e-Informatica Software Engineering Journal, Volume 1, Issue 1, 2007

Program Verifications, Object Interdependencies, and Object Types

Cong-Cong Xing*

*Department of Mathematics and Computer Science, Nicholls State University

cmps-cx@nicholls.edu

Abstract

Object types are abstract specifications of object behaviors; object behaviors are abstractly indicated by object component interdependencies; and program verifications are based on object behaviors. In conventional object type systems, object component interdependencies are not taken into account. As a result, distinct behaviors of objects are confused in conventional object type systems, which can lead to fundamental typing/subtyping loopholes and program verification troubles. In this paper, we first identify a program verification problem which is caused by the loose conventional object typing/subtyping which is in turn caused by the overlooking of object component interdependencies. Then, as a new object typing scheme, we introduce *object type graphs* (OTG) in which object component interdependencies are integrated into object types. Finally, we show how the problem existing in conventional object type systems can be easily resolved under OTG.

1 Introduction and Related Work

Although much of the recent year's work on object-oriented programming (OOP) has focused on large entities such as components, environments, and tools, investigations on issues related to object-oriented languages themselves are still an on-going research and many new improvements can be expected. In particular, typing and program verification are still a critical issue and a problem-prone area in the formal study of object-oriented languages, especially when type-related subjects, such as subtyping and inheritance, are considered. In the contexts of OOP theory research, there are three major lines: Abadi-Cardelli's ς-calculus [2], Fisher-Mitchell's lambda calculus of objects [14, 19, 18, 4], and Bruce's PolyTOIL [7, 6]. The type systems of all these calculi are *conventional* in the following sense: the major behavior indicator of objects – object component interdependencies – is not reflected in object types.

The result of not having such component interdependency information represented in object types is that two behaviorally distinct objects which deserve to be typed differently, may have the same type. For example, let objects a and b be defined, using the ς-calculus [2] notation, as follows: $a \stackrel{\text{def}}{=} [l_1 = 1, l_2 = 1]$, $b \stackrel{\text{def}}{=} [l_1 = 1, l_2 = \varsigma(s\!:\!Self)s.l_1]$ where s is the self variable and $Self$ is the type of s. The behavioral difference between a and b can be revealed by the following computations: Suppose we would like to update l_1 in a to 2.

It is easy to see that before and after this updating operation, the "status" of l_2 in a remains the same, namely, $a.l_2 = 1$ and $(a.l_1 \Leftarrow 2).l_2 = [l_1 = 2, l_2 = 1].l_2 = 1$[1]. However, when the same operation (updating l_1 to 2) is applied to b, the "status" of l_2 in b would be changed after the operation, namely, $b.l_2 = 1$ but $(b.l_1 \Leftarrow 2).l_2 = [l_1 = 2, l_2 = \varsigma(s : \mathit{Self})s.l_1].l_2 = 2$ due to the fact that l_2 "depends on" l_1 (l_2 calls l_1) in b. In conventional type systems, this behavioral difference between a and b is not captured in their types; a and b are of the same type: $[l_1 : int, l_2 : int]$. As a result, elusive programming errors and program verification problems will inevitably occur when subtyping is considered (as shown in the next section).

In this paper, we introduce, as a new way to represent object types, *object type graphs* (OTG) in which object component interdependency information is abstractly revealed, and show that OTG provides an effective support for program verifications. Section 2 presents a program verification problem caused by object typing. Section 3 defines a formal object-oriented language **TOOL** in which object component interdependencies are to be studied. Sections 4 and 5 define OTG and typing/subtyping under OTG respectively. Section 6 demonstrates how the program verification problem shown in Section 2 can be resolved under OTG. Section 7 concludes this paper.

There are some research work in the literature that are (somehow) related to our work. *Behavioral subtyping* is introduced in [20]. Although object behavior and subtyping are the common interests in both [20] and our paper, our typing approach is fundamentally different from that in [20] where object interdependencies are not considered. *Labeled types* and *width subtyping* are proposed in [3, 4, 19], where the type of a method is labeled by a set of methods that it uses. While the idea of labeled types is somewhat related to our idea of object interdependency, they differ substantially in quality and in quantity. For example, the notion of object interdependency is precisely defined in our work whereas the issue of method usages is not formally addressed in labeled types. Furthermore, in our work, object interdependencies fully participate and decisively reshape object subtyping whereas in labeled types the method usages information is barely used in object subtyping. The notion of *object state typing* can be found in, for example, [9, 21]. Just like [20] (as opposed to our work), this approach deals with the issue of object behavior and subtyping in a fundamentally different way from ours, which makes it orthogonal and complemental to our approach.

2 The Problem and Motivation

Points with additional attributes (e.g., color points [5, 8, 15], movable points [2, 4, 15]) have been an interesting study-case in the fundamental research of object-oriented languages. Here, we observe a new problem that is associated with movable points. We first present this problem on a theoretical basis and then demonstrate it using Java.

[1] $a.l_1 \Leftarrow 2$ means that field/method l_1 in a is updated to 2.

2.1 ς-calculus Description of the Problem

We stipulate that a point is *colored* (or *non-colored*, respectively), if this point (object) has a (or has no, respectively) color attribute. Let us consider non-negative movable points[2]. For 1-d movable points, we assume that all points greater than 1 are colored points and all other points are non-colored points (Figure 1). For 2-d movable points, similarly, we assume that all points with a distance from the origin greater than 1 are colored points and all other points are non-colored points (Figure 2). This assumption can be easily extended for higher-dimensional points.

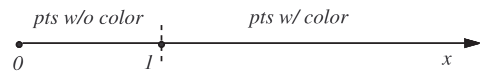

Figure 1: 1-d Colored and non-colored points

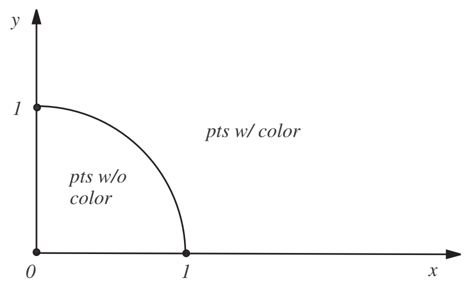

Figure 2: 2-d Colored and non-colored points

For instance, using the ς-calculus (second-order) notation [2], we can define a 1-d non-colored movable point and a 1-d colored movable point as follows:

$$p_{1n} \stackrel{\text{def}}{=} \begin{bmatrix} x = 0.5 \\ mvx = \varsigma(s\!:\!\mathit{Self})\lambda(i\!:\!\mathit{real})(s.x \Leftarrow s.x + i) \\ dist = \varsigma(s\!:\!\mathit{Self})s.x \end{bmatrix},$$

$$p_{1c} \stackrel{\text{def}}{=} \begin{bmatrix} x = 2.0 \\ mvx = \varsigma(s\!:\!\mathit{Self})\lambda(i\!:\!\mathit{real})(s.x \Leftarrow s.x + i) \\ dist = \varsigma(s\!:\!\mathit{Self})s.x \\ clr = blue \end{bmatrix},$$

where *mvx* moves the point to a new position on the x-axis and *dist* returns the dis-

[2]For the sake of simplicity, we only consider non-negative points here. The case for negative points can be easily duplicated with slight changes.

tance from the origin to the current position of the point. The $\Leftarrow$ is the method updating/overriding operation in ς-calculus. The intentions of fields x and clr are obvious.

To characterize the behaviors of 1-d movable points, we define the following types:

$$P \stackrel{\text{def}}{=} \varsigma(Self)\begin{bmatrix} x : real \\ mvx : real \to Self \\ dist : real \end{bmatrix},$$

$$CP \stackrel{\text{def}}{=} \varsigma(Self)\begin{bmatrix} x : real \\ mvx : real \to Self \\ dist : real \\ clr : color \end{bmatrix},$$

$$NCP \stackrel{\text{def}}{=} P,$$

where P is the type of all 1-d movable points, CP is the type of 1-d colored points, and NCP is the type of 1-d non-colored points. Given the objects and types defined as the above, it is easy to check that in conventional object type systems, we have $p_{1n} : NCP$, $p_{1c} : CP$, $CP <: P$, and $NCP <: P$.

