \leftarrow\!+ r = s\ \ in$
  ⋮
    // invalid: let _ = s.delegate(a) in
    $c2 \Leftarrow r \Leftarrow service(5)$
```

Figure 2. A server object s referenced by multiple clients

```
// trait for s
$let\ b = clone(Object)$
        $\leftarrow\!+ service = \lambda self{:}\tau_b.\ \lambda x{:}Nat.\ x + 1\ \ in$
// now define s itself
$let\ s = clone(Object).delegate(b)\ in$
// and finally the clients
  $let\ c1 = clone(Object) \leftarrow\!+ r = \lambda self{:}\tau_c.s\ \ in$
  $let\ c2 = clone(Object) \leftarrow\!+ r = \lambda self{:}\tau_c.s\ \ in$
    . . .
```

If we now change the configuration of s e.g. by changing its delegate from b to a with $s.delegate(a)$, obviously all clients are affected. In particular, it is hard to tell whether s will still work the way its current clients expect it to.

For this reason, we forbid a change of delegation for aliased objects as well as adding or changing methods for such objects if it changes the method's signature. We allow methods to be modified for aliased objects as long as the new method has the same signature as the old method. This allows us to model field updates, for example.

Moreover, we forbid delegation to a linear object (because that would be just like a second explicit reference to that object). Instead, we introduce the *change_linearity* primitive mentioned earlier to explicitly convert a linear into an aliased object that can then be a delegatee. Note that there is no way of turning an aliased object back into a linear one. Figure 2 shows how *change_linearity* must be added to the code above to typecheck properly in EGO.

Intuitively, these restrictions have to do with the typing of objects. Changing a method signature or the delegation changes the type of the object. That means that the aliases to that expression somehow would have to invisibly change their types as well, which would be difficult or impossible for a static type system to track in the general case. Conversely, changing a linear object affects only the type of the expression at hand, which is what a static type checker tracks anyway.

On the typing level we introduce a *linearity flag* for objects and lambdas, which we write as "¡" following Wadler [26]. *change_linearity* explicitly removes this flag for an object, thus allowing it to be aliased. Bodies of linear lambda abstractions have access to the linear variables defined in the current scope abstraction. The type system guarantees that such linear variables are used only once. (We say they are "consumed" on usage.) Figure 3 gives an example, with pairs written as (x, y). Non-linear lambda-abstractions, on the other hand, can only access the non-linear variables in the context. Non-linear variables can be used multiple times.

We call a method linear if it is written with a linear lambda abstraction. Linear methods are consumed upon invocation, i.e. they

```
$let\ lin = clone\,(Object)$
    $let\ o = clone\,(Object) \leftarrow\!\!+\ l =$ ¡$\lambda self{:}\tau.(self, lin)\ in$
    // lin is no longer available
        $let\ (o2, lin2) = o \Leftarrow l\ in$
            // instead we can now use lin2
            // o2 replaces o, but does not contain l any more
```

Figure 3. A linear method consuming a variable on the stack and its linear receiver

are effectively removed from the receiver object. This guarantees the linearity of the context variables: If we could call the linear method l from figure 3 twice, then we would gain two aliases to lin "through the back door". As recursive calls to the same linear method would have the same harmful effect, we have to remove a linear method from its object *before* that method's body is evaluated. Thus the method l in figure 3 is not only no longer available after l was evaluated, but l cannot invoke itself on $self$ again either.

We forbid cloning of objects with linear methods for the same reason: That would result into pairs of linear methods accessing the very same variable. However, the object can be linear (because it is completely duplicated), and the resulting clone is linear in any case. Thus all objects are linear in the beginning of their lifetime and can be converted into a non-linear object explicitly using *change_linearity* (but not back into a linear object).

An alternative to the solution of consuming linear methods upon invocation would be to consume the receiver as a whole. We consider this a bad choice: Only one method could be ever executed on a linear object.

Independent of the linearity of a lambda itself, its argument can be linear or non-linear. A linear lambda argument requires a linear object. The object applied to such a lambda is no longer available at the invocation site after that application (again, we say it is "consumed"). However, the lambda abstraction can return its argument to the caller as the method $o.l$ in figure 3 illustrates. o is no longer available after the last line, but it is passed back into $o2$.

2.4 Method Definition

In order to capture the dynamic manipulations of objects statically, EGO types objects with a recursive record type [1,14] containing an explicit list of all the methods the object defines together with a field for its delegate. A linear object containing an integer field as well as a linear method that takes an integer argument and yields an integer would be typed as follows. The object delegates to an empty object like *Object*. Note that in the body of the type below, t is bound recursively to the entire type expression.

$$t.\text{¡} < field : t \rightarrow int, linMeth : t \multimap int \multimap int; super :<>>$$

We use $\multimap$ for typing linear lambda abstractions and $\rightarrow$ for non-linear ones. Every method body definition must be an explicit lambda abstraction for *self*, the receiver object. The type of *self* essentially lists *all* methods expected to be defined for the receiver, when the method is called. Additional arguments can be captured with nested lambdas.

The requirement that *self* must be typed with a recursive record type is essentially not different from typing an object with a class name in e.g. Java: Since the methods in a Java class cannot be manipulated the class name can be used as a (shorter) synonym for a record type containing all methods defined for that class.

In fact, our system is much more flexible in that different methods of the same object can declare different receiver object types. This is useful to encode typestate-like examples; as the object's type changes over time due to method addition, method removal, and delegation changes, different methods in the object become applicable. Thus the programmer can enforce possible sequences of method invocations on the object, i.e. the object's *protocol*. Figure 4 gives an example of method definitions using typestate. Note that we give *typedefs* for several record types in the beginning to improve readability. They are not part of the core EGO language.

```
$typedef\ entry = t. < name : t{\rightarrow}string,$
        $number : t{\rightarrow}string; super :<>>$
$typedef\ default = t.$¡$ < prepareNew : t{\rightarrow}action,$
        $makeEditable : t{\rightarrow}entry{\rightarrow}action,$
        $confirmDelete : t{\rightarrow}entry{\rightarrow}action; super :<>>$
$typedef\ action = t.$¡$ < prepareNew : default{\rightarrow}t,$
        $makeEditable : default{\rightarrow}entry{\rightarrow}t,$
        $confirmDelete : default{\rightarrow}entry{\rightarrow}t,$
        $ok : default{\multimap}default; super :<>>$

$let\ Entry = clone(Object)$
        $\leftarrow\!\!+\ name = \lambda self{:}entry.$""
        $\leftarrow\!\!+\ phone = \lambda self{:}entry.$""  $in$

