Theoretical Aspects of Object-Oriented Programming

Types, Semantics, and Language Design

Draft of January 18, 1993

edited by

Carl A. Gunter and John C. Mitchell

To be published in 1993 by MIT Press in the **Foundations of Computing Series.**

Theoretical Aspects of Object-Oriented Programming

Foundations of Computing
Michael Garey and Albert Meyer, editors

Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks, Frank Thomson Leighton, 1983

Equational Logic as a Programming Language, Michael J. O'Donnell, 1985

General Theory of Deductive Systems and Its Applications, S. Yu Maslov, 1987

Resource Allocation Problems: Algorithmic Approaches, Toshihide Ibaraki and Naoki Katoh, 1988

Algebraic Theory of Processes, Michael Hennessy, 1988

PX: A Computational Logic, Susumu Hayashi and Hiroshi Nakano, 1989

The Stable Marriage Problem: Structure and Algorithms, Dan Gusfield and Robert Irving, 1989

Realistic Compiler Generation, Peter Lee, 1989

Single-Layer Wire Routing and Compaction, F. Miller Maley, 1990

Basic Category Theory for Computer Scientists, Benjamin C. Pierce, 1991

Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist, Andrea Asperti and Giuseppe Longo, 1991

Semantics of Programming Languages: Structures and Techniques, Carl A. Gunter, 1992

Theoretical Aspects of Object-Oriented Programming

Types, Semantics, and Language Design

Carl A. Gunter and John C. Mitchell

The MIT Press
Cambridge, Massachusetts
London, England

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

(Cataloging-in-Publication Data)

Contents

Series Foreword

Theoretical computer science has now undergone several decades of development. The "classical" topics of automata theory, formal languages, and computational complexity have become firmly established, and their importance to other theoretical work and to practice is widely recognized. Stimulated by technological advances, theoreticians have been rapidly expanding the areas under study, and the time delay between theoretical progress and its practical impact has been decreasing dramatically. Much publicity has been given recently to breakthroughs in cryptography and linear programming, and steady progress is being made on programming language semantics, computational geometry, and efficient data structures. Newer, more speculative, areas of study include relational databases, VLSI theory, and parallel and distributed computation. As this list of topics continues expanding, it is becoming more and more difficult to stay abreast of the progress that is being made and increasingly important that the most significant work be distilled and communicated in a manner that will facilitate further research and application of this work. By publishing comprehensive books and specialized monographs on the theoretical aspects of computer science, the series on Foundations of Computing provides a forum in which important research topics can be presented in their entirety and placed in perspective for researchers, students, and practitioners alike.

Michael R. Garey
Albert R. Meyer

Theoretical Aspects of Object-Oriented Programming Languages

Introduction

This volume comprises fifteen chapters, by selected authors, on theoretical aspects of object-oriented programming languages. The focus is on type systems and semantic models, and how advances in these areas can contribute to new language designs. The collection is divided into five parts: Part I. **Objects and Subtypes**, Part II. **Type Inference**, Part III. **Coherence**, Part IV. **Record Calculi**, and Part V. **Inheritance**. The chapters are organized approximately in order of increasing complexity of the language constructs they consider. Put briefly, the collection begins with variations on Pascal- and Algol-like languages, develops the theory of illustrative record object models, and concludes with research directions for developing a more comprehensive theory of object-oriented programming languages.

Part I discusses the similarities and differences between "objects" and algebraic-style abstract data types, and address general problems associated with subtyping (or subclasses) in the presence of operations that may be applied to objects of more than one type.

Parts II – IV, which form the core of the collection, are concerned with what may be called the *record model* of object-oriented languages. More specifically, these chapters discuss static and dynamic semantics of languages with simple object models that include a type or class hierarchy, but do not explicitly provide what is often called "dynamic binding" or "dynamic method lookup." Part II develops the record model incrementally, beginning with a simple extension of the record operations from the language ML. The denotational semantics of these languages are considered in Part III, with more elaborate record object models and more precise connections with object-oriented methodology developed in Part IV.

Extensions and modifications to record object models are considered in Part V. These chapters bring us closer to the full complexity of practical object-oriented languages. However, the complete theoretical underpinnings of the language features discussed in Part V remain a topic for future research.