Now, suppose we would like to write a program, ms ("move and see"), which takes a 1-d point and moves it along the x-axis. Due to the co-existence of colored and non-colored points on the x-axis, the movement cannot be arbitrary. We specify the behavior of ms as follows: (a) ms moves the argument point to its right a certain amount of distance if the argument point is colored (so that it will not mix with non-colored points). (b) ms moves the argument point to its left half of the distance from the origin to the current position of the argument point if the argument point is non-colored (so that it will not mix with colored points). (c) Let p' be the newly resulted point in cases (a) and (b). In case (a), ms uses the property $p'.dist > 1$ of p' to carry out the computation $\arcsin(1/p'.dist)$; in case (b), ms uses the property $p'.dist \leq 1$ of p' to carry out the computation $\arcsin(p'.dist)$. Because of subtyping and subsumption, inevitably, ms will take higher dimensional points as its arguments. To ensure that ms works fine with higher dimensional points, we require that, in such cases, the higher dimensional point be moved (right or left) along the x-axis, and the amount of distance to be moved follows the same guideline stated above. For example, given a 2-d point p with coordinates (x, y), if p is colored (which means $\sqrt{x^2 + y^2} > 1$), we move it to the right along the x-axis over a distance $\delta > 0$. The distance from the origin to the new position of the point then would be $\sqrt{(x+\delta)^2 + y^2} > \sqrt{x^2 + y^2} > 1$, indicating that the point is still in the colored point area on the x-y plane. If p is non-colored (which means $\sqrt{x^2 + y^2} \leq 1$), we move it to the left along the x-axis half of x. The distance from the origin to the new position of the point then would be $\sqrt{(\frac{1}{2}x)^2 + y^2} < \sqrt{x^2 + y^2} \leq 1$, indicating that the point is still in the non-colored point area. Thus the specification of the program ms is sound and feasible.

With little effort, we can write ms as follows:

$$
\begin{array}{l}
ms \stackrel{\text{def}}{=} \lambda(p : P) \\
\quad \textbf{if } (p.dist > 1) \qquad\qquad\qquad // \ p \text{ is colored} \\
\qquad sin^{-1}(1/(p.mvx(\delta)).dist) \qquad // \ \delta > 0 \\
\quad \textbf{else} \qquad\qquad\qquad\qquad // \ p \text{ is non-colored} \\
\qquad sin^{-1}((p.mvx(-\frac{1}{2}p.x)).dist) \\
\quad \textbf{endif}
\end{array}
$$

Figure 3: The function ms

Now, the question we have is: does ms perform to its specification with all permissible arguments? Or simply, is ms reliable? Can we verify its correctness?

It is easy to check that ms works as expected with p_{1n} and p_{1c}. We now define one colored 2-d point and two non-colored 2-d points as follows:

$$
p_{2c} \stackrel{\text{def}}{=} \begin{bmatrix} x = 2.0 \\ y = 2.0 \\ mvx = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.x \Leftarrow s.x + i) \\ mvy = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.y \Leftarrow s.y + i) \\ dist = \varsigma(s\!:\!Self)\sqrt{(s.x)^2 + (s.y)^2} \\ clr = blue \end{bmatrix},
$$

$$
p_{2n} \stackrel{\text{def}}{=} \begin{bmatrix} x = 0.5 \\ y = 0.3 \\ mvx = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.x \Leftarrow s.x + i) \\ mvy = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.y \Leftarrow s.y + i) \\ dist = \varsigma(s\!:\!Self)\sqrt{(s.x)^2 + (s.y)^2} \end{bmatrix},
$$

$$
p'_{2n} \stackrel{\text{def}}{=} \begin{bmatrix} x = 0.5 \\ y = \varsigma(s\!:\!Self)\frac{1}{4(s.x)} \\ mvx = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.x \Leftarrow s.x + i) \\ mvy = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.y \Leftarrow s.y + i) \\ dist = \varsigma(s\!:\!Self)\sqrt{(s.x)^2 + (s.y)^2} \end{bmatrix}.
$$

Note that p_{2c} and p_{2n} can be regarded as "free" 2-d points since their x and y fields are independent each other, whereas p'_{2n} can be regarded as a "constrained" 2-d point since its y coordinate depends on its x coordinate. Also note that p'_{2n} is a legitimate non-colored point since its coordinate is $(0.5, 0.5)$ which shows that the distance from the origin to this point is less than 1. Moreover, note that although p_{2c}, p_{2n}, and p'_{2n} are defined from scratch, they could have been defined through inheritance from (the classes of) p_{1c} or p_{1n} in class-based object-oriented languages (as shown in the next subsection).

Under conventional object type systems, p_{2c} and p_{2n} have types

$$CP_2 \stackrel{\text{def}}{=} \varsigma(\mathit{Self})\begin{bmatrix} x : \mathit{real} \\ y : \mathit{real} \\ \mathit{mvx} : \mathit{real} \rightarrow \mathit{Self} \\ \mathit{mvy} : \mathit{real} \rightarrow \mathit{Self} \\ \mathit{dist} : \mathit{real} \\ \mathit{clr} : \mathit{color} \end{bmatrix}$$

and

$$NCP_2 \stackrel{\text{def}}{=} \varsigma(\mathit{Self})\begin{bmatrix} x : \mathit{real} \\ y : \mathit{real} \\ \mathit{mvx} : \mathit{real} \rightarrow \mathit{Self} \\ \mathit{mvy} : \mathit{real} \rightarrow \mathit{Self} \\ \mathit{dist} : \mathit{real} \end{bmatrix}$$

respectively, and p'_{2n} has the same type as p_{2n}. That is, $p'_{2n} : NCP_2$. Furthermore, $CP_2 <: P$ and $NCP_2 <: P$, so $ms(p_{2c})$, $ms(p_{2n})$ and $ms(p'_{2n})$ all type-check.

It is easy to check that $ms(p_{2c})$ and $ms(p_{2n})$ work just fine. What about $ms(p'_{2n})$? It is supposed to return the degree of an angle. Unfortunately, the execution of $ms(p'_{2n})$ produces a run-time error, as outlined below: The current position of p'_{2n} is $(0.5, 0.5)$ with $p'_{2n}.\mathit{dist} = \sqrt{0.5^2 + 0.5^2} < 1$. So it is moved to the left $\frac{0.5}{2} = 0.25$ units of distance resulting in another point, say, p''_{2n}. The position of p''_{2n} is $(0.25, \frac{1}{4\times 0.25}) = (0.25, 1)$ and the distance from the origin to p''_{2n} is $p''_{2n}.\mathit{dist} = \sqrt{0.25^2 + 1^2} > 1$. The execution $\mathit{sin}^{-1}(p''_{2n}.\mathit{dist})$ thus crashes because sin^{-1} is undefined for argument greater than 1.

What goes wrong is clear: when the x-coordinate of p'_{2n} is moved (decreased), its y-coordinate is *implicitly* moved too (increased) due to the interdependency between x and y ($y = \frac{1}{4(s.x)}$). The combination of these two movements makes p'_{2n} (a non-colored point) go into the colored point area of the x-y plane, resulting in a point with distance greater than 1 and creating semantics confusions. The importance of object component interdependencies to object behaviors can be seen clearly here. Conceptually, for ms to safely fulfill its specifications, it should not take an arbitrary point as its argument. Any points in which some methods depend on x and affect dist at the same time, for example p'_{2n}, will potentially make the behavior of ms unpredictable and endanger the execution of ms when they are submitted to ms. Thus, allowing points like p'_{2n} to be submitted to ms is a *"wrong idea"*, in the sense that $ms(p'_{2n})$ does not work as specified and therefore ms is unreliable.