$let\ WebPhonebook = clone(Object)$
$\leftarrow\!\!+\ prepareNew = \lambda self : default.$
   $let\ curEntry = clone(Entry)\ in$
   $self \leftarrow\!\!+\ ok =$ ¡$\lambda self : default.$ /* save new entry */
$\leftarrow\!\!+\ makeEditable = \lambda self : default.$
   $\lambda curEntry : entry.$
   $self \leftarrow\!\!+\ ok =$ ¡$\lambda self : default.$ /* save edited entry */
$\leftarrow\!\!+\ confirmDelete = \lambda self : default.$
   $\lambda curEntry : entry.$
   $self \leftarrow\!\!+\ ok =$ ¡$\lambda self : default.$ /* delete selected entry */
```

Figure 4. Web phonebook business logic

We illustrate the business logic of a Web-based phonebook. Such applications are characterized by *two-phased actions*: First, the user indicates the type of action he wants the system to perform (e.g. create a new entry with *prepareNew*). The phonebook application will then present a form to enter the new contact information. The user can now complete the action by sending an *ok* message (or cancel, which we omit).

Our phonebook therefore has a *default* and an *action* state. We see that objects in the *default* state have three methods, while those in *action* have four. The methods applicable to the respective states can be easily identified by the types of their *self* variables. The triggers to switch from one state to the other are the business methods and *ok*, respectively.

The type system ensures that a method can only be called on a receiver that matches its expected receiver type exactly, *after the method itself has been removed in the case of linear methods*.[1] [2] Thus, in the *default* state, the business methods can be called because they expect a receiver of type *default*, but the *ok* method cannot be called because it is not even part of the *default* state.

In the action state, the three business methods are still part of the type, but they cannot be called because these non-linear methods expect an object in the *default* state and the receiver is in the *action*

[1] Removing the method from the type is necessary to ensure that linear methods cannot recursively call themselves. Recursive calls would break the invariant that no linear method is called more than once.

[2] We have chosen to make the type of the receiver reflect the type inside the method, rather than the type as seen by the caller (before the called linear method is removed from it). The other choice would be more intuitive from the client's perspective, but more confusing from the standpoint of the implementor.

state, which has the additional method *ok*. On the other hand, the *ok* method is linear, and it can be called in the *action* state because once you take the *ok* method out of the *action* type, you get the *default* type which is what the *ok* method expects.

Note that *ok* behaves differently depending on the action that is to be performed. Therefore each business method defines its own *ok* method.

Our system tracks the exact type, rather than a supertype, for linear objects, in order to make sure that changes to the object are legal with respect its complete current type. In particular, when changing delegation the type system has to determine the exact new record type of the object, which can only be done on the basis of the exact old type of the object and its new delegate. Otherwise, the object could define a method with a name also used in the new delegate but with a different return type. If that method were not listed in the object type (which could happen if we allowed subtyping for linear objects) then the system would expect the wrong return type (the one defined in the delegate object) from a later call to that method.

The restriction of exact type tracking could be relaxed for aliased objects. Here, subtyping could be introduced to accept objects with more methods than expected, because the object type cannot change in a way that would introduce the problem mentioned above. Even though subtyping is well defined for record types, we elide this extension from our formal core system to keep it as simple as possible.

2.5 Expressive Power

The examples we have seen so far were mostly intended to illustrate syntax and semantics of EGO. This section will present higher-level examples in order to demonstrate the expressiveness of the language. In fact, one was already given in the previous section (figure 4) to illustrate the application of EGO to typestates. We will see typestates again in the examples that follow. The final one will implement the TCP socket from the introduction in EGO.

The examples rely on dynamic inheritance and adding new methods to objects over time. They are therefore not directly expressible in languages with static inheritance like Java. They are expressible in SELF, but SELF would not be able to statically guarantee that the program evaluation will succeed at runtime. Our system does guarantee successful evaluation of the presented examples by virtue of the type safety proof presented later.

Throughout the examples we rely on the intuition of the reader to assume the semantics of certain objects to which we merely refer by name. Explicitly defining a sufficiently large library for interesting examples is outside the scope of this paper.

Consider the EGO program in figure 5. It models the workflow in a company between a manager, her secretary, and her designated worker. We first implement the secretary who can do some work. We also define a prototype worker who, no surprise, can also do some work. We define a concrete secretary as opposed to a prototype worker for purely pedagogical reasons. Both could be prototypes. Also note that we do not use the *trait* idiom known from SELF to generate a worker "class". Instead we define the worker prototype as an object to be cloned to create instances. We feel that this more closely resembles the real world where different workers are different autonomous individuals.

Finally we implement the manager who has fields for her secretary and her worker. By default, the manager forwards all the work she has to do to her worker. We do this simply by delegation. (That forces the complicated typing of *self* in the two *doWork* implementations.) The use of delegation here models delegation in a real company, where work is delegated from one to the other person. Slightly confusing might be the implication that our manager does

```
typedef minit =  /* initial manager */
  t.¡<sec : mwork → u. < doWork : msec → unit >,
     setWorker : (t−setWorker) ⊸ worker ⊸ mwork >
typedef worker = t. < doWork : mwork→unit,
     workerSick : mwork→msec >
typedef mwork =  /* manager delegating to worker */
  t.¡<sec : t → u. < doWork : msec → unit >,
     myworker : msec → worker;
     super: worker >
typedef msec =  /* manager delegating to secretary */
  t.¡<sec : mwork → u. < doWork : t → unit >,
     myworker : t → worker,
     workerRecover : (t−workerRecover) ⊸ mwork;
     super :< doWork : t→unit >>

let Secretary = change_linearity(clone(Object)
←+ doWork = λself:msec.λtask:τ. . . .) in

let WorkerProto = clone(Object)
←+ doWork = λself:mwork.λtask:τ. . . .
←+ workerSick = λself:mwork . self.delegate(self ⇐ sec)
  ←+ workerRecover = ¡λself:msec .
           self.delegate(self ⇐ myworker)  in

let Manager = clone(Object)
←+ sec = λself : mwork ⇐ Secretary
←+ setWorker = ¡λself : minit.
  ¡λ newworker:worker.
  (self ←+ myworker =
    λself:msec . newworker).
  delegate(newworker)
⇐ setWorker(change_linearity(clone(WorkerProto))) . . .
```

Figure 5. Using delegation to implement workflows

```
let PowerSupply = clone(Object).
←+ generatePower =
   λself:t. < generatePower : t → power > . . . . .

let On = change_linearity(clone(Object)
←+ getPower =
   λself:t.¡<supply, on, off; super :< getPower >> .
   self ⇐ supply ⇐ generatePower) in

let Off = change_linearity(clone(Object)) in

let PowerSwitch = clone(Object)
←+ on = λself:t.¡<supply, on, off; super :<>> .
   < self.delegate(On).
←+ off =
   λself:t.¡<supply, on, off; super :< getPower >> .
     self.delegate(Off) >   in

let ps = clone(PowerSwitch) ←+ supply =
   λself:t.¡<supply, on, off; super :< getPower >> .
     PowerSupply in
ps ⇐ on ⇐ getPower ⇐ getPower ⇐ off ⇐ on ⇐
getPower ⇐ off
```

Figure 6. A power network using composition and delegation. Certain types have been abbreviated.