Objects and Subtypes

The first two chapters of the volume, which form Part I, are also the first two works chronologically. Both are by John Reynolds and they discuss themes that are echoed throughout the rest of the volume. Chapter 1 reprints an often overlooked paper on data abstraction discussing the difference between the algebraic view, in which data types consist of a type or sort and associated functions, and what Reynolds called *procedural data structures,* now better known as *objects.* The chapter uses a case study to illustrate the tradeoffs between these two approaches to data abstraction. This is the idea underlying the philosophies of data abstraction that differentiate the designs of languages such as Ada and ML from those of languages that emphasize an *object-oriented* approach.

To understand the work in Chapter 2 it is helpful to appreciate its context. The original paper was presented at a conference on semantics-directed compiler generation and was intended to provide a challenge to researchers in this area. The primary topic is that of *subtypes.* The motivating example is a familiar one: the type of integers is to be viewed as a subtype of that of reals. Given this, one asks what relationship, if any, should hold between the operation of addition on integers and reals. Given a mapping from subclass (integer) to superclass (real), should it be the case that the commands

```
x := integer-to-real(m) + integer-to-real(n)
```

and

```
x := integer-to-real(m+n)
```

have the same meaning? This question illustrates two of the most fundamental themes in studying the semantics of subtypes. The first of these is the notion of a *coercion* map. Such maps can be *explicit,* as in the programs above, or *implicit,* as when the complier can infer the coercion without the need for programmer annotations. The second is the notion of *coherence.* The equivalence of the programs above is a typical coherence question. A slightly more general expression of the condition says that if there are a coercions c_1, c_2, c_3 having types $r \to s$ and $s \to t$ and $r \to t$ respectively, then the composition of the coercions c_1 and c_2 is a coercion from r to t that is equal to c_3. Chapter 2 provides a categorical treatment of the issue of the semantics of subtypes for an ALGOL-like language. Reynolds' concluding remarks in this chapter also include the description of a system of subtyping on records, variants, and function spaces that anticipates many of the ideas in subtyping that would become a focus of study in works such as those in Part III, which expand the discussion to languages that include *parametric polymorphism.*

Types and Type Inference

One of the central objectives of studies in this volume is ensuring desirable characteristics of the run-time behavior of programs. A valuable class of such characteristics can be ensured through the imposition of a *phase distinction* between compile-time and run-time if this separation exploits a *static type discipline* to filter programs that may have undesirable run-time properties. In programs that do not impose such a discipline, programming is less restrained, but fewer assumptions can be made about run-time errors. An example of a language with no type discipline (an *untyped language*) is considered in Chapter 13, but most of the chapters of this volume focus on typed languages.

There are several gradations in typing systems one may impose. One approach is to impose a system of programmer type annotations that is then checked by a decidable or semi-decidable type verification algorithm. On the other hand, one may define a type discipline with the property that a *most general* type can be decidably *inferred* for a program even when the program has few or no explicit (programmer) type annotations. The practical utility of such type disciplines has been demonstrated by the language ML [MTH90, MT91, Pau91], which uses the Hindley-Milner type inference algorithm [Hin69, Mil78] to generate type schemes for programs with very few type annotations. The development of similar systems for languages with object-oriented features such as flexible record types has been one focus of research, and it is the topic of Part II of this volume.

The study of type inference for systems with more flexible record types began with a seminal paper of Mitchell Wand [Wan87]. Part II consists of three works that forged further directions toward a system of type inference for languages with flexible records. The first of these, Chapter 3, offers a proposal by Didier Rémy of a type inference system for a language extending ML. Rémy's type system allows functions to be applied to all records that have at least some minimum set of components required by the function body. This problem is essentially the same as the problem of type-checking programs that manipulate objects, allowing any object to be replaced by another with at least as many methods. Chapter 4 employs a similar system and demonstrates how some of the fundamental concepts of object-oriented languages can be encoded in a system with type inference. In particular, Wand focuses on the concepts of *instance variable* and *inheritance.* The notion of an instance variable is modelled functionally in that chapter as in the most of the rest of this volume using record fields. The result is a clear analysis in a type system that is easy to understand and has very good properties. This work should be compared with other treatments of inheritance that appear as the central topic of Part V.