How can we fix this problem? Is the function ms composed incorrectly? Is there a way to rewrite ms so that we can prove that ms works as specified for all permissible arguments? It seems unlikely. Note that ms is written with P as the type of its argument. ms cannot foresee what kind of extra methods there are in its actual arguments. When p'_{2n} is submitted to ms, p'_{2n}'s y-coordinate is *invisible* to ms. ms does not know the existence of the y-coordinate, and of course, has no way of knowing the interdependencies between y and other methods and the ensuing behavior of p'_{2n}. This is especially the case if p'_{2n} is constructed via inheritance from p_{1c} or p_{1n}. This situation causes the behavior of ms (with various permissible arguments) unpredictable, and is *inevitable* in OOP supported by conventional object type systems.

2.2 Java Version of the Problem

To show that the problem exists not only in object-based languages, but in classed-based languages as well, we present a Java version of the problem with two running scripts in Figure 4.

Classes `P`, `CP`, `CP2`, and `NCP2` correspond to types P (and NCP), CP, CP_2, and NCP_2 respectively. Similarly, objects `p1n`, `p1c`, `p2n`, `p2c` and `p2na` correspond to points p_{1n}, p_{1c}, p_{2n}, p_{2c}, and p'_{2n} respectively. `MPP` and `MPP1` are two applications that use these points. Due to the "class-serves-as-type" feature of Java, the Java version of the problem is twisted a bit: The types of `p2n` and `p2na` are `NCP2` and `NCP2a` respectively. These two types are not the same but enjoy a subtyping relationship `NCP2a` <: `NCP2`. This is different from ς-calculus where p_{2n} and p'_{2n} have the same type, but does not affect the illustration of the problem.

Note that in class `NCP2a` of Figure 4, in order to faithfully implement the desired fact that "y-coordinate depends on x-coordinate", we need to use the combination of the field `y` and the method `y()` to *simulate* it. This is due to the imperative feature of Java. Field `y`, as an instance variable, once acquires a value, will evaluate to the same value each time it is evaluated. So field `y` does not "depend on" anyone in this sense. Then how can we code "y-coordinate depends on x-coordinate"? The use of an auxiliary method `y()` which depends on `x` (as desired) comes into help.

From the execution script of `MPP`, we can clearly see that submitting the "constrained" point `p2na` to the function `ms` causes a run-time bug, which demonstrates that the type `NCP2a` of `p2na` should *not* be regarded as a subtype of the type `P` although `NCP2a` is inherited (indirectly) from `P`. Considering that all `ms(p1c)`, `ms(p2c)`, `ms(p2n)` work fine and all the classes (types) of the three objects `p1c`, `p2c`, `p2n` are inherited (indirectly) from `P` too, we need to distinguish (all) inheritances in Java so that some inheritances (e.g., those as `CP`, `CP2`, and `NCP2`) may imply subtyping and others (e.g., those as `NCP2a`) do not. This can be done by using object interdependency as a measurement. Unfortunately, Java thinks "all inheritance is subtyping". What is more interesting is that due to the way in which Java handles `NaN` (Any arithmetic operation involving `NaN` and other operands produces a `NaN`, but any relational operation involving `NaN` and other operands produces either true or false[3].), this run-time bug can become hidden and difficult to find if the relevant expression is (deeply) involved with other computations. `MPP1` is such an example; by just examining the execution script of `MPP1`, it is hard to tell that `ms(p2na)` has actually caused a run-time bug.

[3]There are other means in Java to make the "illegal value" `NaN` legal, e.g., `(int)(Math.asin(2))` evaluates to 0, which could also help to conceal the `NaN` run-time bugs.

```
// class P. Note that this is also class
// NCP since NCP is defined as P.
public class P {
  protected  double x = 0.5;

  public double getx()
  { return x; }

  public void mvx(double i)
  {  x = x+i; }

  public double dist()
  { return getx();}
}


// class CP, inherited from P
public class CP extends P { String clr = "blue";
  public CP()
  { x = 2.0;}
}


// class CP2, inherited from CP
public class CP2 extends CP {
   protected double y;

   public CP2()
   { y = 2.0;}

   public double gety()
   { return y; }

   public void mvy(double i)
   { y = y+i; }

   public double dist()
   { return Math.sqrt(getx()*getx() + gety()*gety());}
}


// class NCP2, inherited from P
public class NCP2 extends P {
   protected double y;

   public NCP2()
   { y = 0.3;}

   public double gety()
   { return y; }

   public void mvy(double i)
   {  y = y+i; }

   public double dist()
   { return Math.sqrt(getx()*getx() + gety()*gety());}
}


// class NCP2a, inherited from NCP2. Need the combination
// of y and y() to simulate "y depends on x".  Note that
// "y depends on x" is what we want to do, without the use
// of y(), fields x and y would be independent
 public class  NCP2a extends NCP2
 {
   public NCP2a()
   { y = y(); }         // calling y() to get
                        // value for y

   public double y()    // implementation of
   { return 1/(4*x);}   // "y depends on x"

   public double gety()
   { y = y();           // calling y() to get
     return y;          // value for y
   }
}
```

```
// Application that uses P, CP, CP2, NCP2, and NCP2a
 public class MPP {
 public static void ms(P p)
 { if (p.dist() > 1)
   {System.out.println("  This is a colored point");
    p.mvx(1);                 // move p as specified
    System.out.println("  The result is: "+Math.asin(1/p.dist()));
   }
   else
   {System.out.println("  This is a non-colored point");
    p.mvx(-0.5*p.getx());     // move p as specified
    System.out.println("  The result is:  "+Math.asin(p.dist()));
   }
 }
 public static void main(String args[])
 {  P p1n = new P();
    CP p1c = new CP();
    CP2 p2c = new CP2();
    NCP2 p2n = new NCP2();
    NCP2a p2na = new NCP2a();

    System.out.println("making call ms(p1n)..."); ms(p1n);
    System.out.println("making call ms(p1c)..."); ms(p1c);
    System.out.println("making call ms(p2n)..."); ms(p2n);
    System.out.println("making call ms(p2c)..."); ms(p2c);
    System.out.println("making call ms(p2na)..."); ms(p2na);
 }
}

// Application that uses P, CP, CP2, NCP2, and NCP2a
public class MPP1 {
   public static void ms(P p)
   {   System.out.print("  Check to see if the result > PI/4:");
       if (p.dist() > 1)
        { p.mvx(1);                 // move p as specified
          if (Math.asin(1/p.dist()) > (Math.PI)/4)
             System.out.println ("   yes");
          else
             System.out.println ("   no");
        }
       else
       {  p.mvx(-0.5*p.getx());    // move p as specified
          if (Math.asin(p.dist()) > (Math.PI)/4)
             System.out.println ("   yes");
          else
             System.out.println ("   no");
        }
   }
   public static void main(String args[])
   { // omitted, same as the part in MPP }
}

C:\MyJavaPrograms\Point\movable pt problem>java MPP making call ms(p1n)...
  This is a non-colored point
  The result is:  0.25268025514207865
making call ms(p1c)...
  This is a colored point
  The result is: 0.3398369094541219
making call ms(p2n)...
  This is a non-colored point
  The result is:  0.40118821299725976
making call ms(p2c)...
  This is a colored point
  The result is: 0.2810349015028136
making call ms(p2na)...
  This is a non-colored point
  The result is:  NaN

C:\MyJavaPrograms\Point\movable pt problem>java MPP1 making call ms(p1n)...
  Check to see if the result > PI/4:   no
making call ms(p1c)...
  Check to see if the result > PI/4:   no
making call ms(p2n)...
  Check to see if the result > PI/4:   no
making call ms(p2c)...
  Check to see if the result > PI/4:   no
making call ms(p2na)...
  Check to see if the result > PI/4:   no
```

Figure 4: Java Code of the Movable Point Problem

Summarizing what is described in this section, we can state the problem as follows:

- In OOP supported by conventional object type systems, there is no way to implement programs like ms reliably and verify its correctness.

