A Pure Calculus of Subtyping, and Applications (Outline)

Luca Cardelli
Digital Equipment Corporation, Systems Research Center
130 Lytton Avenue, Palo Alto CA 94301, USA
luca@src.dec.com

Abstract

One of the problems in understanding object-oriented languages is understanding their type systems, e.g. making sure that they are sound. To this end, I propose a *typed* foundation for object-oriented languages, based on a small typed λ-calculus with polymorphism and subtyping.

1 Motivation

In recent years we have seen a flourishing of ideas and techniques both in the design and in the study of typed object-oriented languages. New languages and language features are proposed at every turn, and new semantic models and semantic interpretations closely follow.

While, on one hand, one should be gratified by such richness, I cannot help feeling also a bit embarrassed. New mechanisms are justifiably proposed out of necessity, to remedy deficiencies of existing mechanisms. But, eventually, one reaches a point of diminishing returns, where the convenience of additional mechanisms is overshadowed by their added complexity. How many good ideas can there really be?

This kind of exploration should eventually be replaced by consolidation, and *now* may be a good time. In this respect, I would like to strike against two common attitudes. One is the assumption that we understand existing languages well enough, so we can go ahead and create more complex ones. I think it has not been proved yet (nor disproved) that object-oriented programming as currently intended is a "good thing". It is conceivable that even basic features such as *self* will eventually be considered too subtle and powerful for robust software engineering (or verification), and should be abandoned. Some features will of course survive, possibly becoming more general. Consolidation does not mean oversimplification; necessary distinctions must be made, e.g. between types and classes. But do we need both prototypes and multiple inheritance at once? How can we tell when a language is powerful, as opposed to "just too complicated"?

The other objectionable attitude is of a more technical nature. The semantics of typed object-oriented programming has been so far explained in terms of denotational models, and here too we have seen a variety of models and an embarrassing richness of interpretation techniques. In all cases, though, a *typed* object-oriented language is translated into some *untyped* λ-calculus (the language of the model); a typing soundness theorem then must be proven. I think this is a rather indirect and uninformative approach, from a typing perspective, and leads to too many arbitrary choices. Many of the subtle problems we confront these days are in the typing of object-oriented languages (as well as in their meaning). A proof of typing soundness in a denotational model may show that the type rules of an object-oriented language are sound, but I don't think it shows *why* they are sound. What are the essential properties of all these models and interpretations that make the type rules sound?

2 A pure calculus of subtyping

The central question for me is: what is the smallest typed formal system that captures the essence of object-oriented programming? Let us temporarily call this hypothetical system *TFS*. I think one should codify the crucial properties of denotational models (or just our plain intuitions) into *TFS*, and then give meaning to object-oriented languages by a *type-preserving*, *subtype-preserving*, and *meaning-preserving* translation into *TFS*. If we can do this, then we will be able to say that the typing and equational rules of *TFS* capture the essence of typed object-oriented programming.

I have my share or responsibility for producing overcomplex formal systems, but recently I have been investigating a very simple one, with the aims explained above. This system, called *F<:*, is described in [Cardelli Martini Mitchell Scedrov 91] and, just to show its compactness, here is the complete syntax of types:

A,B ::=	Types:
X	type variable.
Top	the supertype of all types.
A→B	function space.
∀(X<:A)B	bounded quantification.

I am not yet claiming that this is the "right" minimal formal system, but certainly I think it is on the right track. The first indication is that many common constructions, such as *fixed-size records*, can be encoded and their type rules can be derived. More significantly, *extensible records*, for which many complex axiomatizations have been proposed, can also be encoded. (Extensible records were investigated for their relevance to functional and imperative update.)