```
$typedef\ open = t.¡ < port : t \to (t, int), read : t \to (t, \tau),$
   $write : t \to \tau \multimap t, close : t \multimap unit; super :<>>$
$typedef\ portt = t.¡ < port : open \to (open, int) >$

$let\ Socket = clone(Object)$
$\leftrightarrow bind = ¡\lambda self : t.¡ <> .$/* bind impl */;
   $self \leftrightarrow port = \lambda self : open.(self, prt)$
      $\leftrightarrow listen = ¡\lambda self : portt.$/* listen impl */;
         $self \leftrightarrow accept = ¡\lambda self : portt.$/* accept impl */;
            $self \leftrightarrow read = \lambda self : open.(self,$ /* result */)
               $\leftrightarrow write = \lambda self : open.¡\lambda data{:}\tau. \ldots ; self$
               $\leftrightarrow close = ¡\lambda self{:}open. \ldots ; unit$
```

Figure 7. A TCP socket object in EGO

not even “see” the work items she delegates to her subordinate. But maybe this is not too unrealistic, either.

Now imagine the worker gets sick. We would invoke the *workerSick* method on our manager. That causes the manager to dump her work onto her secretary from now on. The secretary cannot get sick, so that’s a safe guess. But also, the manager expects her worker to recover eventually. Thus she defines an additional method *workerRecover* to anticipate this event. Note that this changes the manager’s signature. She is now in a different state, the “worker sick” state. *workerRecover* is defined to be linear and thus will be consumed on invocation. The method will also redelegate to the now recovered worker, effectively transferring the manager back to her original state. As a final remark concerning states we point out that the manager is in a sort of initialization state before *setWorker* is called. Only then can she do (or rather, delegate) work.

Next we implement a power network in figure 6. It consists of a power supply, an on-off-switch and a client that requests power. In this example we use delegation to model the different states of the power switch (on and off). Obviously, only the *On* object has a *getPower* method that forwards the power request to the supply configured for that object. Thus our client first has to connect the switch to the supply by adding the *supply* field. Then it can switch on, get power for a while, switch off, and on again to get more power.

The *On* and *Off* objects that implement the two controller states can be aliased by an arbitrary number of switches that all delegate to one of these two objects. The power supply is unique to each switch (both being physical devices) and therefore represented as an instance field to the switch. Without that field defined, the switch is not functional as the signatures for the *on* and *off* methods do not match. It can redirect to a different source later, though.

The power network example uses an implementation strategy that is quite the opposite to the workflow example above. In the power network, we use delegation to express states (on and off) and explicit forwarding (similar to composition in object-oriented programming) to transfer the power from the supply to the consumer. In the workflow example, on the other hand, we added and removed methods to change the state of the manager object. We used delegation to (implicitly) forward calls from one object to the other.

Finally, we look into the TCP socket example from the introduction section again. Figure 7 gives an implementation in EGO. We do not use delegation at all but rather manipulate the object with each method call. The implementation relies on linear methods to enforce that *bind*, *listen*, and *accept* are called exactly once. Each of these generates the following method; therefore a client must follow the prescribed call sequence.

We show as an example how *bind* also generates a field that contains the port on which the socket is going to listen in order to demonstrate that a real socket implementation is a full-blown data structure. Derived fields, as the *port* here, can be added to the object when they are available in EGO, effectively preventing reads from not yet defined fields.

The call to *accept* will generate *read*, *write*, and *close* methods. The first two can now be called an arbitrary number of times. They require a linear *self* and return it unchanged upon completion of the call. *close* also requires a linear *self* but does not give it back, effectively making the object inaccessible. Lending [2] or borrowing [6] for the methods returning *self* would make this explicit return unnecessary. We elide this possible extension to EGO for simplicity.

2.6 Summary

In the preceding sections we gave an informal introduction to EGO. We have seen in detail how programs can be implemented in the language. We discussed its handling of aliasing as well as the notion of typestates which it naturally supports through its method definitions. Finally we could express a number of relevant examples in EGO. We saw that delegation and dynamic method changes are somewhat interchangeable, effectively allowing different programming styles.

The examples were complex enough to imagine that an ad-hoc SELF programmer can introduce bugs that result in runtime errors. That motivates the need for static typechecking for such programs in order to make sure that all object manipulations and method invocations will succeed. Throughout this section we described the restrictions EGO imposes on the programmer to control SELF’s “power of simplicity”. We have seen that they are loose enough to implement interesting programs in EGO, and although the current type system is somewhat complex we believe this can be simplified considerably in a practical system. It is the main result of this paper that these restrictions are also strong enough to ensure EGO’s type safety. This will be formalized in the next section.

3. Formal Model

We now introduce the core EGO language to formalize the intuitions given above. This section contains the full dynamic semantics, the full static semantics, and a summarized type safety proof of EGO. The full type safety proof is available in [4].

3.1 Syntax

Figure 8 presents the syntax of our model. We do not include base types, control flow structures, exceptions, and subtyping into the model as they are well-known from the literature. We omit multiple inheritance and polymorphism as these are orthogonal to the typing issues at hand. Note that an overbar is used to represent a sequence.

An expression is a variable **(x)**, a value **(v)**, a clone of an object **(clone)**, a method invocation **(⇐)**, an object delegation change **(delegate)**, the addition of a new method to an object or the change of a method body (↔), a function application **(f a)** and a change of the type linearity of an object **(change_linearity)**. A method **(M)** is defined as a pair: the name of the method **(m)** and an expression that reduces to a method body.

A method body definition is a lambda expression with a linearity for the function and an explicit type for arguments (type inference is future work). We require that the outermost lambda types the receiver object. Our store **(S)** is a set of pairs: the location of the object and the object descriptor **(Odescr)**.

