A Graph-Based Type Representation for Objects

Cong-Cong Xing[1]

[1] cmps-cx@nicholls.edu
Department of Mathematics and Computer Science
Nicholls State University
Thibodaux, LA, USA

Abstract: Subtyping and inheritance are two major issues in the research and development of object-oriented languages, which have been traditionally studied along the lines of typed calculi where types are represented as a combination texts and symbols. Two aspects that are closely related to subtyping and inheritance – method interdependency, and self type and recursive object type – have either been overlooked or not received sufficient/satisfactory treatments. In this paper, we propose a graph-based notation for object types and investigate the subtyping and inheritance issues under this new framework. Specifically, we (1) identity the problems that have motivated this paper; (2) propose an extension to Abadi-Cardelli's ς-calculus towards fixing the problems; (3) present definitions of object type graphs followed by examples; (4) define subtyping and inheritance using object type graphs; (5) show how the problems can be easily resolved under object type graphs; and (6) summarize the contributions of this paper.

Keywords: Object type, graph transformation

1 Introduction

As pointed out by Markku Sakkinen in [Sak05], although in recent years the emphasis of the research and development in object-oriented programming (OOP) has shifted from programming languages (themselves) to larger entities such as components, environments, and manipulating tools, it does not mean that the existing object-oriented languages are perfect and no improvement is needed. In particular, *typing* is 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.

One aspect related to subtyping is object method interdependencies: the invocation relationship among methods. The failure of keeping track of this invocation structure in object types can cause elusive programming errors which will inevitably occur, undermine the program reliability, and burden the program verification. One aspect related to inheritance is self type vs. recursive object type: which one is the *true* type of the self variable (in the context of inheritance). The failure of not distinguishing these two types sufficiently can lead to some well-known fundamental problems. While the former aspect has been overlooked in the literature, the latter has not received sufficient attentions and/or satisfactory treatments, in either theoretical studies (e.g., [AC96, FHM94, LC96, Liq98, BL95]) or main stream practice (e.g., Java and C++) of OOP. In the next section, we present concrete examples to illustrate this point.

2 Motivations

We present two problems that have motivated the writing of this paper.

2.1 Method Interdependency

We call a rectangle *free* if its two sides (height and width) are independent, *constrained* otherwise. In conventional type systems, the type of a free rectangle and the type of a constrained rectangle are not distinguished. We show, in this subsection, that this type confusion opens the door to let the different semantics of free rectangles and constrained rectangles be mixed, which is serious enough to be able to cause a program *not* to perform to its specification and thus weakens its reliability.

Using the first-order ς-calculus notation [AC96], we can construct a free rectangle *fRect*, a constrained rectangle *cRect*, and their types *FR*, *CR* as follows:

$$FR \stackrel{\text{def}}{=} \mu(Self)\begin{bmatrix} h : int \\ w : int \\ mvh : int \to Self \\ mvw : int \to Self \\ geth : int \\ getw : int \end{bmatrix}, \quad fRect \stackrel{\text{def}}{=} \varsigma(s:FR)\begin{bmatrix} h = 1 \\ w = 2 \\ mvh = \lambda(i:int)(s.h \Leftarrow s.h + i) \\ mvw = \lambda(i:int)(s.w \Leftarrow s.w + i) \\ geth = s.h \\ getw = s.w \end{bmatrix},$$

$$CR \stackrel{\text{def}}{=} \mu(Self)\begin{bmatrix} h : int \\ w : int \\ mvh : int \to Self \\ mvw : int \to Self \\ geth : int \\ getw : int \end{bmatrix}, \quad cRect \stackrel{\text{def}}{=} \varsigma(s:CR)\begin{bmatrix} h = 1 \\ w = 2(s.h) \\ mvh = \lambda(i:int)(s.h \Leftarrow s.h + i) \\ mvw = \lambda(i:int)(s.w \Leftarrow s.w + i) \\ geth = s.h \\ getw = s.w \end{bmatrix},$$

where $\Leftarrow$ is the field update operation, and the intentions of methods in these two rectangles are obvious. Note that in *fRect*, the height (h) and the width (w) are independent, whereas in *cRect*, the width *depends on* the height ($w = 2(s.h)$). Also note that $FR = CR$, that is, the types of these two rectangles are confused (in conventional type systems).

