Towards A New Definition of Object Types

Cong-Cong Xing
Fantasia International Inc.
Baton Rouge, LA 70809, USA
congcong.xing@gmail.com

ABSTRACT

The idea of integrating method interdependency information into object types, representing the resulting new types as graphs, and subsequently using algebraic graph transformation techniques to reason the subtyping under this new typing scheme, is proposed.

1. THE PROBLEM

We call a rectangle *free* if its two sides are independent of each other, *constrained* otherwise. Using a syntax similar to that of Abadi-Cardelli's object-calculus [2], an example of free rectangles would be

$$a \stackrel{\text{def}}{=} \varsigma(s\!:\!FR)[x = 1, y = 2, setx = \lambda(i\!:\!int)(s.x \Leftarrow i)],$$

and an example of constrained rectangles would be

$$b \stackrel{\text{def}}{=} \varsigma(s\!:\!CR)[x = 1, y = s.x + 1, setx = \lambda(i\!:\!int)(s.x \Leftarrow i)],$$

where in each case s is the self variable and FR and CF are its types respectively. The method *setx* changes/updates the value of x in an obvious way. Note in a, the two sides x and y are both constants and therefore are independent of each other; in b, the side y *depends on* the side x in the sense that the evaluation of y uses/needs the evaluation of x.

In conventional type systems, such method dependency information is not considered in object types. As a result, the type of the free rectangles and the type of the constrained rectangles are the same, i.e., $FR = CR$ in conventional type systems. Consequently, the following problems may occur.

Suppose we have a function f which works as follows: f takes a free rectangle as argument, changes its side x (just x, not y), and does some more stuff, say, $f \stackrel{\text{def}}{=} \lambda(r : FR)(r.setx(k), ...)$ where k is any constant. Since $FR = CR$, f is actually allowed to take as arguments both free rectangles and constrained ones. While f works just fine for free rectangles (such as a), it may cause problems when taking constrained rectangles, for example b: when f changes the value of side x of b, the value of side y of b is also *implicitly* changed since y depends on x in b. But, the change of the value of y may not be the intention of f and f may not be aware of this change. Consequently, if y is used somewhere in the rest of the code of f, it will inflict subtle and hard-to-find computational errors, and will burden the program verification task.

Dually, suppose we have a function g that takes a constrained rectangle (specifically, side y depends on side x) as argument, changes its *both* sides, and does some more things, say $g \stackrel{\text{def}}{=} \lambda(r : CR)(r.setx(k), ...)$ where k is any constant. (Here, since g knows that its argument's side y depends on the side x, g only needs to change the side x in order to change both x and y.) Again, because $FR = CR$, g is actually allowed to take as arguments both constrained rectangles and free ones. This time, while g works just fine for constrained rectangles, it may cause potential problems when taking a free rectangle as argument, for example a: when g changes the side x of a, the side y of a is *not* changed (although g expects y to be changed). Similarly to the case of f, if y is used somewhere in the rest of the code of g, it will create computational bugs and give unwanted burdens to program verifications.

2. THE IDEA

Based on the problem described above, we attempt to integrate the method interdependency information in objects into their types, and thereby propose a new way for object typing. Specifically, for any object type (in the syntax of object-calculus)

$$[l_1\!:\!\sigma_1, \ldots, l_n\!:\!\sigma_n],$$

we attache a set L_i of method labels to each method l_i to indicate the method interdependency information, with the following stipulations:

- If $L_i = \emptyset$, then it indicates the l_i depends on no other methods.
- If $l_j \in L_i$, then it indicates that l_i depends on l_j.
- If $l_j^? \in L_i$, then it indicates that l_i may depend on l_j and may not depend on l_j. (This is the case modeled by the object types in current type systems: i.e., for each method (label) l_i, we do not know/care if l_i depends on l_j for some method l_j.)

Roughly speaking, the first two cases give us a means to further refine the current object types, and the last case allows us to recapture the notion of current object types. For example, the types of a and b, under this newly proposed type representation, would be

$$A \stackrel{\text{def}}{=} [x(\emptyset)\!:\!int,\ y(\emptyset)\!:\!int] \quad \text{and} \quad B \stackrel{\text{def}}{=} [x(\emptyset)\!:\!int,\ y(\{x\})\!:\!int],$$

respectively. Note that the fact that y depends on x in b is signified by the set $\{x\}$ after the method (label) y in B. The following type

$$C \stackrel{\text{def}}{=} [x(y^?)\!:\!int,\ y(x^?)\!:\!int]$$

would be a supertype of both A and B. That is, $a{:}A <: C$ and $b{:}B <: C$.