An object descriptor is a pair: the location of the super object, and a sequence of methods defined for this object. While our syntax for methods follows that of functional languages in order to connect more directly to previous linear type systems [26], our record-based object encoding is similar to standard object encodings [1,5,14,15].

The primary difference in our encoding is that we must represent inheritance explicitly–since it might be changed–rather than just merging inherited methods into the object itself as previous systems have done.

There are four kinds of types: for variables **(t)**, for non-linear functions **($\longrightarrow$)**, for linear functions **($\multimap$)** and finally for objects **(t.R)**. The object type is a recursive type where t is bound to R. The record type **(R)** is a list of the types of the methods **(B)** defined for the object and the type of the super object **(super)**. In [15] a row type is also a row variable, a row lambda abstraction and a row application. We omit these row types as we don't need them in our system to express any kind of row polymorphism. The type of a linear object is presented by **¡**. We use **[¡]** to represent that the object might be linear or non-linear. Optional syntax is enclosed in [ ].

Instance variables are represented by parameterless methods. Locations L are not part of the source code. We assume to have a first object **(Object)** defined when we want to evaluate a program.

$$
\begin{array}{lrcl}
\text{(expressions)} & e & ::= & x \mid v \mid e \Leftarrow m \mid e_1.delegate(e_2) \\
 & & & \mid clone(e) \mid e \leftarrow\!\!+\, m = e \\
 & & & \mid e_1\ e_2 \\
 & & & \mid change\_linearity(e) \\
\text{(values)} & v & ::= & L \mid [\text{¡}]\lambda x{:}\tau\ .\ e_0 \\
\text{(heap)} & S & ::= & Object \mapsto super : Object \\
 & & \mid & L \mapsto Odescr, S \\
\text{(object desc)} & Odescr & ::= & super : L \\
 & & \mid & Odescr \leftarrow\!\!+\, M \\
\text{(method desc)} & M & ::= & m = e \\
\text{(types)} & \tau & ::= & t \mid \tau' \rightarrow \tau'' \mid t.R \mid \tau' \multimap \tau'' \\
\text{(records)} & R & ::= & [\text{¡}]<> \mid [\text{¡}]< B; super : \tau > \\
 & B & ::= & \epsilon \mid m : \tau, B \\
\text{(heap location)} & L & & \\
\text{(variable)} & x & & \\
\text{(type variable)} & t & & \\
\text{(method name)} & m & &
\end{array}
$$

Figure 8. Syntax of the language, store, types.

3.2 Dynamic Semantics

The dynamic semantics we defined for EGO is a standard small step operational semantics. The store **(S)** is a function from locations **(L)** to object descriptors **(Odescr)**. Figure 9 summarizes the rules for evaluating expressions. We describe each rule in turn.

$(R - Appl)$ shows how a method is applied to its arguments. We write $[v/x]e_0$ for the result of replacing x by v in expressions e_0.

$(R - LInvk)$ invokes a linear method on an object. The method is owned by the receiver and is linear. As the type system does not allow another call to that linear method we remove it from the store. The location **L** is passed as an argument to the method because **self** is not a free variable in the lambda expression. The type system does not support this in order to not have aliasing issues. The result of the reduction is a method apply with **L** as an argument and a store without the method **m** in it.

$(R - NInvk)$ invokes a non-linear method. The result of the reduction is the same as the one above except that the store is unchanged now: The type system allows the client to invoke a non-linear method more than once .

$(R - Clone)$ creates a new object from an existing one. The list of methods and the address of the super object are copied from the cloned object to the newly created location.

$(R - Deleg)$ changes the reference to the super object of the receiver object. The result of the reduction is the modified location of the receiver. Here the overbar represents a sequence of method bindings $\overline{M}$.

$(R - AddM)$ adds a new method to the receiver object. **dom($\overline{M}$)** represents the set of the methods name for an object descriptor. The result returned is the modified location of the receiver.

$(R - ChanMBd)$ changes the body of method **(m)** of the receiver.

$(R - ChanLin)$ does not effect the memory. It changes the linearity of an object from linear to non-linear. The result of the reduction is the location passed as argument for the expression.

$$\frac{}{([\text{¡}]\lambda x : \tau'.e_0)v, S \longrightarrow [v/x]e_0, S}\ \text{R -- Appl}$$

$$\frac{mbody(S[L], m) = v \quad v = \lambda x...}{L \Leftarrow m, S \longrightarrow v\ L, S}\ \text{R -- NInvk}$$

$$\frac{\begin{array}{c} mbody(S[L], m) = v \quad v = \text{¡}\lambda x... \\ S' = S[L \rightarrow (S[L] \setminus m = v)] \end{array}}{L \Leftarrow m, S \longrightarrow v\ L, S'}\ \text{R -- LInvk}$$

$$\frac{\begin{array}{c} S[L] = Odescr \quad L'' \notin domain(S) \\ S' = S[L'' \rightarrow Odescr] \end{array}}{clone(L), S \longrightarrow L'', S'}\ \text{R -- Clone}$$

$$\frac{\begin{array}{c} S[L_1] = super : L_1 \leftarrow\!\!+\, \overline{M} \\ S' = S[L_1 \mapsto super : L_2 \leftarrow\!\!+\, \overline{M}] \end{array}}{L_1.delegate(L_2), S \longrightarrow L_1, S'}\ \text{R -- Deleg}$$

$$\frac{\begin{array}{c} S[L] = super : L' \leftarrow\!\!+\, \overline{M} \quad m \notin dom(\overline{M}) \\ S' = S[L \rightarrow super : L' \leftarrow\!\!+\, \overline{M} \leftarrow\!\!+\, m = v] \end{array}}{L \leftarrow\!\!+\, m = v, S \longrightarrow L, S'}\ \text{R -- AddM}$$

$$\frac{\begin{array}{c} S[L] = super{:}L' \leftarrow\!\!+\, m_1{:}v_1... \leftarrow\!\!+\, m_i{:}v... \\ S'{=}S[L{\rightarrow}super{:}L' \leftarrow\!\!+\, m_1{:}v_1... \leftarrow\!\!+\, m_i{:}v'...] \end{array}}{L \leftarrow\!\!+\, m_i = v', S \longrightarrow L, S'}\ \text{R -- ChanMBd}$$

$$\frac{}{change\_linearity(L), S \longrightarrow L, S}\ \text{R -- ChanLin}$$

Figure 9. Evaluation rules for expressions

3.3 Static Semantics

Figure 11 presents the typing rules for expressions. Every typing rule has the standard form, $\Sigma; A \vdash e : \tau \Longrightarrow list_e$ that contains a store type **(Σ)**, an assumption list **(A or A')**, an expression that is typed **(e)**, the type of the expression **(τ)**and the list of linear objects **(list$_e$)** that are used to type the expression.

We use a type store Σ to store the types of our objects:

$$\Sigma ::= Object : t. <> \quad \mid \Sigma;\ L{:}\tau$$

The assumption list A (or A') contains the types of the bound variables in the expression **e** that is typechecked. An assumption list, A, is defined as:

$$A ::= \cdot \mid A, x : \tau$$

We use $\cdot$ to present the empty assumption list. An assumption list is non-linear if each assumption x$_i$:τ_i in it has a non-linear type τ_i.

$$\frac{\overline{M}[m] = v}{mbody(super : L \leftrightarrow \overline{M}, m) = v}$$

$$\frac{m \notin dom(\overline{M})}{mbody(super : L \leftrightarrow \overline{M}, m) = mbody(S[L], m)}$$

Figure 10. Rules for lookup of methods body.

Note that linear variables will be removed from the assumption list upon usage.

The type expression $t.[¡]{<}m_1{:}\tau_1, ..., m_k{:}\tau_k;$ $super : t'.[¡]{<}m'_1{:}\tau'_1, ..., m'_j{:}\tau'_j >>$ is a type t with the property that when we invoke a method m_i for $1 \le i \le k$ or a method m'_i for $1 \le i \le j$ to any element x of this type, like $x.m_i$, the result has type τ_i or τ'_i with t substituted for $t.R$. Thus $t.R$ is a form of recursively-defined type.

