J. Functional Programming 4 (2): 125–126, April 1994

Type systems for object-oriented programming

Functional programming has had a broader impact on computing than the design of a few specific functional languages. One of its contributions is a deeper understanding of type systems. This special issue of *The Journal of Functional Programming* focuses on the difficult and important question of type systems for object-oriented programming.

This issue had an unusual gestation. When I heard of Kim Bruce's expository paper on a type system for object-oriented programming, I immediately invited him to submit it for publication in *JFP*. Papers by Pierce and Turner and by Abadi then came to my attention, and I solicited them as well. By this point we had a bonus-size special issue, without the usual mechanism of a call for papers. Mitchell Wand graciously provided an introduction that carefully compares the approaches of the three papers.

This issue provides an overview of the state-of-the-art, but it is certainly not the last word. Further submissions on this topic are most welcome.

An extra-size issue requires extra work. Thanks are due to the contributors and referees for the effort they have put into producing an extraordinary issue.

—PHILIP WADLER

Introduction

Object-oriented programming (OOP) has become one of the cornerstones of modern programming methodology. Yet there is little agreement on what object-oriented programming is. Different object-oriented languages typically implement different collections of facilities, and heated discussions of which facilities are necessary for true object-oriented programming flare up regularly.

In the light of these discussions, this issue of *JFP* presents three papers that study the theoretical bases of object-oriented programming. These papers illustrate the variety of choices that can be made in the design of a theory of OOP.

Kim Bruce's paper, 'A Paradigmatic Object-Oriented Programming Language: Design, Static Typing, and Semantics', seeks to model via denotational semantics as many features of conventional object-oriented languages as is possible within the functional framework. He defines a language that supports classes, objects, methods, hidden instance variables and inheritance. He presents static typing rules for the language, and then gives a model for his types using PERs. He shows the soundness of the typing rules by giving a denotational semantics, and showing that the semantics is sensible: if a phrase has static type σ, then its denotation is a value

of type σ. This enterprise is complicated by the necessity of dealing with the **self** keyword, which is modelled using recursive types. Instance variables, however, are modelled using existential types.

By contrast, the paper by Benjamin Pierce and David N. Turner, 'Simple Type-Theoretic Foundations for Object-Oriented Programming', seeks to discover what kind of lambda-calculus is need to model OOP. They show that the theory $F^{\omega}_{\leq}$, a polymorphic lambda-calculus with existential types and higher-order bounded quantification, is sufficiently powerful to model encapsulation, message passing and inheritance with **self** and **super** without introducing recursive types. Unlike Bruce, they do not introduce an explicit object language, but consider object-oriented facilities as syntactic sugar for terms of $F^{\omega}_{\leq}$. Soundness of typing and subtyping rules, and a subject-reduction theorem, are obtained *a fortiori* from the known results for $F^{\omega}_{\leq}$.

The last paper in this issue, 'Baby Modula-3 and a theory of objects', by Martin Abadi, takes a still different approach. Abadi's goal is to find the smallest object-oriented language that will illustrate the theoretical difficulties involved in understanding OOP. Here he takes a functional subset of Modula-3. Unlike the others, this language is object- rather than class-based, and uses delegation or extension rather than inheritance. His treatment of **self** is different from either of the other papers; it can perhaps be described best as being primitive in the theory. A subject-reduction theorem is proved. Abadi gives a denotational semantics for his language by interpretation in a fairly standard model of *untyped* lambda calculus. He then gives an interpretation of types as ideals in this model, with recursive types modelled using the usual metric space on ideals. The interpretation of object types is subtle, so as to accommodate subtyping correctly. He then shows that the typing and subtyping judgements of his theory are sound in this model.

These papers are long and often subtle, but they will reward the reader who takes the time to understand their details.

—MITCHELL WAND

https://doi.org/10.1017/S0956796800001027 Published online by Cambridge University Press