Motivated by this problem, we propose, in the subsequent sections, a new typing scheme for objects.

3 A Simple Typed Object-Oriented Language

To illustrate our approach, we define a simple typed object-oriented language (**TOOL**) in this section.

3.1 Syntax

The terms and types of **TOOL** are defined as follows.

$$
\begin{array}{l}
M ::= x \mid \lambda(x{:}\sigma).M \mid M_1 M_2 \mid M.l \mid M.l \Leftarrow \varsigma(x{:}\mathcal{S}(A))M' \\
\qquad\quad \mid [l_i = \varsigma(x{:}\mathcal{S}(A))M_i]_{i=1}^n \\
\sigma ::= \kappa \mid t \mid \sigma_1 \to \sigma_2 \mid \mu(t)\sigma \mid A \mid \mathcal{S}(A) \\
A ::= \iota(t)[l_i(L_i){:}\sigma_i]_{i=1}^n \quad L_i \subseteq \{l_1, \ldots, l_n\} \text{ for each } i
\end{array}
$$

Terms in **TOOL** are standard λ-terms and ς-terms [2]. In particular, $[l_i = \varsigma(x : \mathcal{S}(A))M_i]_{i=1}^n$ represents an object, $M.l$ represents method invocation, and $M.l \Leftarrow \varsigma(x : \mathcal{S}(A))M'$ represents method updating.

Types in **TOOL** are standard ground type, function type, recursive type, and the newly proposed object type. In object type $\iota(t)[l_i(L_i) : \sigma_i]_{i=1}^n$, ι is the self-type binder, each method l_i has type σ_i, and L_i is the set of *links* of l_i (defined in the next subsection). $\mathcal{S}(A)$ denotes the self type induced by the object type A. $A = \iota(t)[l_i(L_i){:}\sigma_i(t)]_{i=1}^n$ if and only if $A = [l_i(L_i){:}\sigma_i(\mathcal{S}(A))]_{i=1}^n$.

We provide a simple example to illustrate the syntax of types and terms. Let

$$
A \stackrel{\text{def}}{=} \iota(t) \begin{bmatrix} l_1(\{l_2, l_3\}) : t \\ l_2(\emptyset) : int \\ l_3(\{l_2\}) : int \to int \end{bmatrix}.
$$

It specifies that l_1, l_2, and l_3 are of self type (associated with A), int, and $int \to int$ respectively. The sets of links for l_1 and l_3 are $\{l_2, l_3\}$ and $\{l_2\}$. l_2 has no links. An object of type A could be

$$
a \stackrel{\text{def}}{=} \begin{bmatrix} l_1 = \varsigma(s{:}\mathcal{S}(A))s \\ l_2 = 1 \\ l_3 = \varsigma(s{:}\mathcal{S}(A))\lambda(x{:}int)(x + s.l_2) \end{bmatrix}.
$$

3.2 Definition of Links

Links are used to signify the structure of component dependency of objects. Informally, in object type $\iota(t)[l_i(L_i) : \sigma_i]_{i=1}^n$, $l_j \in L_i$ means that the value of method l_i depends (partially) on the value of method l_j. The link mechanism makes the types of objects in **TOOL** substantially different from that in conventional object type systems.

Definition 1 (Link) Given an object $a = [l_i = \varsigma(s : \mathcal{S}(A))M_i]_{i=1}^n$, (1) l_i is said to be ***dependent*** on $l_j (i \neq j)$ if there exists a M such that $a.l_i$ and $(a.l_j \Leftarrow \varsigma(s : S(A))M).l_i$ evaluate to different values; (2) l_i is said to be ***directly dependent*** on $l_j (i \neq j)$ if (a) l_i is dependent on l_j, and (b) if all such $l_k (i \neq k, j \neq k)$ where l_i is dependent on l_k and l_k is dependent on l_j, are removed from a, l_i is still dependent on l_j; (3) The set of ***links*** of l_i (or equivalently, of M_i with respect to object a), denoted by $L(l_i)$ (or equivalently, by $L_a(M_i)$), contains exactly all such l_j on which l_i is directly dependent.

Example 1 Take the object a and its type A defined at the end of section 3.1, by the definition of links, we see that the links of the methods in a are:

$$\begin{aligned}
&L(l_1) = L_a(s) = \{l_2, l_3\} \\
&L(l_2) = L_a(1) = \emptyset \\
&L(l_3) = L_a(\lambda(x : int)(x + s.l_2)) = \{l_2\}
\end{aligned}$$

which match the corresponding link specifications in type A.

4 Object Type Graphs

4.1 Definitions

To reveal the structure of object component interdependencies more clearly and facilitate the study of object subtyping and behaviors, we introduce a graphical representation of object types – object type graphs. We define directed colored graphs first.

Definition 2 (Directed Colored Graph) A ***directed colored graph*** G is a 6-tuple (G_N, G_A, C, sr, tg, c) consisting of: (1) a set of ***nodes*** G_N, and a set of ***arcs*** G_A; (2) a ***color alphabet*** C; (3) a ***source map*** $sr : G_A \rightarrow G_N$, and a ***target map*** $tg : G_A \rightarrow G_N$, which return the source node and target node of an arc, respectively; and (4) a ***color map*** $c : G_N \cup G_A \rightarrow C$, which returns the color of a node or an arc.

Definition 3 (Ground Type Graph) A ***ground type graph*** is a single-node colored directed graph which is colored by a ground type.

Definition 4 (Function Type Graph) A ***function type graph*** $(s, G_1, G_2)_{(G_N,G_A,C,sr,tg,c)}$ is a directed colored graph consisting exactly of a ***starting node*** $s \in G_N$, and two type graphs G_1 and G_2, such that, (1) $c(s) = \rightarrow$; (2) there are two arcs associated with the starting node s, *left* arc $l \in G_A$ and *right* arc $r \in G_A$, such that $c(l) = in$, $c(r) = out$;

l connects G_1 to s by $sr(l) = s_{G_1}$, $tg(l) = s$, and r connects s to G_2 by $sr(r) = s$, $tg(r) = s_{G_2}$, where s_{G_1} and s_{G_2} are the starting nodes of G_1 and G_2, respectively; (3) G_1 and G_2 are disjoint; (4) if there is an arc $a \in G_A$ with $c(a) = rec$, then $sr(a) = s_{G_i}$, $tg(a) = s$, $c(s_{G_i}) = \rightarrow$, $i = 1, 2$.

Definition 5 (Object Type Graph) An ***object type graph*** $(s, A, R, L, S)_{(G_N, G_A, C, sr, tg, c)}$ is a directed colored graph consisting exactly of a ***starting node*** $s \in G_N$, a set of ***method arcs*** $A \subseteq G_A$, a set of rec-colored arcs $R \subseteq G_A$, a set of ***link arcs*** $L \subseteq G_A$, and a set of type graphs S, such that (1) $c(s) = self$. (2) $\forall a \in A$, $sr(a) = s$, $tg(a) = s_F$ for some type graph $F \in S$, and $c(a) = m$ for some method label m; $c(a) \neq c(b)$ for $a, b \in A$, $a \neq b$. (3) $\forall r \in R$, $c(r) = rec$, $tg(r) = s$, $sr(r) = s_F$ for some $F \in S$, and $c(s_F) = self$. (4) $\forall l \in L$, $sr(l) = s_F$, $tg(l) = s_G$ for some $F, G \in S$, and $c(l) = bym$ for some method label m.