Chapter 5 describes an approach of Peter Buneman and Atsushi Ohori to achieving flexible record types in a system with type inference. Their system is based on the notion of *conditional types* which annotate type variables with restrictions, an idea reminiscent of *bounded quantification,* which is studied in subsequent chapters. Unlike the systems using bounded quantification, however, Buneman and Ohori's system has inferred principle types. This chapter also discusses the programming of database primitives in the proposed system.

Semantics and Coherence

While the chapters forming Part II all consider ways to make a record system more flexible through quantification over type variables, the common theme of Part III is to combine such quantification with the idea of subtyping along the lines discussed at the end of Chapter 2. A central theme of Part III is the idea of *coherence.* Although coherence was already discussed in Chapter 2, the form in which it is studied in Part II involves a number of new ideas. Prior to Reynold's work most of the work on the semantics of subtypes focused on interpreting subtypes as subsets. Chapter 2 breaks with this idea and proposes that there are important instances in which subtypes should not be considered subsets from a semantic perspective. Chapter 6, by Kim Bruce and Peppe Longo, proposes a new idea for understanding the semantics of subtyping in a way that has the intuitive flavor of interpreting subtypes as subsets while also addressing the problem that the coercion from a subtype into a supertype may not be an injection (1-1 mapping). Their idea is to use *partial equivalence relations* (*PER*'s) over Kleene's applicative algebra of partial recursive functions to serve as the interpretations of types. The coercion of a subtype into a supertype could then be given as the 'inclusion' between PER's in a sense compatible with the idea that the coercion need not be an injection. Proving that the corresponding meaning function is well-defined—the coherence problem for the PER interpretation—is one of the primary results of Chapter 6. A different approach to coherence for essentially the same language is pursued in Chapter 7, a chapter by Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Rather than interpreting the system of interest directly, they translate the calculus syntactically into an extension of the Girard-Reynolds polymorphic λ-calculus that has an explicit notion of coercion, but no subtyping relation. The translation is defined by induction on the height of a derivation of a type for a term; to give a well-defined semantics, it is then shown that the translation is coherent in the sense that no matter which derivation of a given term is used for the translation, the result is always the same up to provable equality in the target calculus. This shows that any model of the target calculus can be seen as a model of the calculus with subtyping; since the models of the former are fairly well understood, this provides a simple system for generating models. The proofs of coherence in Chapters 6 and 7 are among the most technically intricate ones that appear in this volume. In Chapter 8, the final chapter of Part III, Pierre-Louis Curien and Giorgio Ghelli address the question of how to prove such results using term rewriting techniques. Their exposition is based on a calculus called $F_{\leq}$ (pronounced "F sub") that simplifies the syntax of the language being studied without changing the central issues involved in proving coherence for languages such as those considered in the two previous chapters. Rather than defining rewriting directly for proofs in $F_{\leq}$, Curien and Ghelli define an auxiliary type system in which terms carry complete information about their typing derivations. A normalizing rewriting system is defined for terms in this language; properties of this rewriting system can then be related to the proofs of $F_{\leq}$ in a way that implies the desired coherence result.

Varieties of Record Calculi

In presenting semantic models in Part III, several different illustrative languages are used. A common core of all of these is a typed lambda calculus with polymorphism and subtyping. Although originally formulated by Cardelli and Wegner as part of a language called *Bounded Fun* [CW85], the name that has now become standard for this language is $F_{\leq}$. The polymorphic lambda calculus with subtyping is an extension of Girard's original polymorphic lambda calculus [Gir71, Gir72], which he called *System F*. The difference between $F_{\leq}$ and F is that $F_{\leq}$ has a subtype relation on types, which is often written using the symbol $\leq$ (or, alternatively, $<:$ or $\subseteq$).