Due to the newly added information into object types (and thus the increased complexity) and in order to carry out the subtyping analysis effectively under the new typing scheme, we can represent the object types as colored, directed graphs. For example, the types of a and b just described above can be denoted as A and B in Figure 1. Note that the fact that y depends on x in b is indicated, in graph B, by an edge colored by byx.

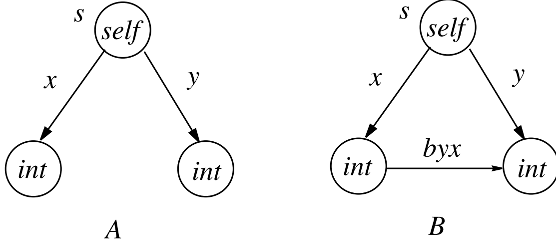

Figure 1: Object Type Graphs of a and b.

The type C above can be depicted as graph C in Figure 2. The fact that x (or y) may and may not depend on y (or x) in an object of this type is shown by the two dotted edges colored by byx and byy respectively. We expect to have $A <: C$, $B <: C$, $A \not<: B$, and $B \not<: A$ (both A and B are subtypes of C, A is not a subtype of B, and B is not a subtype of A either.)

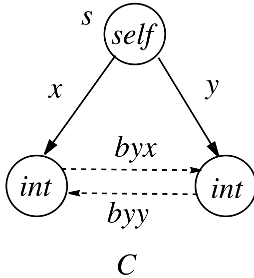

Figure 2: Object Type Graph for type C.

As for exactly how to define and conduct a subtyping analysis under this graphical notation of object types, we can borrow the techniques established in the are of algebraic graph transformations which stared in 1970s and has gained a tremendous momentum recently [3, 5, 4]. (Most recent results on graph transformations can be found at the International Conference on Graph Transformations: http://www.cs.le.ac.uk/events/icgt2008/.) For example, we can try to define the notion of graph morphisms among object type graphs and then show that these morphisms together with all object type graphs form a category. From there, we can examine the properties of pushouts in this category to reformulate the graph transformation rules towards the goal of establishing a desired subtyping relation among object type graphs.

3. (PREDICATED) PROS AND CONS

At this point, we can foresee that the proposed new typing scheme increases the flexibility over the current type systems.

- On the one hand, it allows users to specify more (detailed) types as needed; for example, we can replace the types FR and CR in functions f and g by the new types A and B as shown in Figure 1, then, under this new typing system, f will not be able to take a constrained rectangle as argument and g will not be able to take a free rectangle as argument, and thus the problem would be easily resolved.
- On the other hand, it retains the abstraction capability possessed by the current object type systems. For example, suppose we would like to write a function h which takes a rectangle as argument and performs something that has nothing to do with the interdependencies between the sides of the rectangle (say, just print out the values of the two sides), in other words, the functionality of h does not depend on the interdependencies of two sides of the argument so there is no need for h to care about the method interdependency information of its argument. In this case, we can just specify the type of the parameter of h to be C (as shown in Figure 2).

A potential difficulty of this work is that, typically, graphs are not built, searched, and checked for desired morphisms (e.g. isomorphism) in linear time. But, in this regard, we can try to build simpler approximations of the graphs that can be manipulated in linear time. A successful example along this line which can be studied is the XDuce projet of Benjamin Pierce [1].

As a characteristic summary, we might say that the new typing scheme allows us to assert the following: (1) a free rectangle is not a constrained rectangle, and (2) a constrained rectangle is not a free rectangle either, but (3) both are rectangles; whereas the current type systems do not allow us to do this.

4. REFERENCES

[1] *http://www.cis.upenn.edu/ bcpierce/papers/index.shtml.*

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

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

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

[5] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph grammars: An algebraic approach. In *IEEE Conference of Automata and Switching Theory*, pages 167–180, 1973.