Remarks: Directed colored graph is the foundation of graph grammar theory [10, 11, 12, 13, 22]. Object type graphs are adapted from directed colored graphs. Ground type graphs are trivial. Function type graphs are straightforward. They need to be defined because an object type graph may include them as subgraphs. An object type graph is formed by a starting node s and a set S of type graphs with each $F \in S$ being connected to s by a method arc that goes from s to F. The starting node s is colored by *self* and is used to denote the self type. The method interdependencies are specified by arcs in L. If $L(m)$ is the set of links of method m, then for each $l \in L(m)$ there is an arc (colored by *byl*) that goes from l to m. Recursive object types are specially indicated by rec-colored arcs in R.

For the sake of brevity, we drop the subscripts in $(s, G_1, G_2)_{(G_N, G_A, C, sr, tg, c)}$ and $(s, A, R, L, S)_{(G_N, G_A, C, sr, tg, c)}$ whenever possible throughout the paper.

4.2 Examples of Object Type Graphs

We now provide some examples to illustrate the definition of object type graphs.

Example 2 In Figure 5, A, B, and C are the type graphs for ground types *int*, *real*, and *bool* respectively. D is the type graph for function type $int \rightarrow int$ and E is the type graph for $(int \rightarrow real) \rightarrow (real \rightarrow int)$.

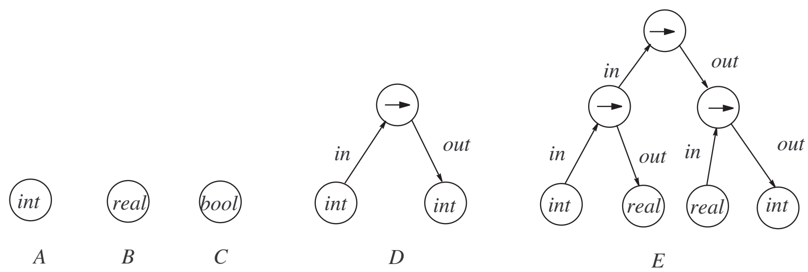

Figure 5: Examples of Ground Type Graphs and Function Type Graphs

Example 3 In Figure 6, graph A denotes the object type $[x\!:\!int, y\!:\!int]$, where methods x and y are independent of each other. Graph B denotes the type $[x : int, y(\{x\}) : int]$ where y depends on x. Note that the direction of the link arc in B is from x to y, (not from y to x), signifying the fact that changes made to method x will affect method y.

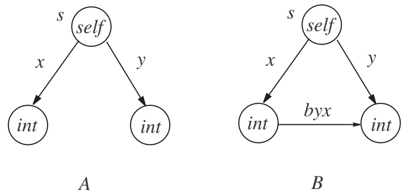

Figure 6: Examples of Object Type Graphs

Example 4 In Figure 7, graph C represents the object type $\mu(t)\iota(s)[a : int, b : t, c : s]$. Method a is of type *int*; method b is of recursive object type C. Method c is of the self type induced by the object type C. Note the structural difference between the type of b and the type of c revealed in the type graph[4]. Graph D represents the type of a simplified 1-d movable point $[x = 1, mvx = \varsigma(s\!:\!\mathcal{S}(D))\lambda(i\!:\!int)(s.x \Leftarrow s.x + i)]$. The facts that mvx depends on x and returns a modified self are indicated by the *byx*-colored arc and the *out*-colored arc in D.

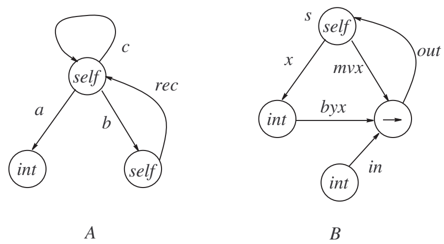

Figure 7: Examples of Object Type Graphs

Example 5 Two more object type graphs are shown in Figure 8. They are the types of some variations of point objects. Graph A is the type of the object

$$\begin{bmatrix} x = 1, \\ m_1 = \varsigma(s\!:\!\mathcal{S}(A))\lambda(i\!:\!int)p \\ m_2 = \varsigma(s\!:\!\mathcal{S}(A))\lambda(i\!:\!int)s \end{bmatrix}$$

[4]This structural setting, potentially, will allow the type of c to remain as self type and the type of b to be changed after some operations on graph C are performed.

where p is some point object of type A. Graph B is the type of the object

$$\begin{bmatrix} x = 1 \\ y = 2 \\ d = \varsigma(s\!:\!\mathcal{S}(B))(s.x + s.y)/2 \\ e_1 = \varsigma(s\!:\!\mathcal{S}(B))\lambda(p\!:\!B)(p.x = s.x \ \wedge p.y = s.y) \\ e_2 = \varsigma(s\!:\!\mathcal{S}(B))\lambda(p\!:\!\mathcal{S}(B))(p.x = s.x \ \wedge p.y = s.y) \end{bmatrix}.$$

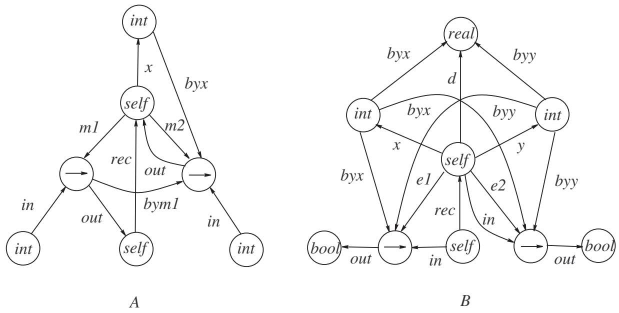

Figure 8: Examples of Object Type Graphs

5 Object Typing/Subtyping Under OTG

We now investigate the issue of typing/subtyping under OTG. We first define object subtyping through a series of definitions and then present the typing/subtyping rules with a brief discussion. Note that OTG is just another way (a graphical way, specifically) to represent object types. There is a natural 1-1 correspondence between OTG and the normal textual representations of object types in **TOOL**. So the typing rules presented in this section naturally apply to object type graphs. What makes OTG significant is its facilitation of the formulation of object subtyping with the presence of links in object types (as addressed below).

Definition 6 (Type Graph Premorphism) Let $\mathbf{\Phi}$ be the set of ground types. Given two type graphs $G = (G_N, G_A, C, sr, tg, c)$ and $G' = (G'_N, G'_A, C', sr', tg', c')$, a ***type graph premorphism*** $f: G \rightarrow G'$ is a pair of maps $(f_N: G_N \rightarrow G'_N, f_A: G_A \rightarrow G'_A)$, such that (1) $\forall a \in G_A$, $f_N(sr(a)) = sr'(f_A(a))$, $f_N(tg(a)) = tg'(f_A(a))$, and $c(a) = c'(f_A(a))$; (2) $\forall v \in G_N$, if $c(v) \in \mathbf{\Phi}$, then $c'(f_N(v)) \in \mathbf{\Phi}$; otherwise $c(v) = c'(f_N(v))$.

Definition 7 (Base, Subbase) Given an object type graph $G = (s, A, R, L, S)$. The ***base*** of G, denoted by $Ba(G)$, is the graph $(s, A, t(A), L)$, where $t(A) = \{tg(a) \mid a \in A\}$. A ***subbase*** of G is a subgraph $(s, A', t(A'), L')$ of $Ba(G)$, where $A' \subseteq A$, $L' \subseteq L$, $t(A') = \{tg(a) \mid a \in A'\}$, and for each $l \in L'$ there exist $a_1, a_2 \in A'$ such that $sr(l) = tg(a_1)$ and $tg(l) = tg(a_2)$.