Let us describe each rule and give a brief justification with examples for selected cases. Note that the word location is used somewhat ambiguously because sometimes it refers to the label of a location and sometimes it is used to refer to the object at that location. However, what is meant is always obvious from the context.

$(T - Loc)$, $(T - NLoc)$ A location is well-typed if it is defined in Σ. The list returned is empty if the location is non-linear or **L** if the location is linear.

$(T - Method)$ A non-linear method is well-typed if its body is well-typed. The restriction on the assumption list to be non linear is because we want each variable needed to type the expression to be non-linear so we can safely call the method more than once. There is no restriction on the arguments of the methods because if they are linear there is no way of duplicating them. The returned list is empty as the objects used here are all non-linear.

$(T - LMethod)$ A linear method, too, is well-typed if its body is well-typed. However in this rule, there is no restriction on the assumption list. Linear variables can safely be used in a linear method because it will be called only once during the program. The list returned is the one returned from the type rule applied to the method body.

$(T - Var)$ The type of a variable is the one that it has in the assumption list. The returned list is empty as there are no objects used to type it. The assumption list has only the record to type the variable. The type system does not need to forget information in the assumption list during the typing of an expression.

$(T - Kill)$ This rule is used in the case we have to delete a record from the assumption list in order to typecheck an expression. We need it in typing cases like $¡\lambda x{:}X\lambda y{:}Y.x$ where y can be non-linear. As long as it is not used, $(T - Kill)$ can remove it from the context to type this linear method. The list returned is the same as the expression that is typed with the new assumption list.

$(T - Copy)$ This rule makes another copy of a non linear variable in the assumption list. We use it in cases like $\lambda x{:}Nat \;.\; x{+}x$ or $\lambda x{:}X \;.\; (x \leftrightarrow m = \lambda y{:}Y \;.\; x)$ where x is non-linear. The type system has to explicitly duplicate x in order to use it multiple times.

$(T - Clone)$ A $clone$ expression is well-typed if **e** (the prototype object) is well-typed and the super object of **e** has a non-linear type (which is true automatically by virtue of $(T - Deleg)$. The methods defined for the cloned object must all be non-linear in order not to copy references to linear objects through the back door. The new object created has a linear type.

$$\frac{\Sigma(L) = t.¡< B; super : t'.R >}{\Sigma; \cdot \vdash L : t.¡< B; super : t'.R > \Longrightarrow \{L\}}\;\text{T − Loc}$$

$$\frac{\Sigma(L) = t. < B; super : t'.R >}{\Sigma; \cdot \vdash L : t. < B; super : t'.R > \Longrightarrow \{\}}\;\text{T − NLoc}$$

$$\frac{\begin{array}{c}\Sigma; A, x : \tau' \vdash e_0 : \tau'' \Longrightarrow \{\}\\ x \notin A, nonlinear\ A\end{array}}{\Sigma; A \vdash (\lambda x : \tau'.e_0) : \tau' \rightarrow \tau'' \Longrightarrow \{\}}\;\text{T−Method}$$

$$\frac{\Sigma; A, x : \tau' \vdash e_0 : \tau'' \Longrightarrow list_{e_0} \quad x \notin A}{\Sigma; A \vdash (¡\lambda x : \tau'.e_0) : \tau' \multimap \tau'' \Longrightarrow list_{e_0}}\;\text{T − LMethod}$$

$$\frac{}{\Sigma; x : \tau \vdash x : \tau \Longrightarrow \{\}}\;\text{T − Var}$$

$$\frac{\Sigma; A \vdash u : U \Longrightarrow list_u}{\Sigma; A, x : X \vdash u : U \Longrightarrow list_u}\;\text{T − Kill}$$

$$\frac{\Sigma; A, x : X, x : X \vdash u : U \Longrightarrow list_u}{\Sigma; A, x : X \vdash u : U \Longrightarrow list_u}\;\text{T − Copy (nonlinear X)}$$

$$\frac{\begin{array}{c}\Sigma; A \vdash e : t.[¡] < B; super : \tau >> \Longrightarrow list_e\\ \forall m{:}\tau' \in B \;.\; \tau' = \tau''{\rightarrow}\tau'''\end{array}}{\Sigma; A \vdash clone(e) : t.¡< B; super : \tau >> \Longrightarrow list_e}\;\text{T − Clone}$$

$$\frac{\begin{array}{c}\Sigma; A \vdash e : \tau' \Longrightarrow list_e\\ mtype(m, \tau', \tau') = \tau' \rightarrow \tau''\\ \tau' = t. < B; super : \tau >\end{array}}{\Sigma; A \vdash e \Leftarrow m : [t.{<}B; super{:}\tau{>}/t]\tau'' \Longrightarrow list_e}\;\text{T−Invk}$$

$$\frac{\begin{array}{c}\Sigma; A \vdash e : t.¡< ..., m{:}\tau'[\rightarrow / \multimap]\tau''; super{:}\tau >\\ \Longrightarrow list_e\\ \tau' = t.¡< ..., [m{:}\tau'{\rightarrow}\tau''/ -]; super : \tau >\end{array}}{\begin{array}{c}\Sigma; A \vdash e \Leftarrow m : [t.¡< ..., [m{:}\tau'{\rightarrow}\tau''/ -];\\ super : \tau > /t]\tau'' \Longrightarrow list_e\end{array}}\;\text{T − LInvk}$$

$$\frac{\begin{array}{c}\Sigma; A \vdash e_1 : t.¡< B; super : \tau' > \Longrightarrow list_{e_1}\\ \Sigma; A' \vdash e_2 : \tau \Longrightarrow list_{e_2} \quad m \notin B\end{array}}{\begin{array}{c}\Sigma; A, A' \vdash e_1 \leftrightarrow m = e_2 :\\ t.¡< B, m : \tau; super : \tau' > \Longrightarrow list_{e_1}, list_{e_2}\end{array}}\;\text{T−AddM}$$

$$\frac{\begin{array}{c}\Sigma; A \vdash e_1 : t.¡< ..., m : \tau', ...; super : \tau'' >\\ \Longrightarrow lst_{e_1}\\ \Sigma; A' \vdash e_2 : \tau \Longrightarrow lst_{e_2}\end{array}}{\begin{array}{c}\Sigma; A, A' \vdash e_1 \leftrightarrow m = e_2 :\\ t.¡< ..., m{:}\tau, ...; super{:}\tau''{>} \Longrightarrow lst_{e_1}, lst_{e_2}\end{array}}\;\text{T−LChanMBd}$$

$$\frac{\begin{array}{c}\Sigma; A \vdash e_1 : t. < ..., m : \tau', ...; super : \tau >\\ \Longrightarrow list_{e_1}\\ \Sigma; A' \vdash e_2 : \tau' \Longrightarrow \{\} \quad \tau' \text{ is nonlinear}\end{array}}{\begin{array}{c}\Sigma; A, A' \vdash e_1 \leftrightarrow m = e_2 :\\ t.¡< ..., m : \tau', ...; super : \tau > \Longrightarrow list_{e_1}\end{array}}\;\text{T − ChanMBd}$$

Figure 11. Static semantics of expressions. {} represents the empty list.

$(T - Invk)$ A $\Leftarrow$ m expression is well-typed in the non-linear case if **e** (the receiver) and **m** are well-typed, both non-linear and the argument type of **m** is the same as the type of the object. The **mtype** function (see Figure 12) returns the type of the method that is invoked. The type returned is the type where the type of **self** is updated with the type of the receiver of the method. The type system does not allow **self** as a free variable for aliasing issues. Instead it requires an explicit lambda for **self**. The following example will help clarifying the problem

$let\ o\ =\ clone(Object)$
$\leftarrow\!\!+\ m = \lambda_{\_} : unit.self$
$\leftarrow\!\!+\ m' = \lambda_{\_} : unit.self \Leftarrow m\ in$
$(o \Leftarrow m) \Leftarrow m'$ /*method m is invoked twice*/

$(T - LInvk)$ The difference of this rule from the one above is that **e** (the receiver) is linear and **m** can be either linear or non-linear. The new type of **e** does not allow the client to call **m** again if **m** is linear. This is expressed in the rule by the annotation $[m{:}\tau'{\rightarrow}\tau''/\ -]$. We do that by deleting the record for **m** from the record type of the object. This prevents aliasing of linear objects in the assumption list (the stack). The following example illustrates the idea