Now, suppose we would like to have a function with the following specification (contract):

> *This function takes a rectangle and then doubles both its height and its width.*

With little effort, such a function can be written as:

$$ds \stackrel{\text{def}}{=} \lambda(r : FR)(r.mvh(r.geth)).mvw(r.getw).$$

It is easy to check that ds will double its argument's both sides when taking a free rectangle as argument. However, when ds takes a constrained rectangle as argument, for example $ds(cRect)$ (due to the fact $CR = FR$, *cRect* will type-check), it will fail to do so, as it is supposed to (by the specification). In detail,

$$ds(cRect) = (cRect.mvh(cRect.geth)).mvw(cRect.getw)$$
$$= \varsigma(s:CR)\begin{bmatrix} h = 2 \\ w = 6 \\ \ldots no\ change \ldots \end{bmatrix}.$$

Clearly, the height of *cRect* is doubled, but its width is *tripled* (not doubled)! The reason for this is the interdependency between the height and the width in *cRect*: when the height of *cRect* is changed to 2, its width is *implicitly* changed to 4 due to the width's dependence on the height.

Considering that the widely-agreed notion of program reliability refers to (e.g. [Seb07]) "program performs to its specification under all circumstances" and that the fact that *ds* does not live up to its specification when taking *cRect* as its argument, we argue that the reliability of *ds*, in the environment of conventional type systems, is substantially low. Furthermore, such elusive computation fault may be hard-to-detect when *ds* is embedded in large software systems. To resolve this problem effectively, *ds* should be written in such a way that it only takes free rectangles, that is, $ds(cRect)$ should be caught by the type checker. This observation calls for the separation of the type of free rectangles from that of constrained ones.

2.2 Self Type and Recursive Object Type

The notion of self type is coined to describe the type of the self variable in an object, especially when the object contains a self returning method. Then the question is: What is the (semantics of) self type? Is it just the (recursive) object type or something else? For example, using the notation of ς-calculus again, an object which consists of two methods, one returning the constant 1 and one returning the hosting object itself can be coded as $a \stackrel{\text{def}}{=} [l_1 = 1,\ l_2 = \varsigma(s:X)s]$, where s is the *self* variable and X is the type of s – the self type. How do we interpret X? One "natural" way is that X is just the object type itself (recursively defined), that is, $X = \mu(Y)[l_1 : int,\ l_2 : Y]$. As this interpretation works to some extent but runs into substantial problems (see, e.g., [AC96] for details), other explanations of the self type have been sought. For example, the second-order self quantifier [AC96] and the MyType [Bru94] are proposed. Nevertheless, no mater how self type is interpreted, an object type has always been managed to be a subtype of the associated self type. This setup, combined with inheritance and dynamic dispatch of methods, leads to the well-known "method-not-found" error as illustrated below (adapted from [Bru94]).

$$PT \stackrel{\text{def}}{=} ObjectType(MyType)\{x : int, eq : MyType \to Bool\}$$
$$CPT \stackrel{\text{def}}{=} ObjectType(MyType)\{x : int, c : color,\ eq : MyType \to Bool\}$$
$$pt_0 \stackrel{\text{def}}{=} object(self : MyType)\{x = 0, eq = fun(p : MyType)(p.x = self.x)\}$$
$$pt \stackrel{\text{def}}{=} object(self : MyType)\{x = 1, eq = fun(p : MyType)(p.x = self.x)\}$$
$$cpt \stackrel{\text{def}}{=} inherited\ \ from\ pt\ with\ \{c = red,\ eq = fun(p : MyType)[(p.x = self.x) \wedge (p.c = self.c)]\}$$
$$F \stackrel{\text{def}}{=} fun(p : PT)(p.eq(pt_0))$$

Given these definitions, it is easy to check that $pt_0 : PT$, $pt : PT$, and $cpt : CPT$. Note that in the definition of F, we actually have assumed (as [Bru94] does) that the type of pt_0, PT,

is a subtype of the self type associated with PT, $MyType$ in this case, so that $p.eq(pt_0)$ type-checks. Now, if inheritance implies subtyping (as we have been practicing in C++ and Java), then $cpt : CPT <: PT$ and $F(cpt)$ will type check. However, $F(cpt)$ will crash and produce a "method-not-found" error because $cpt.eq(pt_0)$ expects its argument, pt_0, to have a color field and uses that color field in the body of eq of cpt, but pt_0 does not have the color field.

Traditionally, it is this kind of problem that has prompted us to claim that "inheritance is not subtyping" [CHC90]. However, "inheritance implies subtyping" is a strongly desirable property in OOP. Without it, the software hierarchy build through inheritance will be much less useful since in this case a subobject (object from a subclass) cannot be regarded as having the same type with its superobject (object from the superclass), and cannot use any existing programs that have been written for superobjects. Program reusability will thus be greatly reduced. Towards keeping this hierarchy useful and resolving the method-not-found problem at the same time, we propose that an object type should not only be treated differently from its associated self type, but not be regarded as a subtype of its associated self type either.

3 Enhancing Object Types

In order to address the problems outlined in the previous section, we extend Abadi-Cardelli's ς-calculus by adding a mechanism called links that capture the method interdependencies in objects, and by distinguishing (recursive) object types from their associated self types. The terms (M) and types (σ) of this extended calculus are as follows.