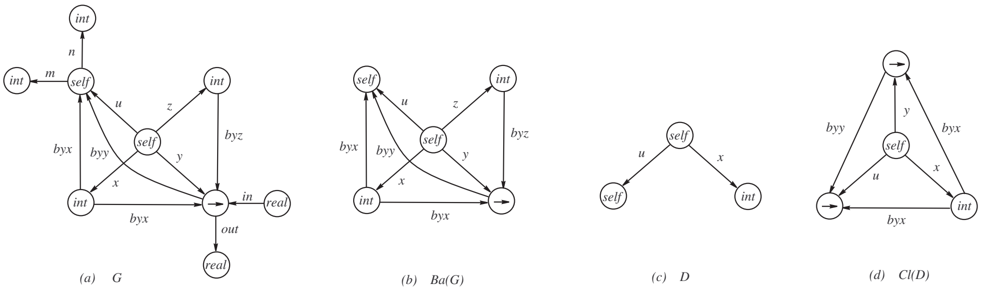

Figure 9: (a) An object type graph G; (b) The base $Ba(G)$ of G; (c) A subbase D of G; (d) The closure $Cl(D)$

Definition 8 (Closure, Closed) The ***closure*** of a subbase $D = (s, A', t(A'), L')$ of an object type graph $G = (s, A, R, L, S)$, denoted by $Cl(D)$, is the union $D \cup E_1 \cup E_2$, where (1) $E_1 = \{l \in L \mid \exists a_1, a_2 \in A' \text{ with } tg(a_1) = sr(l),\ tg(a_2) = tg(l)\}$, and (2) $E_2 = \{l, h, a, t(l) \mid l, h \in L,\ a \in A,\ a \notin A',\ tg(l) = sr(h) = tg(a), \text{ and } \exists a_1, a_2 \in A' \text{ such that } tg(a_1) = sr(l),\ tg(a_2) = tg(h)\}$. A subbase D is said to be ***closed*** if $D = Cl(D)$.

Definition 9 (Covariant, Invariant) Given an object type graph (s, A, R, L, S). Let $t(A) = \{tg(a) \mid a \in A\}$. For each $v \in t(A)$, if v is not incident with any links, or if v is the target node of some links but not the source node of any links, then v is said to be ***covariant***; otherwise, v is said to be ***invariant***.

Definition 10 (Object Subtyping) Given two object type graphs $G = (s_G, A_G, \emptyset, L_G, S_G)$ and $F = (s_F, A_F, \emptyset, L_F, S_F)$. $F <: G$ if and only if the following conditions are satisfied: (1) There exists a premorphism f from $Ba(G)$ to $Ba(F)$ such that $f(Ba(G)) = Cl(f(Ba(G)))$. That is, $f(Ba(G))$ is closed. (2) For each node v in $f(Ba(G))$, let u be its preimage in $Ba(G)$ under f, $F_v \in S_F$ be the type graph with v as its starting node, and $G_u \in S_G$ be the type graph with u as its starting node. (i) If v is invariant, then $F_v \cong G_u$. (ii) If v is covariant, then $F_v <: G_u$.

Remarks: Type graph premorphism is adapted from graph morphism which is a fundamental concept in algebraic graph grammars [13, 10, 22, 11, 12]. It preserves the directions and colors of arcs and the colors of nodes up to ground types. The base of an object type graph singles the method interdependency information out of the entire object type graph so that the structure of the method interdependencies can be better studied. The closure of a subbase captures the complete behavior of the subbase by including, in addition to all methods and links in the subbase, a set E_2 of methods (and associated links) outside of the subbase in the following way: for any method l in E_2, (1) l depends on some methods inside the subbase, and (2) there exist some methods inside the subbase that depend on l. An example of base, subbase, and closure is shown in Figure 9. Object subtyping is defined using the ideas of type graph premorphism, base, subbase, closure, and variance property. It first ensures that the behavior of a subobject (indicated by method interdependencies) is the same as that of a superobject through the closure requirement. Then, it uses the variance information of each method to check the subtyping feasibility of each method type (graph) in a subobject with its counterpart in a superobject[5]. Note that in the definition of object subtyping, we only consider the case $R = \emptyset$ (i.e., no recursive object types). The case $R \neq \emptyset$ requires complicated graph grammar operations and is beyond the scope of this paper.

The typing/subtyping rules of **TOOL** are shown in Table 1. The rules that are affected by links are (TObj) and (TUpd). Note that in these rules, the set of links computed from terms are checked against the set of links specified in types.

$$\frac{}{\emptyset \rhd \diamond}(\mathrm{TC}\emptyset) \qquad \frac{\Gamma \rhd \sigma \quad x \notin dom(\Gamma)}{\Gamma, x{:}\sigma \rhd \diamond}(\mathrm{TCVar}) \qquad \frac{\Gamma \rhd M:\sigma \quad x \notin dom(\Gamma)}{\Gamma, x{:}\tau \rhd M:\sigma}(\mathrm{T}x)$$

$$\frac{\Gamma \rhd \diamond}{\Gamma \rhd \kappa}(\mathrm{TyCons}) \qquad \frac{\Gamma \rhd \sigma \quad \Gamma \rhd \tau}{\Gamma \rhd \sigma \to \tau}(\mathrm{TyFun})$$

$$\frac{\Gamma \rhd \sigma_i \quad \forall i \in \{1,\ldots,n\}}{\Gamma \rhd \iota(t)[l_i(L_i){:}\sigma_i(t)]_{i=1}^n}(\mathrm{TyObj},\ L_i \subseteq \{l_1,\ldots,l_n\} \text{ for each } i)$$

$$\frac{\Gamma \rhd \diamond \quad x{:}\sigma \in \Gamma}{\Gamma \rhd x{:}\sigma}(\mathrm{TVar}) \qquad \frac{\Gamma, x{:}\sigma \rhd M{:}\tau}{\Gamma \rhd \lambda(x{:}\sigma).M : \sigma \to \tau}(\mathrm{TAbs}) \qquad \frac{\Gamma \rhd M{:}\sigma \to \tau \quad \Gamma \rhd N{:}\sigma}{\Gamma \rhd MN : \tau}(\mathrm{TApp})$$

$$\frac{\Gamma, s{:}\mathcal{S}(A) \rhd M_i{:}\sigma_i \quad L_i = L_a(M_i) \quad \forall i \in \{1,\ldots,n\}}{\Gamma \rhd a : A}\left(\mathrm{TObj},\ \begin{array}{l} a = [l_i = \varsigma(s{:}\mathcal{S}(A))M_i]_{i=1}^n \\ A = \iota(t)[l_i(L_i){:}\sigma_i(t)]_{i=1}^n \end{array}\right)$$

$$\frac{\Gamma \rhd M : A \quad j \in \{1,\ldots,n\}}{\Gamma \rhd M.l_j : \sigma_j(A)}(\mathrm{TInv1},\ A = \iota(t)[l_i(L_i){:}\sigma_i(t)]_{i=1}^n = [l_i(L_i){:}\sigma_i(\mathcal{S}(A))]_{i=1}^n)$$

$$\frac{\Gamma \rhd s : \mathcal{S}(A) \quad j \in \{1,\ldots,n\}}{\Gamma \rhd s.l_j : \sigma_j(A)}(\mathrm{TInv2},\ A = \iota(t)[l_i(L_i){:}\sigma_i(t)]_{i=1}^n = [l_i(L_i){:}\sigma_i(\mathcal{S}(A))]_{i=1}^n)$$

$$\frac{\Gamma \rhd N{:}A \quad \Gamma, s{:}\mathcal{S}(A) \rhd M{:}\sigma_i \quad L_i = L_N(M) \quad i \in \{1,\ldots,n\}}{\Gamma \rhd N.l_i \Leftarrow \varsigma(s{:}\mathcal{S}(A))M : A}(\mathrm{TUpd},\ A = \iota(t)[l_i(L_i){:}\sigma_i(t)]_{i=1}^n)$$

$$\frac{\Gamma \rhd \sigma}{\Gamma \rhd \sigma <: \sigma}(\mathrm{SRefl}) \qquad \frac{\Gamma \rhd \sigma <: \tau \quad \Gamma \rhd \tau <: \delta}{\Gamma \rhd \sigma <: \delta}(\mathrm{STran}) \qquad \frac{\Gamma \rhd a{:}A \quad \Gamma \rhd A <: B}{\Gamma \rhd a{:}B}(\mathrm{SSump})$$

$$\frac{\Gamma \rhd \sigma' <: \sigma \quad \Gamma \rhd \tau <: \tau'}{\Gamma \rhd \sigma \to \tau <: \sigma' \to \tau'}(\mathrm{SFun})$$

$$\frac{\Gamma \rhd G_A <: G_B}{\Gamma \rhd A <: B}\left(\mathrm{SObj},\ \begin{array}{l} G_A \text{ and } G_B \text{ are the OTGs of } A \text{ and } B \text{ respectively} \\ A = \iota(t)[l_i(L_i){:}\sigma_i(t)]_{i=1}^n, B = \iota(t)[l'_i(L'_i){:}\sigma'_i(t)]_{i=1}^{n'} \end{array}\right)$$