While $F_{\leq}$ provides a framework for studying object-oriented languages, many important features are apparently missing from $F_{\leq}$. The most notable is that $F_{\leq}$ does not have any constructs resembling objects. Therefore, to represent the basic concepts of object-oriented languages in a simple way, it seems natural to add some form of objects to $F_{\leq}$. Beginning with Cardelli's seminal paper on records and subtyping [Car88] (which originally appeared in conference form in 1984), it has been traditional to regard objects as derived from record structures in some way. Therefore, a substantial portion of the research on foundations of object-oriented languages is concerned with record operations that could be added to $F_{\leq}$. In this volume, this line of work is represented by first three chapters of Part IV.

The first chapter of Part IV, written by Luca Cardelli and John Mitchell, begins by summarizing a number of calculi of record operations that had been developed prior to the original publication of this chapter, and then presents a calculus of record operations that subsumes most of the prior proposals. The main operations on records in this chapter are (i) add a field, or component, to a record, (ii) remove a field of a record, and (iii) select the value of a specified component of a record. Along with these operations, the main focus of the paper is a form of polymorphism over record types that allows a single function to manipulate all types of records sharing certain specified fields. The importance and relevance of this form of polymorphism may be illustrated by example. In a program where there are several kinds of two-dimensional (geometric) points, it would be useful to be able to move all, or almost all, kinds of points using the same function. Thus we would like a single `move` function to be applicable to ordinary points with only `x,y`-coordinates, colored points having an additional `color` field, and other extensions of the basic class of points. This was originally addressed in the language *Bounded Fun* of [CW85], however, as discussed in Section 6 of Chapter 6, *Bounded Fun* has significant limitations. These are lifted in Chapter 9.

One limitation of the record operations in Chapter 9 is that it is impossible to type concatenation of records in a direct manner. (This is discussed in Section 4.4 of Chapter 9.) One programming situation that calls for record concatenation is relational databases, where it seems natural to represent a relation as a finite sets of records. The join of two relations is obtained by concatenating all pairs of records (one from each relation) that agree at appropriate fields. A straightforward approach to record concatenation is presented in Chapter 10, by Didier Rémy. This chapter gives a general method for defining record concatenation in any language that allows records to be extended with new fields and that has sufficient functional programming capabilities. This method also shows how multiple inheritance can be reduced to single inheritance, in principle.

A surprising feature of $F_{\leq}$, explained in detail in Chapter 11 by Luca Cardelli, is that a range of record operations may in fact be encoded directly in $F_{\leq}$. In this encoding, records are represented as a certain form of polymorphic function. While the encoding is probably too inefficient to be useful in practice, it shows that, from a semantic or theoretical point of view, a substantial range of record operations may be represented in the "basic" language $F_{\leq}$ with no record operations at all.

While Part II is concerned with the existence of algorithms for type checking, or inferring types for expressions containing record operations, these algorithmic concerns are absent from the first three chapters of Part IV. The final chapter of Part IV, written by Benjamin Pierce, shows that there is no typing algorithm for $F_{\leq}$. This holds regardless of whether the types of variables and formal function parameters are written explicitly in programs. The reason is that the subtype relation itself is undecidable. While this shows that there would be significant implementation problems in building a practical programming language around full $F_{\leq}$, the concluding section of the chapter gives some suggestions for avoiding this algorithmic problem in practice.

Inheritance

Records provide the most basic feature of objects, aggregation of functions and data. In addition to records, the record calculi of Parts II – IV also provide forms of subtyping and parametric polymorphism that allow programs to operate uniformly over all records (or objects) that share common properties. Roughly speaking, the languages considered in Parts II – IV are more concerned with operations on objects than the ways that objects are created or defined. In Part V, which contains the final three chapters of the collection, the focus changes to inheritance mechanisms, which are used to define objects or classes of objects incrementally by specifying how a new object should differ from those already defined.

The first chapter of Part V, written by Samuel Kamin and Uday Reddy, presents a series of model languages that illustrate the difference between Pascal-style records and the form of object and class declarations that appear in languages such as SMALLTALK [GR83], Eiffel [Mey92] and C^{++} [ES90]. The first language, called *ObjectTalk,* provides only a simple form of object with instance variables and methods. The semantic interpretation of an *ObjectTalk* object is a "message environment" binding message names to their methods. This is essentially the same as a record associating a value with each component (or field) name. However, as pointed out in the chapter, an object denotes a recursively defined message environment (or record), since the methods of an object may refer to each other recursively. In effect, the language *ObjectTalk* shows how recursive definitions allow us to model objects as records.