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

x, $\lambda(x{:}\sigma).M$, and $M_1 M_2$ are the standard λ-terms. $[l_i = \varsigma(x{:}\mathscr{S}(A))M_i]_{i=1}^n$ represents an object consisting of n methods, with names l_i and bodies M_i for each i. ς is the self-binder. $M.l$ means the invocation of method l in M. $M.l \Leftarrow \varsigma(x{:}\mathscr{S}(A))M'$ is the updating operation which evaluates to an object obtained by replacing method l in M by M'.

κ, t, $\sigma_1 \to \sigma_2$, and $\mu(t)\sigma$ are ground types, type variables, function types, and recursive types respectively. Object types are represented by $\iota(t)[l_i(L_i){:}\sigma_i]_{i=1}^n$ where each method l_i has type σ_i, and L_i is the set of *links* of l_i (defined below). ι is the self-type binder. An alternative way to represent self type is $\mathscr{S}(A)$ which denotes the self type associated with the object type A. The two notations are related by $A = \iota(t)[l_i(L_i) : \sigma_i(t)]_{i=1}^n = [l_i(L_i) : \sigma_i(\mathscr{S}(A))]_{i=1}^n$. Terms that can be of a self type are restricted to self (variable) or a modified self (for the sake of self variable specialization during inheritance).

Definition 1 Given an object $[l_i = \varsigma(s{:}\mathscr{S}(A))M_i]_{i=1}^n$. The only terms that are of type $\mathscr{S}(A)$ are s or $s.l_i \Leftarrow \varsigma(s{:}\mathscr{S}(A))M$ for some M.

The set of links, which is a part of the newly proposed object types, is defined as follows.

Definition 2 (*Links*) Given an object $a = [l_i = \varsigma(s{:}\mathscr{S}(A))M_i]_{i=1}^n$, (1) l_i is said to be ***dependent*** on $l_j (i \neq j)$ if there exists a M such that $a.l_i$ (or $a.l_i(\alpha)$ for some appropriate argument list α) and $(a.l_j \Leftarrow \varsigma(s{:}\mathscr{S}(A))M).l_i$ (or $(a.l_j \Leftarrow \varsigma(s{:}\mathscr{S}(A))M).l_i(\alpha)$) 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 in object a (or equivalently, of M_i with respect to object a), denoted by $L_a(l_i)$ (or equivalently, by $L_a(M_i)$), contains exactly all such l_j on which l_i is directly dependent.

4 Object Type Graphs

The notion of links introduces new structures into object types. Object types are thus enriched but also become more complicated. To effectively analyze and reason about the structure of the new object types, we present a graph-based representation for object types – object type graphs (OTG).

4.1 Definitions

Definition 3 *(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*** $sr : G_A \to G_N$, and a ***target map*** $tg : G_A \to G_N$, which return the source node and target node of an arc, respectively; and (4) a ***color map*** $c : G_N \cup G_A \to C$, which returns the color of a node or an arc.

Definition 4 *(Ground Type Graph)* A ***ground type graph*** is a single-node colored directed graph which is colored by a ground type.

Definition 5 *(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) $c(s) = \to$; (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 $sr(a) = s_{G_i}$, $tg(a) = s$, $c(s_{G_i}) = \to$, $i = 1, 2$.

Definition 6 *(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 [EPS73, Ehr78, Roz97]. 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 OTG

We now provide some examples to illustrate the definitions introduced in the last section. Throughout this section, if the type of an object a is represented by a graph A, we will say the type of a is A, and vice versa.

Example 1 *In Figure 1, A, B, and C are the type graphs for the three ground types int, real, and bool respectively. They are just a node colored by the appropriate ground types. D is the type graph for function type int → int. E is the type graph for (int → B) → int, where B is the object type in Figure 2(a) which will be explained in the next example.*

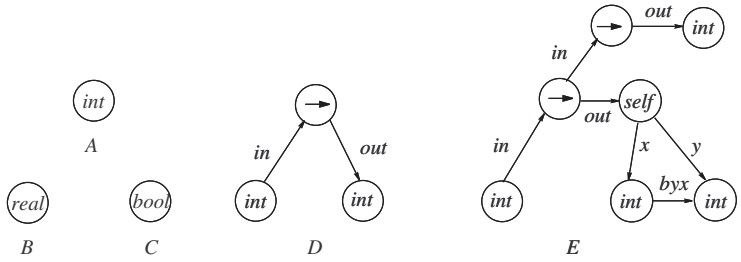
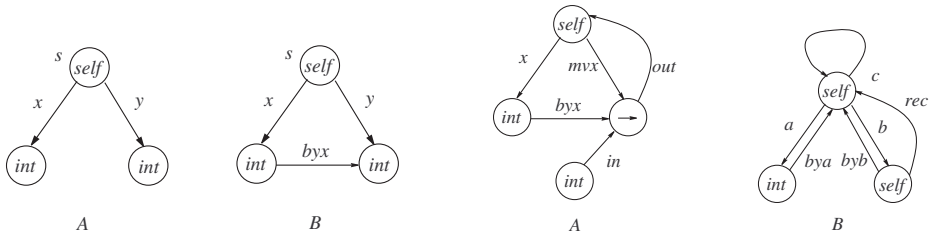


Figure 1: Examples of Ground Type Graphs and Function Type Graphs.