Table 1: Typing and Subtyping Rules for **TOOL**

We would like to emphasize that the purpose of object type graphs is to facilitate the formulation and reasoning of object subtyping when method interdependencies are considered in object types. This can be seen in the object subtyping rule (SObj) where the determination of $A <: B$ for object types A and B depends on whether their object

[5]Ground subtyping and function subtyping which are involved in object subtyping are standard as in the literature.

type graphs G_A and G_B have a subtyping relationship which, in turn, can be decided by the Definition 10. (Definition 10 suggests an immediate algorithm for how to compute $G_A <: G_B$.)

6 Verification of the Program ms under OTG

We have shown, in section 2, that under conventional object type systems, there is no way to code the function ms satisfactorily in the sense that we are unable to prove that ms performs to its specification for all permissible arguments. In this section, we show that this problem can be easily resolved under OTG typing/subtyping. That is, we show that ms can be coded reliably under OTG typing/subtyping and prove that it performs to its specification in all situations.

Given the code of ms in Figure 3 and under the OTG notation, the type of the point p_{1n} (which is also the type of the parameter in the function ms) and the type of the point p'_{2n} are depicted as P and Q'_{2n} in Figure 10[6]. Let f be the premorphism from base $Ba(P)$ to base $Ba(Q'_{2n})$, $f(Ba(P))$ and its closure $Cl(f(Ba(P)))$ are also shown in Figure 10. By the OTG object subtyping definition (Definition 10), we can see that $Q'_{2n} \not<: P$ because $f(Ba(P)) \neq Cl(f(Ba(P)))$ (i.e., $f(Ba(P))$ is not closed). Hence, p'_{2n} cannot be viewed as having type P and $ms(p'_{2n})$ does not type-check. The run-time error of $ms(p'_{2n})$ is therefore prevented by type checking at compile-time. Hence, the code of ms in Figure 3 is safe under the OTG typing/subtyping.

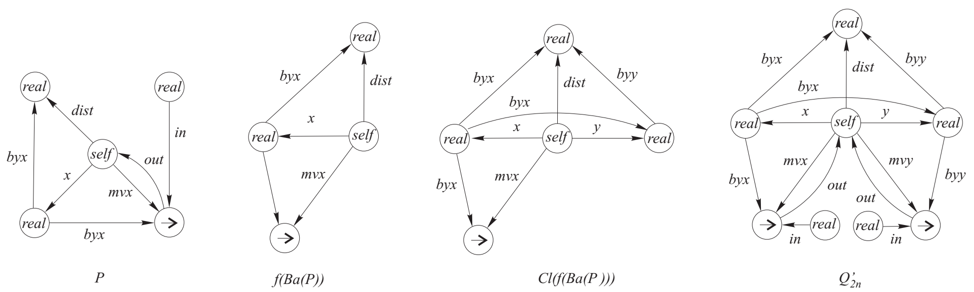

Figure 10: Resolution of the Movable Point Problem in OTG

To fully revisit of the movable point problem in the context of OTG, the type graphs of p_{1c}, p_{2c}, and p_{2n} are depicted in Figure 11 as Q_{1c}, Q_{2c}, and Q_{2n}, respectively. We can easily check, using Definition 10, that $Q_{1c} <: P$, $Q_{2c} <: P$, and $Q_{2n} <: P$ all hold. This shows that the desired executions $ms(p_{1c})$, $ms(p_{2c})$, and $ms(p_{2n})$ are all supported by OTG typing/subtyping scheme.

From Figure 10 and Figure 11, we see that the type of p'_{2n} and the type of p_{2n} are different under OTG (as opposed to the same in conventional type systems). The fact that method y depends on method x in p'_{2n} and method y does not depend on method x in

[6]For the sake of conciseness, some unimportant links that do not affect the result of illustration, such as the link from method *dist* to method *mvx*, are not shown in Figure 10.

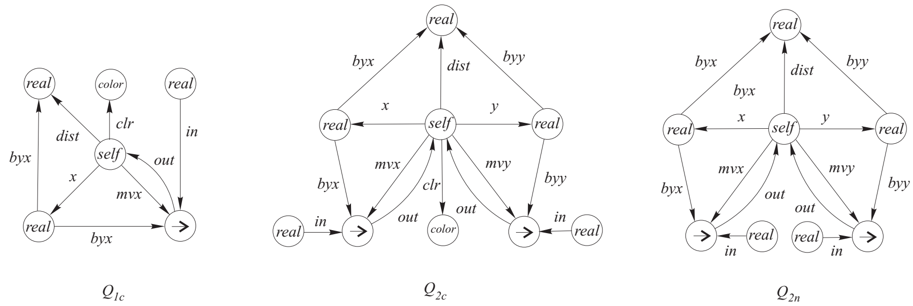

Figure 11: Types of p_{1c}, p_{2c}, and p_{2n} in OTG

p_{2n} (i.e., p_{2n} and p'_{2n} have different behaviors) is faithfully captured in their type graphs as the presence/absence of a link from method x to method y. Indeed, this distinction is necessary in order to prevent run-time errors such as those caused by $ms(p'_{2n})$. This observation leads to the following proposition.

Proposition 1 *Let A be the type of an object a in which there is a link between method x and method y. Let B be the type of an object b which is modified from a by deleting the link between method x and method y. Then $A \neq B$.*

Also note that in Figure 10 and Figure 11, we have $Q'_{2n} \not<: Q_{2n}$ (we can easily verify this by Definition 10). This disallowance of subtyping is also necessary in order to statically prevent similar run-time errors caused by $ms(p'_{2n})$. Thus,

Proposition 2 *Let A and B be as specified in Proposition 1. Then $A \not<: B$.*

We now show the correctness of ms in Figure 3 under the OTG typing scheme. We assume that all arguments (1-d points, 2-d points, ...) submitted to ms are "correctly" coded. In particular, if p is an n-dimensional point with coordinates $x_1, \ldots, x_n$, then its method *dist* must have $\sqrt{x_1^2 + \cdots + x_n^2}$ as the body; and its method *mvx* must have $\lambda(i\!:\!real)s.x \Leftarrow (s.x+i)$ as the body; how other methods in p are coded is irrelevant to the proof. This is a reasonable assumption, for if p is coded "incorrectly" or arbitrarily (say, p's *dist* body is $\sqrt{x_1^2 + 4x_2^2 + \cdots + n^2x_n^2}$), then there would be no way to expect what kind of behavior ms can have with p as its argument.