$let\ obj\ =\ clone(Object) \leftarrow\!\!+\ m = \text{¡}\lambda_{:} unit.self\ in$
$let\ obj2\ =\ obj \Leftarrow m\ in$
$let\ obj3\ =\ obj2 \Leftarrow m$ /*we have two references to obj*/

$(T - AddM)$ The type-system adds new methods only to linear objects because aliases to an object would not be aware of the new method. The assumption list used to type the expression is split to type the two different expressions, e_1 and e_2, in order to track the linearity of the objects. The list returned is the concatenation of the lists returned from the typing rules of **e** and **m**.

$(T - LChanMBd)$ This rule checks if the object is linear and then checks if the new method body is well-typed. We can change the type of the method when the receiver is linear just like we can add new methods.

$(T - ChanMBd)$ This rule checks if the object is non-linear and that the new method body has the same type as the existing one. We do that for the same typing problem we can have in the $T - AddM$ or $T - Deleg$.

$(T - Deleg)$ This rule permits only to change delegation for a linear object for the same reason the type-system only permits new methods for linear objects. The rule assures also to delegate to a non-linear object because if the type system allows the client to delegate to linear objects then we effectively have more than one reference to it.

$(T - ChanLin)$ This rule changes the linearity of an object from linear to non-linear. The super object is non-linear anyway.

$(T - Appl)$ This rule checks if the first expression **e**$_1$ has a function type and that the second expression **e**$_2$ has the same type of the argument of e_1.

Figure 13 contains the rules for type-checking the store **S**. $list_i$ is a list of all linear objects used to type the location $S[L_i]$. The store **S** is well-typed if every location in the store is well-typed. A location **L** is well-typed if the super object which it inherit is well-typed and each method's body defined for that location is well-typed. $list_{e_i}$ is a list of linear objects used during the typing of the expression **e**.

3.4 Soundness

In this section we describe the approach taken for proving type safety for our system. We define important conditions and present the key lemmas needed for proving preservation and progress. We also briefly consider the most interesting cases of the progress and preservation proofs. The full type safety proof is available in [4].

$$\frac{\begin{array}{c}\Sigma; A \vdash e_1 : t_1.\text{¡}< B_1; super : t'.R_1 > \Longrightarrow list_{e_1}\\ \Sigma; A' \vdash e_2 : t_2. < B_2; super : t''.R_2 > \Longrightarrow list_{e_2}\end{array}}{\begin{array}{c}\Sigma; A, A' \vdash e_1.delegate(e_2) :\\ t_1.\text{¡}< B_1; super : (t_2. < B_2; super : t''.R_2 >)[t_1/t_2] >\\ \Longrightarrow list_{e_1}, list_{e_2}\end{array}}\ \mathrm{T-Deleg}$$

$$\frac{\Sigma; A \vdash e : t.\text{¡}R \Longrightarrow list_e}{\Sigma; A \vdash change\_linearity(e) : t.R \Longrightarrow list_e}\ \mathrm{T-ChanLin}$$

$$\frac{\begin{array}{c}\Sigma; A \vdash e_1 : \tau'[\rightarrow / \multimap]\tau'' \Longrightarrow list_{e_1}\\ \Sigma; A' \vdash e_2 : \tau' \Longrightarrow list_{e_2}\end{array}}{\Sigma; A, A' \vdash e_1 e_2 : \tau'' \Longrightarrow list_{e_1}, list_{e_2}}\ \mathrm{T-Appl}$$

$$\frac{m \in B \quad B =< ..., m : \tau, ... >}{mtype(m, t.[\text{¡}] < B, super : t'.R >, t''.R'') = \tau[t''.R''/t]}$$

$$\frac{m \notin B}{\begin{array}{c}mtype(m, t.[\text{¡}] < B, super : t'.R >, t.[\text{¡}] < B, super : t'.R >)\\ = mtype(m, t'.R, t.[\text{¡}] < B, super : t'.R >)\end{array}}$$

Figure 12. Rules for expressions, continued, and rules for looking up a method's type in a record type.

$$\frac{\forall L_i \in dom(\Sigma).\Sigma; \cdot \vdash S(L_i) : \Sigma(L_i) \Longrightarrow list_i}{\Sigma; \cdot \vdash S : \Sigma \Longrightarrow concat(list_i)}\ \mathrm{T-Store}$$

$$\frac{\Sigma(L) = t'.R \quad \forall m_i : e_i.\Sigma; \cdot \vdash e_i : \tau_i \Longrightarrow list_{e_i}}{\begin{array}{c}\Sigma; \cdot \vdash < super : L, m_1 : e_1, ..., m_n : e_n >:\\ t.[\text{¡}] < \overline{\tau_i}; super : t'.R > \Longrightarrow concat(list_{e_i})\end{array}}\ \mathrm{T-Odescr}$$

Figure 13. Static semantic of Store

Preservation Preservation ensures that the type of an expression is preserved during its evaluation.

The most interesting issue in type preservation is ensuring that linear references are not duplicated. For example, in an untyped version of EGO consider an object A that contains a linear method M, which in turn contains a reference to a linear object L. If we could clone A, then we would get a copy of M in the clone, and we could invoke both versions of M to extract two references to L. Our type system prohibits this by checking that clone is never called on an object with linear methods.

For the proof of preservation, we need two standard properties about the substitution operation as it occurs in function application.

Lemma 1 (Properties of Typing)
(i) (Weakening) If $\Sigma; A, A' \vdash$ e'$: \tau' \Longrightarrow list_{e'}$ *and* τ *is non-linear then* $\Sigma; A, x : \tau, A' \vdash e' : \tau' \Longrightarrow list_{e'}$.
(ii) (Substitution)If $\Sigma; A, x : \tau, A' \vdash e' : \tau' \Longrightarrow list_{e'}$ *and* $\Sigma; \cdot \vdash e : \tau \Longrightarrow list_e$ *then* $\Sigma; A, A' \vdash \{e/x\}e' : \tau' \Longrightarrow list_{e'}, list_e$

The first property follows from rule T-Kill, the second is proved by rule induction on the typing judgment for e'.

As a program executes, the number of locations in the store can expand as **clone** operations are performed, and the types of locations can change as a result of method addition or delegation changes. We formalize the way the store type can change as a store extension operation $\Sigma' \geq_L \Sigma$. This judgment means that Σ' differs from Σ because of L in one of two cases :

1. Σ' may have an additional L in its domain

$$\frac{dom(\Sigma') = dom(\Sigma) \cup \{L\} \quad \forall L' \in dom(\Sigma).\Sigma(L') = \Sigma'(L')}{\Sigma' \geq_L \Sigma}$$

2. L was linear in Σ but is non-linear in Σ'

$$\frac{dom(\Sigma') = dom(\Sigma) \quad \forall L' \in \{dom(\Sigma) - L\}.\Sigma(L') = \Sigma'(L')}{\Sigma' \geq_L \Sigma}$$

The first differ case of Σ' and Σ is introduced by the **clone** typing rule and the second case is introduced by the **addMethod**, **delegate** or **change_linearity** typing rules. This lemma is used for the proof of T-AddM, T-Deleg and T-Appl.

Next, we define two lemmas that are useful in ensuring that linear methods and objects remain unaliased as the program executes.

Lemma 2
If $\Sigma; A \vdash e : \tau \implies list_e$, $\Sigma' \geq_L \Sigma$ and $L \notin list_e$ then $\Sigma'; A \vdash e : \tau \implies list_e$.

This lemma is used to proof that if the old list **(**$list_{e_1}, list_{e_2}, list_S$**)** of linear objects has no duplicates and part of that list **(**$list_{e_1}, list_S$**)** has changed **(**$list_{e'_1}, list'_S$**)** because of an evaluation rule then the modified list **(**$list_{e'_1}, list_{e_2}, list'_S$**)** has no duplicates. This is because if there are no duplicates in the bigger list there could not possibly be duplicates in the smaller one.

The proof is by rule induction on the typing judgment for e.