Example 2 *In Figure 2(a), 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), and that the link is colored by byx, signifying that changes made to method x will affect method y. For instance, an object of type A may be* $[x = 1,\ y = 2]$ *(which is actually a record), and an object of type B may be* $[x = 1,\ y = \varsigma(s : \mathscr{S}(B))(s.x + 2)]$.

Note also that although the presence of the link in B or the absence of the link in A serves as an extra condition (compared to conventional type systems) for selecting objects to be typed as A or B respectively, there are still infinitely many objects that are of type A or type B. For example, objects $[x = m, y = n]$ *with* $m, n \in \mathbf{N}$ *are all of type A; objects* $[x = n, y = \varsigma(s : B)(a(s.x) + b)]$ *with* $n, a, b \in \mathbf{N}$ *are all of type B. In this sense, OTG is (still) an abstract specification of object behaviors.*



(a) (b)

Figure 2: Example of Object Type Graphs

Example 3 *In Figure 2(b), A is the type graph for* $\iota(t)[x:int, mvx(\{x\}):int \to t]$ *which is the type of a simplified 1-d movable point* $[x=1, mvx=\varsigma(s:B)\lambda(i:int)(s.x \Leftarrow s.x+i)]$. *The facts that mvx depends on x and returns a modified self object are indicated by the byx-colored arc and the out-colored arc respectively. Note the direction of the out-colored arc goes to the starting node of the type graph directly, indicating that this is a self type (as opposed to recursive objet type). Graph B represents the type of the object* $[a=1,\ b=p,\ c(\{a,b\})=\varsigma(s:\mathscr{S}(B))s]$ *where p is some predefined object of type B. Here, note that the fact that b is of recursive object type is depicted by a sel f-colored node and a rec-colored arc going from this node to the starting node of the graph; and that the fact that c is of self type is depicted by its method arc going directly to the starting node of the graph. The difference between recursive object type and self type is clearly represented in object type graphs.*

5 Subtyping under OTG

Given the definition of OTG, we now investigate the issue of subtyping under OTG. Throughout the paper, we write $A_\sigma <: B_\tau$ iff $\sigma <: \tau$ where σ and τ are types and A_σ and B_τ are their type graphs. We first present the necessary definitions and then provide some subtyping examples.

5.1 Definitions

Definition 7 *(Type Graph Premorphism)* Let Φ 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*** $f:G\to G'$ is a pair of maps $(f_N:G_N\to G'_N, f_A:G_A\to G'_A)$, 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 $c(v)\in\Phi$, then $c'(f_N(v))\in\Phi$; otherwise $c(v)=c'(f_N(v))$.

Definition 8 *(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 $tg(l)=tg(a_2)$.

Definition 9 *(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 10 *(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 11 *(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 is isomorphic to 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 [EPS73, Ehr78, Roz97]. 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.

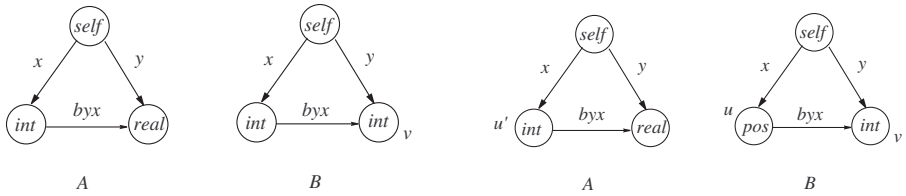
5.2 Examples

We now present some simple subtyping examples.

Example 4 *Given the two type graphs in Figure 3(a), clearly we can find a premorphism f from base of A to base of B such that $f(Ba(A))$ is closed; note also that node v in B is covariant. Thus, $B <: A$. As an example, we can regard the object $[x = 1,\ y = \varsigma(s : \mathscr{S}(B))(s.x + 1)]$ of type B as having type A.*