To facilitate the proof, we rewrite the functional program ms in Figure 3 equivalently into an imperative one in Figure 12, where a holds the computation result. We would like to prove, under the framework of Hoare logic (e.g. [16, 17]), that the two Hoare triples

$$(\!|p.dist > 1 \wedge p\!:\!P|\!)ms(p)(\!|p.dist > 1 \wedge p\!:\!P|\!)$$
$$(\!|p.dist \leq 1 \wedge p\!:\!P|\!)ms(p)(\!|p.dist \leq 1 \wedge p\!:\!P|\!)$$

are valid for any point p of type P in Figure 10. The first triple specifies that ms keeps a colored point in the colored point area after moving it. The second triple specifies that ms keeps a non-colored point in the non-colored point area after moving it. Before proving the validity of the triples, we prove a lemma first. Let colored points and non-colored points be defined as in section 2, we can show that

$$
\begin{array}{l}
ms \stackrel{\text{def}}{=} fun(p : P)\ \{ \\
\quad real\ a; \\
\quad \textbf{if}\ (p.dist > 1)\{ \\
\qquad p.mvx(\delta); \quad //\ \delta > 0 \\
\qquad a = sin^{\text{-}1}(1/p.dist); \\
\quad \} \\
\quad \textbf{else}\ \{ \\
\qquad p.mvx(-\frac{1}{2}p.x); \\
\qquad a = sin^{\text{-}1}(p.dist); \\
\quad \} \\
\}
\end{array}
$$

Figure 12: The Imperative Version of the Program *ms*.

Lemma 1 *Given an n-dimensional point p, if p is a non-colored point and is of type P in Figure 10, then after being moved, along the x-axis and towards the origin, half of the projection of the distance from the origin to p's current position over the x-axis, p is still in the non-colored point area in the space.*

Proof: Without loss of generality, we assume that the coordinates of p are $x_1, x_2, \ldots, x_n$ ($n > 1$) with x_1 being the x-coordinate, x_2 being the y-coordinate, ... Since p is a non-colored point, we have $\sqrt{x_1^2 + \cdots + x_n^2} \leq 1$. After p is moved as specified, its x-coordinate would be changed to $\frac{1}{2}x_1$. Since p is n-dimensional and $n > 1$, the actual type of p must be a subtype of P. By the definition of OTG subtyping (Definition 10), we know that the x-coordinate change of p will *not* affect any other coordinates $x_2, \cdots, x_n$ of p because all $x_2, \cdots, x_n$ occur in the method *dist* of p and *dist* appears in type P[7]. Thus, $x_2, \ldots, x_n$ all retain their old values after p's move. Therefore, the distance from the origin to the new position of p is $\sqrt{(\frac{1}{2}x_1)^2 + x_2^2 + \cdots + x_n^2} < \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \leq 1$, indicating the p is still in the non-colored point area. □

The validity of the second Hoare triple is given in Theorem 1 below. The proof of the first Hoare triple is similar and omitted.

Theorem 1 *Given the program ms in Figure 12, the Hoare triple*

$$(\!|p.dist \leq 1 \wedge p\!:\!P|\!)ms(p)(\!|p.dist \leq 1 \wedge p\!:\!P|\!)$$

is valid.

Proof: The proof, shown in Figure 13, is an application of the standard imperative program verification rules (see e.g. [17]). In Figure 13, $p.d$ and $p.m$ stand for $p.dist$ and $p.mvx$, and A, B, C, D, E, F, G stand for the following triples respectively:

[7] Here is a subtle point indicated by the OTG object subtyping: if any of the coordinates $x_2, \ldots, x_n$, say x_i, does not occur in method *dist* (or in any other method included in type P), then we allow x_i be affected by the changes of x_1 while requiring that the type of p is a subtype of type P.

$\llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis \{p.m(-\frac{1}{2}p.x)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis \{a = \mathit{sin}^{\text{-}\mathit{1}}(p.d)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis \perp \rrparenthesis \{p.m(\delta); a = \mathit{sin}^{\text{-}\mathit{1}}(\mathit{1}/p.d)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis \{p.m(-\frac{1}{2}p.x); a = \mathit{sin}^{\text{-}\mathit{1}}(p.d)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis p.d \leq 1 \wedge p : P \wedge p.d > 1 \rrparenthesis \{p.m(\delta); a = \mathit{sin}^{\text{-}\mathit{1}}(\mathit{1}/p.d)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis p.d \leq 1 \wedge p : P \wedge p.d \leq 1 \rrparenthesis \{p.m(-\frac{1}{2}p.x); a = \mathit{sin}^{\text{-}\mathit{1}}(p.d)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis ms(p) \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$.
The validity of triple A on the top of the proof tree is provided by Lemma 1. □

$$\dfrac{\dfrac{C}{E}(\text{implication}) \qquad \dfrac{\dfrac{\dfrac{}{A}(\text{Lemma 1}) \quad \dfrac{}{B}(\text{assignment})}{D}(\text{composition})}{F}(\text{implication})}{G}(\text{if-statement})$$

Figure 13: The Proof of ms's Property

7 Conclusion and Future Work

Typing is an efficient means in program verifications. Object component interdependency information is critical in determining and predicting object behaviors and in shaping object types. If this information is not captured in object typing, as is the case in conventional object type systems, then a statically well-typed program may go wrong at run-time causing run-time errors and program verification troubles. We proposed object type graphs (OTG) as an initial treatment for handling object component interdependencies in object typing and program verifications. We have seen that due to OTG's ability of revealing more information about object behaviors,

- Programs that go wrong at run-time in conventional object type systems can be effectively detected at compile-time under OTG typing/subtyping.
- Program verifications that cannot be done with conventional object type systems can be easily carried out with the support of OTG typing/subtyping.

This demonstrates that OTG is a safer typing scheme than conventional ones, and provides a valuable support for OOP program verifications. The following issues are of immediate interests for future work:

- Devise a link computation algorithm and assess its complexity.
- Prove/disprove that the standard properties of type systems, such as subject reduction and soundness, hold under OTG.
- As far as applying the idea of OTG to practical object-oriented languages is concerned, we believe that a direct approach would be to adapt OCaml [1] by modifying its type for classes. Influenced by OOP theory research, Ocaml, unlike other object-oriented languages (e.g. Java) where classes are the sole type of objects, gives a

type for each of its classes. In a sense, the type of a class in OCaml is the (more abstract) type of the object generated by that class. This is a typical case where practice benefits from theory, and it would be very interesting to keep extending OCaml along this line.

References

[1] *http://caml.inria.fr/ocaml/.* 2007.

[2] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[3] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping for Extensible, Incompete objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.

[4] V. Bono and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects. In *Proc. of International Conference of Computer Science Logic*, number 933 in LNCS, pages 16–30. 1995.

[5] K. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[6] K. Bruce. *Foundations of Object-Oriented Languages*. MIT Press, 2002.

[7] K. Bruce, A. Schuett, R. van Gent, and A. Fiech. Polytoil: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems*, 25(2):225–290, 2003.

[8] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings of the 17the Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[9] R. Deline and M. Fahndrich. Typestates for objects. In *ECOOP 2004*, 2004.

[10] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Applications to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, 1978.

[11] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1999.

[12] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3. World Scientific, 1999.

[13] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: An algebraic approach. In *IEEE Conference of Automata and Switching Theory*, pages 167–180, 1973.

[14] K. Fisher, F. Honsell, and J. Mitchell. A lambda calculus of objects and method specialization. *Nodic Journal of Computing*, 1:3–37, 1994.

[15] J. Hickey. *Introduction to OCaml, http://caml.inria.fr/tutorials-eng.html.* 2002.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[17] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, 2nd edition, 2004.

[18] L. Liquori. On object extension. In *ECOOP'98 Object-oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 498–522. Springer–Verlag, 1998.

[19] L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. Number 1179 in Lecture Notes in Computer Science, pages 129–141. Springer–Verlag, 1996.

[20] B. Liskov and J. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[21] O. L. Madsen. Towards Integration of State Machines and Object-Oriented Languages. In *Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, 1999.

[22] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, 1997.