Lemma 3
For any rule, $e, S \rightarrow e', S'$, where $\Sigma; \cdot \vdash e : \tau \implies list_e$, $\Sigma; \cdot \vdash S : \Sigma \implies list_S$ and no duplicates $list_e, list_S$, and for $\Sigma' \geq_L \Sigma$ and $\Sigma'; \cdot \vdash e' : \tau \implies list_{e'}$, $\Sigma'; \cdot \vdash S' : \Sigma' \implies list_{S'}$ and no duplicate in $list_{S'}, list_{e'}$ then $\{list_{S'}\} \cup \{list_{e'}\} \subseteq \{list_S\} \cup \{list_e\} \cup \{L\}$.

The proof is by rule induction on the evaluation judgment.

Theorem 4 (Preservation)
If $\Sigma; \cdot \vdash e : \tau \implies list_e$ and $\Sigma; \cdot \vdash S : \Sigma \implies list_S$ and there are no duplicates in $list_e, list_S$ and $e, S \rightarrow e', S'$ then for some $\Sigma' \geq_L \Sigma$ we have $\Sigma'; \cdot \vdash e' : \tau \implies list_{e'}$ and $\Sigma'; \cdot \vdash S' : \Sigma' \implies list_{S'}$ and there are no duplicate in $list_{e'}, list_{S'}$.

Proof: By rule induction on the derivation of $e, S \rightarrow e', S'$.

To give an idea of the preservation proof we present the case where e is typed with rule T-Deleg. There are three subcases, two for congruence rules (of which we show the first; the second is symmetric) and one for the evaluation rule:

Case

$$\frac{e_1, S \rightarrow e'_1, S'}{e_1.delegate(e_2), S \rightarrow e'_1.delegate(e_2), S'}$$

$e_1, S \rightarrow e'_1, S'$ Subderivation
$\Sigma; \cdot \vdash e_1.delegate(e_2) :$
$\quad t_1.\text{¡}< B_1; super : t_2. < B_2; super : t''.R_2 > [t_1/t_2] >$
$\quad \implies list_e$ Assumption
$\Sigma; \cdot \vdash S : \Sigma \implies list_S$ Assumption
No duplicate $list_e, list_S$ Assumption
$\Sigma; \cdot \vdash e_1 : t_1.\text{¡}< B_1; super : t'.R_1 >\implies list_{e_1}$ By inversion
$\Sigma; \cdot \vdash e_2 : t_2. < B_2; super : t''.R_2 >\implies list_{e_2}$ By inversion
$\Sigma' \geq_L \Sigma$, $\Sigma'; \cdot \vdash e'_1 : t_1.\text{¡}< B_1; super : t'.R_1 >\implies list_{e'_1}$ By i.h.
$\Sigma'; \cdot \vdash S' : \Sigma' \implies list_{S'}$ By i.h.
No duplicate $list_{e'_1}, list_{S'}$ By i.h.
$L \in list_{e'_1}, list_{S'}$ By definition of $\Sigma' \geq_L \Sigma$

No duplicates in $list_{e'_1}, list_{e_2}, list_{S'}$ because if $list_{e_2}, liste_1, list_S$ has no duplicates then from lemma 3 and $L \notin list_{e_2}$ we know that $list_{e_2}, list_{e'_1}, list_{S'}$ has no duplicates.

$\Sigma'; \cdot \vdash e_2 : t_2. < B_2; super : t''.R_2 >\implies list_{e_2}$ By lemma 2
$\Sigma'; \cdot \vdash e'_1.delegate(e_2) :$
$\quad t_1.\text{¡}< B_1; super : t_2.R_2 >\implies list_{e'_1}, list_{e_2}$ By rule

Case

$$\frac{S[L_1] = super : L \leftrightarrow \overline{M} \quad S' = S[L_1 \mapsto super : L_2 \leftrightarrow \overline{M}]}{L_1.delegate(L_2), S \rightarrow L_1, S'}$$

$\Sigma; \cdot \vdash L_1.delegate(L_2) :$
$\quad t_1.\text{¡}< B_1; super : t_2. < B_2; super : t''.R_2 >>$
$\quad \implies \{L_1\}$ Assumption
$\Sigma; \cdot \vdash S : \Sigma \implies list_S$ Assumption
No duplicate $L_1, list_S$ Assumption*
$\Sigma; \cdot \vdash L_1 : t_1.\text{¡}< B_1; super : t'.R_1 >\implies \{L_1\}$ By inversion
$\Sigma; \cdot \vdash L_2 : t_2. < B_2; super : t''.R_2 >\implies \{\}$ By inversion
let $\Sigma' = \Sigma[L_1 \rightarrow t_1.\text{¡}< B_1; super : t_2. < B_2; super : t''.R_2 >>]$
then $\Sigma'; \cdot \vdash L_1 : t_1.\text{¡}< B_1; super : t_2.R_2 >\implies \{L_1\}$ By rule
$\Sigma'; \cdot \vdash L_2 : t_2. < B_2; super : t''.R_2 >\implies \{\}$ By lemma 2
$\forall L' \in dom(\Sigma).\Sigma(L') = \Sigma'(L')$ By definition of $\Sigma' \geq_{L_1} \Sigma$
$\forall L' \in \{dom(S) - L_1\}$ and $\forall m \in S(L')$ then
$\Sigma'; \cdot \vdash M(m) : \tau \implies list_m$ By Assumption* and lemma 2
$\Sigma'; \cdot \vdash S' : \Sigma' \implies list_S$ By rule
No duplicate $L_1, list_S$ By assumption*

■

Progress Progress asserts that the evaluation of closed well-typed expressions will never get stuck, i.e. the expression is a value or can make an evaluation step.

The critical observation behind the proof is that a value of function type will indeed be a function and a value of object type be an object. We state these critical properties in inversion lemmas, because they are not immediately syntactically obvious.

Lemma 5 (Value inversion)
(i) If $\Sigma; \cdot \vdash v : t.R \implies list_v$ then $v = L$.
(ii) If $\Sigma; \cdot \vdash v : \tau'[\rightarrow / \multimap]\tau'' \implies list_v$ then $v = [\text{¡}]\lambda x : \tau'.e_0$.

Theorem 6 (Progress)
If $\Sigma; \cdot \vdash e : \tau \implies list_e$ and $\Sigma; \cdot \vdash S : \Sigma \implies list_S$ then either

(i) $e, S \rightarrow e', S'$ for some S' and e', or
(ii) e is a value v

Proof: By induction on the derivation of the typing judgment, analyzing all possible cases. ■

4. Related work

This section summarizes related work in language foundations, aliasing, and state-based method dispatch. SELF [25] was the first prototype-based language and also defined mechanisms for dynamic modifications of object definitions. In this paper, our goal is to statically typecheck many uses of SELF's mechanisms for dynamic object updates.

Abadi and Cardelli [1] use prototype-based object calculi to study issues of subtyping, quantification, and the typing of the receiver object *self*. Our work builds on this foundation, but because we incorporate first-class functions and linearity we use a notation taken more from the lambda calculus. Our calculus also differs from previous work in that we must model the *super* field directly because its value may change, whereas previous systems generally compile inheritance away once an object is created.

Variants of the Abadi-Cardelli object calculus taking into account object extensions are presented in [19, 22, 23]. In [15] Fisher, Honsell and Mitchell describe a delegation-based object calculus and method specialization where method extension represents delegation. Furthermore, in [14] they add a limited form of subtyping and type inference to their calculus. Compared to these systems, our work focuses on the orthogonal issue of statically checking the type safety of operations such as adding and removing methods or changing inheritance.

Re-classification in Fickle [12] allows an object to change its class at runtime in class-based OO languages. In this manner class-based OO languages can achieve the same effect as changing delegation at runtime. Fickle is more limited than our system because it restricts re-classification to a fixed set of state classes rather than supporting arbitrary changes to the methods and inheritance hierarchy of an object. Furthermore, because it does not track aliasing of fields, Fickle cannot track the state of an object in a field as EGO does.

Our work builds on Philip Wadler's linear type system [26], which in turn builds on a foundational linear logic developed by Girard [17]. The concept of linear types in [26] is used for resources that should not be duplicated or lost. In contrast, our system uses linear types to allow programs to safely change the type of an object, thus enabling highly dynamic language features for non-aliased objects. In the area of linear type systems, the primary contribution of this paper is showing how to naturally meld method or function linearity with object linearity. This issue is challenging due to the subtle interactions between method and object linearity in the presence of inheritance, method update, and method execution.