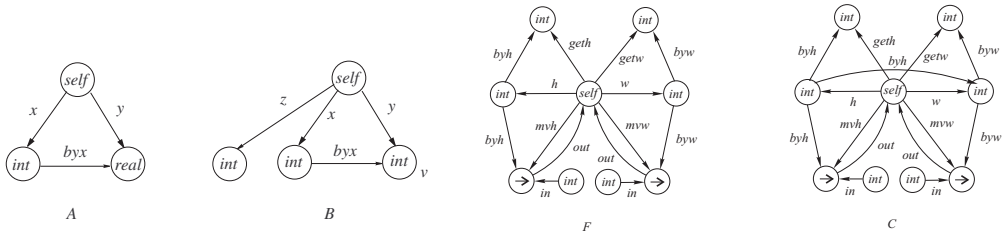
Example 5 *For the two type graphs in Figure 3(b), we can also find a premorphism from the base of A to the base of B, and node v in B is also covariant. But, node u in B is invariant which requires the corresponding node u' in A have the same color – pos (standing for positive integer) in order to have $B <: A$. But u' is colored by int, hence, $B \not<: A$.*

As an example to justify that $B \not<: A$, let $b = [x = 1,\ y = \varsigma(s : \mathscr{S}(B))(\log(s.x) + 1)]$, it is easy to check $b : B$. If $B <: A$, then $b : A$, and in this case we can update the x field in b to a negative integer, say, -1, resulting an object like $b = [x = -1,\ y = \varsigma(s : \mathscr{S}(B))(\log(s.x) + 1)]$. In this object, the invocation of method y will crash since log *is not defined over negative integers.*



(a) (b)

Figure 3: Examples of Object Subtyping



(a) (b)

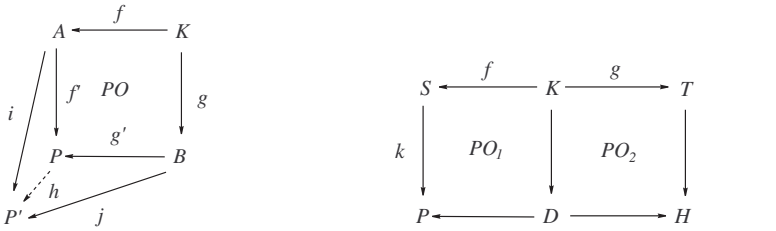
Figure 4: Examples of Object Subtyping

Example 6 *Considering the graphs A and B in Figure 4(a), it is easy to check (similar to the case of example 4) that* $B <: A$. *As an example for this subtyping, an object* $[x = 1,\ y = \varsigma(s : \mathscr{S}(B))(s.x + 1),\ z = 1]$ *which is of type B, can clearly be regarded as having type A.*

Example 7 *Let us revisit the two object types in Figure 2(a). We have* $B \not<: A$, *since we cannot find a premorphism* f *from* $Ba(A)$ *to* $Ba(B)$ *such that* $f(Ba(A))$ *is closed. Similarly, there exists no such a premorphism* g *from* $Ba(B)$ *to* $Ba(A)$ *such that* $g(Ba(B))$ *is closed, so* $A \not<: B$.

One may wonder what kind of type (graph) can be of a subtype of A or B respectively. Any subtype of A must not have a link between methods x and y; and any subtype of B must have a link going from x to y. This is the structural requirement in Definition 11. As a result, object $[x = 1,\ y = 1]$ *cannot be regarded as having the same type with the object* $[x = 1,\ y = \varsigma(s : \mathscr{S}(B))(s.x + 1)]$, *and vice versa. One may contend that this subtyping is too restrictive so that some "good" subtyping instances are not allowed by it; we argue that this is the trade-off in the sense that the strictness of this subtyping can block and prevent potential programming errors, as shown in the next example.*

Example 8 *As the last example, we show how the "free or constrained rectangles problem" described in section 2.1. The (new) types of the free rectangle fRect and the constrained rectangle cRect are depicted as F and C in Figure 4(b). Note that the independence between the height and the width in the free rectangle fRect and their dependency in the constrained rectangle cRect are faithfully shown by the absence and presence of a byh-colored link between methods h and w in F and C, respectively. It is easy to check that* $C \not<: F$. *So if we modify the function ds of section 2.1 by replacing its parameter type FR by the new type F in Figure 4(b), then the call* $ds(cRect)$



(a) Graph gluing definition (b) Direct derivation definition

Figure 5: Definitions of Graph Gluing and Direct Derivation

will be rejected under OTG by the compiler since it does not type-check.

6 Inheritance under OTG

We now turn to the issue of inheritance. Some basic notions in graph grammar are needed before we can define inheritance formally.

6.1 Definitions

Definition 12 *(Type Graph Production)* A ***type graph production*** p is a pair of type graph premorphisms $f : K \to A_1$ and $g : K \to A_2$, where A_1 is called the *left side*, A_2 the *right side*, and K the *interface*. This is denoted as $p = (A_1 \xleftarrow{f} K \xrightarrow{g} A_2)$.