The subsequent languages of Chapter 13 are *ClassTalk, InheritTalk* and *SmallTalk,* the final language being an abstraction of SMALLTALK-80. The first, *ClassTalk,* has a form of class, which is a function that creates objects sharing a common set of methods. Since most common object-oriented languages are class-based, this brings us closer to the form of language generally encountered in practice. However, *ClassTalk* is easily translated into *ObjectTalk.* (This may again be interpreted as a positive result about the record model.) *InheritTalk* adds a form of inheritance to *ClassTalk,* allowing one class to be defined from another. The final example language, *SmallTalk,* differs from *InheritTalk* in that when class A inherits from class $B,$ and message to an object of type A results in invocation of a method of class $B,$ any reference to *self* in this method leads to the method of class A rather than class B. This rather technical-sounding difference is important in object-oriented programming, and explained in more detail in the chapter. While most chapters use typed *functional* languages, with no notion of global store, to illustrate the main ideas, Chapter 13 uses untyped imperative languages. These are given denotational semantics using the traditional Scott-Strachey approach, which reduces imperative programs to functional programs with store operations.

The remaining chapters are Chapter 14, by William Cook, Walter Hill and Peter Canning, and Chapter 15, by John Mitchell. Both are concerned with inheritance, Chapter 14 in a class-based language and Chapter 15 in a delegation-based language. The main difference between class- and delegation-based languages is that in class-based languages, every object is created by a class and the class determines the implementation of each method. In contrast, delegation-based languages allow one object to be created by "cloning" another, with the implementation of each inherited method determined by the object that is cloned. While these two approaches to inheritance may sound very different, the basic nature of inheritance, as a mechanism for re-using methods, is very similar in both cases. (Some comparison between class- and delegation-based languages may be found in [Ste87].)

Chapters 14 and 15 are both concerned with typing and its relation to inheritance. A significant issue in both papers is that (stated in class-based terms), if one class is defined from another by inheritance, it may be desirable to change the types of some of the inherited methods. This is significant since it may make objects created by the inheriting class unusable as substitutes for objects for the base (or super) class. For example, suppose we have a class `Point` of points with `x,y`-coordinates, and some associated methods, including an equality method `eq`. If we define a class `Colored_Point` of points with `x,y`-coordinates and `color`, inheriting `eq` and other methods, then it would be natural to redefine `eq` for colored points. Specifically, if we compare two objects of class `Point`, they should be equal if they have the same `x,y`-coordinates. In contrast, two objects of class `Colored_Point` would only be equal if they have the same `x,y`-coordinates *and* `color`. In this situation, the `eq` method of a `Colored_Point` may only be used for comparison with another `Colored_Point`.

We may understand the title, "Inheritance is not subtyping," of Chapter 14 using the example of points and colored points. The definition of subtyping that is used is that `A` is a subtype of `B` precisely if any object of type `A` mau be used in any context where an object of type `B` is required, without producing a type error. Suppose `Point` and `Colored_Point` are as described above, with `p` an object of class `Point` and `q` an object of type `Colored_Point`. Then clearly `p eq p` is sensible, since the equality method of `p` allows `p` to be compared with any other `Point`, including itself. If `Colored_Point` were a subtype of `Point`, then we would expect to be able to replace either occurrence of `p` by `q` without causing a type error. The expression `p eq q` does not cause any problem, since the `eq` method of `p` only requires the `x,y`-coordinates of `q`. But if we replace the first occurrence of `p` by `q`, resulting in `q eq p`, then we have an error. Specifically, the `eq` method of `q` requires the `x,y`-coordinates and `color` of `p`, but `p` does not have a `color`. This produces the error *message not understood* in SMALLTALK-like systems. From the point of view of Chapters 14 and 15, this is a type error. Thus inheritance does not produce a subtype.