3 An applied calculus of subtyping

Trying to get closer to our goals, we can define an extended calculus, *F<:ρ* (*F<:* with rows), that has extensible records already built-in [Cardelli 91]. This extended calculus is independently axiomatized with its own type, subtype, and equality rules. The type structure of this calculus is as follows:

L	Finite sets of labels $l_1 ... l_n$.
A,B ::= ...	Types as in *F<:*, plus:
Rcd(R)	record type over a *row type R* (a list of labelled fields).
R ↑L→B	functions from rows in *R* lacking *L*-labeled fields to values.
∀(X ↑L)B	quantification over a row type *X* lacking *L*-labeled fields.
R ::=	Rows types:
X	row type variable (bound by some *∀(X ↑L)*).
Etc	the empty row type (it lacks *L*-labeled fields for any *L*).
l:A,R	row type with an initial field of label *l* and type *A*, followed by a row type *R* with no *l*-labeled field.

The extra richness and convenience of this calculus is however an illusion. We can prove that there is a translation from *F<:ρ* to *F<:* that preserves typing, subtyping, and equality. Hence *F<:ρ* is in principle no more complex than to *F<:*, although it has a much more appealing notation.

This *F<:ρ* calculus does not yet resemble an object-oriented language, but we can express many object-oriented techniques, especially through clever uses of recursive types (that can be added to both *F<:* and *F<:ρ*) and of extensible records. *Record concatenation*, an always troubling subject related to multiple inheritance, can be encoded as well, following R•my. With recursion and higher-order features we can also emulate *F-bounded quantification*, and we can capture all the crucial features of my (much larger) Quest language.

4 Understanding the type structure of object-oriented languages

The next step is to consider some semi-realistic object-oriented language and attempt to give it meaning in *F<:ρ*. An interesting case study is the following language, originally due to Kim Bruce, which combines two troublesome feature: the type *Self*, and updating of instance variables.

This language is based on a distinction between objects and classes (the latter being object-generators), with a corresponding distinction between object-types and class-types. Classes can be extended and overridden; objects, generated from classes, can be updated and can receive messages.

$Object\{v_j:S_j \mid (Self)m_i:T_i\}$	an object type with *instance variables* $v_j:S_j$, and *methods* $m_i:T_i$ where the type *Self* may appear in the T_i.
$Class\{v_j:S_j \mid (Self)m_i:T_i\}$	a class type, similarly.

class{v_j=d_j | (self:Self)m_i=e_i}	a class value with instance variables v_j=d_j, and *methods* m_i=e_i where the type *Self* and the value *self:Self* may appear in the e_i .
c with {v=d | (self:Self)m=e}	a class *c* extended with (e.g.) a new variable *v* and a new method *m*.
c but { (self:Self)m=e}	overriding an existing method of class *c*.
new c	creating an object from a class.
o←v	extracting a instance variable from an object.
o←m	extracting a method from an object.
o gets {v=d}	updating a variable *v* of object *o* (yielding a modified copy of *o*).

This object-oriented language is independently axiomatized with its own typing, subtyping, and equivalence rules; these strongly resemble the rules of a Simula-style language.

The interesting point is that for this language, and for similar ones, it is possible to find a translation into $F_{<:}^{\rho}$ that again preserves typing, subtyping, and equality. As a corollary, we obtain that the non-trivial type system of our object-oriented language is sound (since $F_{<:}$ is sound).

5 Conclusions

In conclusion, I argue that we should be looking for *typed semantics* of object-oriented languages, given by translation into small typed λ-calculi. The advantages of this approach are that (1) the translation process "explains" the type rules of the source language in terms of more fundamental type rules of the target calculus, and (2) the target calculus, being small, has fewer, cleaner and more powerful features, and relatively simple models.

If it turns out that this kind of translation is practically unfeasible for some particular object-oriented language, it may mean that we have the wrong approach. But, as we were discussing at the beginning, it may also be a mesaure of the fact that the language is "just too complicated".

6 References

[Cardelli Martini Mitchell Scedrov 91] L.Cardelli, J.C.Mitchell, S.Martini, A.Scedrov: An extension of system F with subtyping, Proc. TACS'91, LNCS 526, Springer-Verlag.

[Cardelli 91] L.Cardelli: Extensible records in a pure calculus of subtyping, to appear.

(Additional references can be found in the papers above.)