Definition 13 *(Type Graph Gluing)* Given two type graph premorphisms $f : K \to A$ and $g : K \to B$. The ***gluing*** of A and B along K is the ***pushout*** of $K \xrightarrow{f} A$ and $K \xrightarrow{g} B$ in the category formed by type graphs together with type graph premorphisms (Figure 5(a)).

Definition 14 *(Direct Derivation)* Given type graphs P, D, H, a type graph production $p = (S \xleftarrow{f} K \xrightarrow{g} T)$, and a premorphism $k : S \to P$ (called a context map). We say that H is ***directly derived*** from P via p by k, denoted by $P \overset{(p,k)}{\Longrightarrow} H$, if P is the result of gluing S and D along K and H is the result of gluing D and T along K (Figure 5(b)).

Definition 15 *(Unfolding Production and Operation)* A graph production $u = (S \xleftarrow{f} K \xrightarrow{g} G)$ is called an unfolding production if (1) K is a graph consisting of two nodes v_1 and v_2, and $c(v_1) = c(v_2) = sel\,f$; (2) S is a graph consisting of two nodes u_1 and u_2 and an arc t such that $c(u_1) = c(u_2) = sel\,f, c(t) = rec, sr(t) = u_2$, and $tg(t) = u_1$; (3) $f(v_i) = u_i, i = 1,2$; g is a partial morphism with $g(v_2) = s_G$ where s_G is the starting node of G. Given an unfolding production $u = (S \xleftarrow{f} K \xrightarrow{g} G)$, an object graph F, and a premorphism $i : S \to F$, we say F unfolds to P if $F \overset{(u,i)}{\Longrightarrow} P$.

Definition 16 *(Addition Production and Operation)* A graph production $a = (S \xleftarrow{f} K \xrightarrow{g} G)$ is called an addition production, if (1) K consists of only one node v, and $c(v) = sel\,f$; (2) S consists of only one node u, and $c(u) = sel\,f$; (3) $f(v) = u$ and $g(v) = s_G$ where s_G is the starting node of G. Given an addition production $a = (S \xleftarrow{f} K \xrightarrow{g} G)$, an object graph F, and a premorphism $j : S \to F$, we say that P is the result of adding G into F if $F \overset{(a,j)}{\Longrightarrow} P$.

Definition 17 *(Link Production and Operation)* A graph production $l = (S \xleftarrow{f} K \xrightarrow{g} G)$ is called a link production, if (1) G is a graph consisting of two nodes v_1, v_2 and an arc a connecting these two nodes. $c(v_i)$ $(i = 1,2)$ is either a ground type or a $\to$ or a $sel\,f$, and $c(a) = bym$, where m is the color of one of the methods in G; (2) K is a graph consisting of two nodes u_1 and u_2 with $c(u_i)$ is either a ground type or a $\to$ or a $sel\,f$, $i = 1,2$; (3) S is isomorphic to K, that is, f is an isomorphism from K to S; g is an injection with $g(u_i) = v_i$, $i = 1,2$; Given a link production $l = (S \xleftarrow{f} K \xrightarrow{g} G)$, an object graph F, and a premorphism $k : S \to F$, we say that P is the result of embedding G into F if $F \overset{(l,k)}{\Longrightarrow} P$.

Definition 18 *(Inheritance Construction of Object Type Graphs)* Given object type graphs F and G. F is said to be inherited from G if G can be transformed into F through a finite sequence of unfolding operations, addition operations, and link operations.

Definition 19 *(Inheritance)* Given object type graphs S and T, an object of type T can be constructed by inheritance from an object of type S if T is inherited from S.

The central idea here is that inheritance of objects should be guided and guarded by object types. We devise, through some basic graph transformation techniques, an "inheritance" notion on object types, and then use this notion to judge whether an object can be built through inheritance from another object.

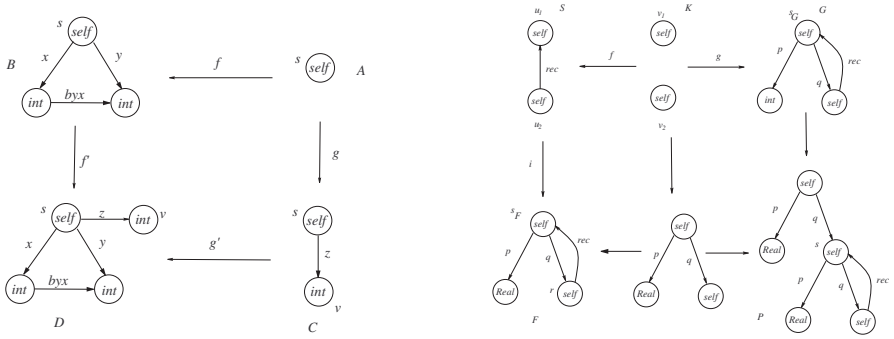
6.2 Examples

We now give some examples to demonstrate the graph operation definitions in the last section.