Linear logic is used as a tool for modeling OO programming in logic [3, 7, 11, 18]. In [7] methods are characterized as resources that reside within objects, and are consumed right after having been selected for evaluation upon invocation. We apply the intuition from this technique in a more concrete setting (i.e., operational semantics instead of an encoding in logic) in order to control aliasing for linear methods.

Predicate classes [8] and their more general form, predicate dispatch [13] support method dispatch based on predicates over the run-time state of the object. When a message is sent in these systems, the predicates of all relevant methods are evaluated, and the method chosen is the one with the most specific predicate that evaluates to true. Dynamic inheritance and dynamic method modification are complimentary ways to get similar behavior: instead of dispatching indirectly based on the state of an object, the state is encoded through the dispatch hierarchy. These mechanisms each appropriate in different situations; one advantage of our approach is that it can change the type of an object, rather than just which method is selected at run time.

Typestates were initially introduced by [24] for procedural programming languages. [9] defines a resource-controlling system for such languages based on keys. Keys can optionally be parameterized with typestates. This class of systems is formally modeled in [20] as refinement types that layer additional, changing resources on a conventional static type system. All these approaches do not consider inheritance and effectively only allow linear types. Thus they are unsuitable for object-oriented languages.

The State design pattern in [16] allows implementing different behavior for a method depending on the main object's state. However, there is no way of statically restricting the available methods for a state. [10] defined a model for statically tracking typestates in object-oriented languages. In particular, they address the issue of typestates in the presence of subtyping. In our work, objects have a dynamically changing type instead of a changing typestate layered on top of a fixed type. Our work also differs in supporting method addition, removal, and delegation change, rather than simply prohibiting calls to methods not applicable in the current typestate.

5. Conclusions

EGO is a prototype-based language that has expressiveness, simplicity and a static typechecker. The expressiveness follows from dynamic inheritance, adding methods, changing method bodies, and even changing method types dynamically. Its simplicity follows from the lack of the class concept, from the concept of cloning instead of instantiation, and from the unification of fields and methods.

EGO imposes restrictions on the programmer in order to control SELF's "power of simplicity". These are loose enough to allow interesting programs using EGO's dynamic features. But these restrictions are also strong enough to ensure EGO's static type safety. Its static typechecker provides a safer and more efficient paradigm than SELF: EGO programs will only contain valid method invocations.

We are currently implementing an interpreter for EGO, which will allow us to write and execute simple examples in the language, and test whether the restrictions on aliasing are reasonable in practice. In future work, we plan to investigate adding more advanced object-oriented language features to the system, including multiple inheritance, parametric polymorphism, and multiple dispatch. Our system can easily be extended to support subtyping for non-linear objects, but in the presence of dynamic type updates on linear objects, subtying is more challenging. Recent developments in typestate systems may provide a path forward here [10].

Acknowledgments

This work was supported in part by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298, NSF grant CCR-0204047, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems."

References

[1] M. Abadi, L. Cardelli. A theory of objects. Springer, 1996.

[2] J. Aldrich, V. Kostandinov, C. Chambers. Alias Annotations for Program Understanding. Proc. Object-Oriented Programming, Systems, Languages, and Applications, November 2002.

[3] J. M. Andreoli, R. Pareschi. Linear objects: Logical Processes with Built-In Inheritance. New Generation Computing, 9:445-473, 1991.

[4] A. Bejleri. A Type Checked Prototype-based Model with Linearity. Draft senior thesis, published as Carnegie Mellon Technical Report CMU-ISRI-04-142, December 2004.

[5] V. Bono, K. Fisher. An Imperative, First-Order Calculus with Object Extension. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1998.

[6] J. Boyland. Alias burying: Unique variables without reads. Journal : Software—Practice and Experience 31(6):533-553, May 2001.

[7] M. Bugliesi, G. Delzanno, L. Liquori, M. Martelli. Object Calculi in Linear Logic. Journal of Logics and Computation, 10(1): 75-104, 2000.

[8] C. Chambers. Predicate classes. Proc. European Conference on Object-Oriented Programming, 1993.

[9] R.DeLine, M. Fähndrich. Enforcing High-Level Protocols in Low-Level Software. Proc. Programming Language Design and Implementation, June 2001.

[10] R. DeLine, M. Fähndrich. Typestates for Objects. Proc. European Conference on Object-Oriented Programming, 2004.

[11] G. Delzanno, M. Martelli. Objects in Forum. ILPS 1995.

[12] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, P. Giannini. More Dynamic Object Reclassification: Fickle. ACM Transaction on Programming Languages and Systems 24(2):153-191 (2002).

[13] M. D. Ernst, C. Kaplan, C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. Proc. European Conference on Object-Oriented Programming, 1998.

[14] K. Fisher, J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. Proc. Fundamentals of Computation Theory, 1995.

[15] K. Fisher, F. Honsell, J. C. Mitchell. A lambda calculus of objects and method specialization. Nordic J. Computing (formerly BIT), 1:3-37, 1994.

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, October 1994.

[17] J.-Y. Girard. Linear logic. Theoretical Computer Science 50(1):1-102, 1987.

[18] N. Kobayashi, A. Yonezawa. Type-Theoretic Foundations for Concurrent Object-Programming. In Proceedings of the Ninth ACM-SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, 31-45, 1994.

[19] Luigi Liquori. An Extended Theory of Primitive Objects: First Order System. Proc. ECOOP'97.

[20] Y. Mandelbaum, D. Walker, R. Harper. An effective theory of type refinements. Proc. International Conference on Functional Programming, 2003.

[21] R. Milner, M.Tofte, R. Harper, D. MacQueen. The Definition of Standard ML (Revised). MIT Press, 1997.

[22] D. Rèmy. From Classes to Objects via Subtyping. In ESOP'98.

[23] J.C. Riecke, C.A. Stone. Privacy via Subsumption. FOOL'98 1998.

[24] R. E. Strom, S. Yemini. Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Engineering 12(1):157-171, January 1986.

[25] D. Ungar, R. B. Smith. Self: The power of simplicity. Proc. Object-Oriented Programming Systems, Languages, and Applications, 1987.

[26] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, Programming Concepts and Methods, North Holland, 1990.