Example 9 *A graph gluing example is shown in Figure 6(a), where f and g map the only node in A to the $sel\,f$-colored node in B and C respectively, and D is the result of gluing B and C along A. Intuitively, this gluing operation entails the connection of B and C by identifying their starting nodes.*

Example 10 *Figure 6(b) shows an unfolding operation. $F \overset{(u,i)}{\Longrightarrow} P$, where $u = (S \xleftarrow{f} K \xrightarrow{g} G)$, $f(v_i) = u_i$, $i = 1,2$, $g(v_2) = s_G$, $i(u_1) = s_F$, $i(u_2) = r$. As we can see, P can be understood as constructed by deleting the rec-colored arc from F, and then glue the result with a copy of the original F by identifying the starting node of the former with the source node of the rec-arc of the latter.*

Example 11 *We finally show how the "colored point problem" addressed in section 2.2 can be*



(a) Graph gluing (b) Unfolding operation

Figure 6: Examples of Graph Gluing and Unfolding Operation

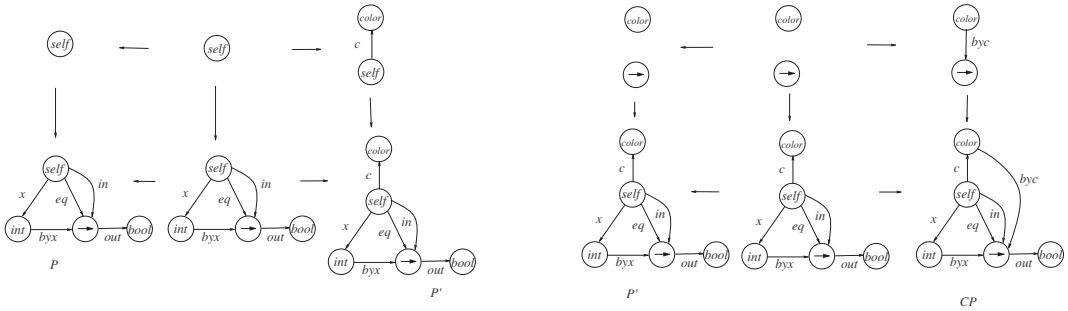


Figure 7: *CP* is Inherited from *P*.

resolved under OTG inheritance and subtyping. The type of points pt and pt_0 *and the type of color point cpt are depicted as P and CP in Figure 7 respectively. We can see that CP is inherited from P (through one addition operation and one link operation), which means that cpt can be constructed by inheritance from pt (or* pt_0*). Moreover, it is easy to check that* $CP <: P$ *under OTG subtyping, which indicates that inheritance and subtyping are congruent in this case. (Note that this contrast with the "inheritance is not subtyping" slogan in the literature which is mainly motivated by this colored point example.) Finally, the crash of* $F(cpt)$ *is prevented since the function F, as defined in section 2.2 and in the literature, does not type-check under OTG typing. Note in its definition,* $F = fun(p : PT)(p.eq(pt_0))$*, with* $PT = ObjectType(MyType)\{x : int, eq : MyType \rightarrow Bool\}$*, p.eq requires an argument of the self type associated with PT,* $\mathscr{S}(PT)$*, but* pt_0 *has type PT, and PT is neither the same as nor the subtype of* $\mathscr{S}(PT)$ *under OTG.*

7 Resolution of the Problems

As examples of OTG subtyping and inheritance, we have demonstrated in the last section that the two problems outlined in section 2 can be successfully resolved under OTG subtyping and inheritance mechanisms. Here, we just summarize some major points.

- OTG subtyping takes into consideration the method interdependencies in objects. An object in which there is no dependence between two methods can never be regarded as having the same type with or a subtype of that of an object in which there is an interdependency between these two methods, and vice versa. The problem addressed in section 2.1 can be naturally resolved in OTG since there is an interdependency between height and width in constrained rectangles, and there is no such interdependency in free rectangles. Consequently, these two kinds of rectangles have different types.
- OTG inheritance replies on basic graph derivations. The fundamental idea in this respect is that inheritance on objects should be regulated using type information of the relevant objects. An "inheritance" relation over object types is first defined using graph derivations and then used to determine whether an object can be constructed by inheritance from another one.
- "Inheritance is not subtyping" has been advocated in the literature for quite a while. Despite that, the mainstream OOP still adheres to the practice that "inheritance indicates subtyping". One of the reasons for this is that without this practice, the software hierarchy built by inheritance would be almost useless. Thus this practice is highly desirable. The"colored point problem" described in section 2.2 is one of the motivating examples that has prompted "inheritance is not subtyping", because otherwise we will face some "method-not-found" error. Under OTG, we give this problem a new solution in the sense that "inheritance indicates subtyping" is retained and "method-not-found" error is avoided.

8 Related Work

Representing object types as directed colored graphs and subsequently addressing the subtyping and inheritance issues by graph transformations is our original idea. It uniquely connects the type theory of object-oriented languages to algebraic graph transformation theory. The foundations of type theory can be found in [Bar92, AC96, Pie02], and the recent results and directions in type theory research are reflected in, for example, [PRB07, Che07, DHC07]. The origin of algebraic graph grammar and graph transformation can be traced back to [EPS73, Ehr78], and [Roz97, EEPT06] present a comprehensive coverage of this research area. For current trends and developments in graph transformation, see for example, the proceedings of GT-VMT and ICGT [EG07, CEM+06].

Incidentally, it is interesting to note that the phrase "type graph" has been used inconsistently in the literature. For example, it is used to denote the disjunctive rational trees in Prolog type analysis and database query algebra [HCC93, Sch01], to facilitate the investigation of quantification in Type Logical Grammar [BS06], and to give types for (some other) graphs in graph transformation study [GL07, EEPT06]. Obviously, none of these relates to the object type graphs introduced in this paper in a clear manner.

9 Final Remarks

Subtyping and inheritance are two major issues in OOP. Although both issues have been studied extensively, problems still persist. Two particular problems, method interdependencies and "inheritance is not subtyping", are identified and subsequently addressed by a graph-computing

(OTG) approach in this paper. It is demonstrated that both problems can be resolved effectively under OTG subtyping and inheritance mechanisms.

Bibliography

[AC96] M. Abadi, L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Bar92] H. Barendregt. Lambda Calculi with Types. In S. Abramsky (ed.), *Handbook of Logic in Computer Science*. Volume 2, pp. 117–309. Clarendon Press, Oxford, 1992.

[BL95] V. Bono, L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of International Conference of Computer Science Logic*. LNCS 933, pp. 16–30. 1995.

[Bru94] K. Bruce. A paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming* 4(2):127–206, 1994.

[BS06] C. Barker, C. chieh Shan. Types as Graphs: Continuations in Type Logical Grammar. *J. of Logic, Language and Information* 15(4), 2006.

[CEM+06] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (eds.). *Proc. of ICGT'06*. LNCS 4178, Springer, 2006.

[CHC90] W. Cook, W. Hill, P. Canning. Inheritance is not Subtyping. In *Proc. of POPL*. Pp. 125–135. 1990.

[Che07] J. Chen. A typed intermediate language for compiling multiple inheritance. In *Proc. of POPL'07*. Pp. 25–30. 2007.

[DHC07] D. Dreyer, R. Harper, M. Chakravarty. Modular type classes. In *Proc. of POPL'07*. Pp. 63–70. 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[EG07] K. Ehrig, H. Giese (eds.). *Proc. of GT-VMT'07*. http://eceasst.cs.tu-berlin.de/, 2007.

[Ehr78] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Applications to Computer Science and Biology*. LNCS 73, pp. 1–69. Springer-Verlag, 1978.

[EPS73] H. Ehrig, M. Pfender, H. J. Schneider. Graph grammars: An algebraic approach. In *IEEE Conference of Automata and Switching Theory*. Pp. 167–180. 1973.

[FHM94] K. Fisher, F. Honsell, J. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nodic Journal of Computing* 1:3–37, 1994.

[GL07] E. Guerra, J. de Lara. Adding Recursion to Graph Transformation. In *Proc. of GT-VMT'07*. 2007.

[HCC93] P. V. Hentenrck, A. Cortesi, B. L. Charlier. Type Analysis of Prolog Using Type Graphs. Technical report, Brown University, Technical Report CS-93-52, 1993.

[LC96] L. Liquori, G. Castagna. A Typed Lambda Calculus of Objects. LNCS 1179, pp. 129–141. Sringer–Verlag, 1996.

[Liq98] L. Liquori. On Object Extension. In *ECOOP'98 Object-oriented Programming*. Lecture Notes in Computer Science 1445, pp. 498–522. Sringer–Verlag, 1998.

[Pie02] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PRB07] P. Permandla, M. Roberson, C. Boyapati. A type system for preventing data races and deadlocks in the java virtual machine language. In *Proc. of LCTES'07*. Pp. 1–10. 2007.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 1. World Scientific, 1997.

[Sak05] M. Sakkinen. Wishes for Object-oriented Languages. In *Proc. of Langages et Modeles a Objets (LMO 2005, invited talk)*. 2005.

[Sch01] K.-D. Schewe. On the Unification of Query Algebras and Their Extension to Rational Tree Structures. In *Proc. of 12th Australasian Database Conference*. Pp. 52–59. 2001.

[Seb07] R. Sebesta. *Concetps of Programming Languages*. Addison Wesley, 8th edition, 2007.