A difference between Chapters 14 and 15 is the way that objects are represented. The first uses recursively-defined records, in a calculus allowing record concatenation (or "combination," as it is called in the chapter), while the second uses, essentially, self-application in place of recursive definition. Some comparison between these two approaches appears in Section 7 of Chapter 14, and in the conference paper [Bru92], which was written after this collection was assembled. In both cases it is possible to use record operations to account for inheritance. A point that is not made in either comparison is that the recursively-defined records diverge under call-by-value evaluation order, while the alternative representation is satisfactory under both call-by-value and call-by-name.

Acknowledgements

We would like to thank Bob Prior, our editor at MIT Press, for suggesting the idea of collecting a selection of the work on this subject into a single volume. We are also indebted to the authors of the articles in the collection for their cooperation and their enthusiasm for the project.

Most of the articles in the volume are reprinted from previous publications where they have received varying levels of review. New submissions appearing in the volume were not subjected to a formal review process prior to their inclusion.

Bibliography

[Bru92] K. Bruce. The equivalence of two semantic definitions of inheritance in object-oriented languages. In *Proc. Mathematical Foundations of Programming Language Semantics*, pages 102–124, Berlin, 1992. Springer LNCS 598.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Special issue devoted to *Symp. on Semantics of Data Types,* Sophia-Antipolis (France), 1984.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[ES90] M. Ellis and B. Stroustrop. *The Annotated* C^{++} *Reference Manual.* Addison-Wesley, 1990.

[Gir71] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, Amsterdam, 1971.

[Gir72] J.-Y. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These D'Etat, Universite Paris VII, 1972.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation.* Addison Wesley, 1983.

[Hin69] J.R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Pau91] L.C. Paulson. *ML for the working programmer.* Cambridge Univ. Press, 1991.

[Ste87] L.A. Stein. Delegation is inheritance. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 138–146, 1987.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 37–44, 1987. Corrigendum in *Proc. IEEE Symp. on Logic in Computer Science,* page 132, 1988.

Part I

Objects and Subtypes

Chapter 1

User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. John C. Reynolds

First appeared in Stephen A. Schuman editor: New Directions in Algorithmic Languages, IFIP Working Group 2.1 on Algol, INRIA, pages 157-168. Reprinted in David Gries editor: Programming Methodology, A Collection of Papers by Members of IFIP WG 2.3, 1978, Springer-Verlag, pages 309-317.

Chapter 2

Using Category Theory to Design Implicit Conversions and Generic Operators. John C. Reynolds

First appeared in Neil D. Jones editor: Semantics-Directed Compiler Generation, Springer-Verlag Lecture Notes in Computer Science 94, 1980, pages 211-258.

Part II

Type Inference

Chapter 3

Type Inference for Records in a Natural Extension of ML.

Didier Rémy

Chapter 4

Type Inference for Objects with Instance Variables and Inheritance. Mitchell Wand

Chapter 5

Static Type Inference for Parametric Classes. Atsushi Ohori and Peter Buneman

In: Proceedings of ACM OOPSLA Conference, SIGPLAN Notices, Vol 24, Number 10, October 1989, pages 445-456.

Part III

Coherence

Chapter 6

A Modest Model of Records, Inheritance and Bounded Quantification. Kim Bruce and Guiseppe Longo

Information and Computation 87(1990), pages 196-240.

Chapter 7

Inheritance as Implicit Coercion. Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov

Information and Computation 93(1991), pages 172-221.

Chapter 8

Coherence of Subsumption, Minimum Typing and Type-Checking in $F_{\leq}$.

Pierre-Louis Curien and Giorgio Ghelli

Mathematical Structures in Computer Science, 2(1992), pages 55-91.

Part IV

Record Calculi

Chapter 9

Operations on Records. Luca Cardelli and John C. Mitchell

Mathematical Structures in Computer Science, 1(1991), pages 3-48.

Chapter 10

Typing Record Concatenation for Free. Didier Rémy

In: Nineteenth Annual Symposium on Principles Of Programming Languages, 1992, pages 166-176.
COPYRIGHT: The Association for Computing Machinery, Inc. 11 West 42nd Street New York, New York 10036.

Chapter 11

Extensible Records in a Pure Calculus of Subtyping. Luca Cardelli

Digital Equipment Corporation, Systems Research, Technical Report Number 81, 1992.

Chapter 12

Bounded Quantification is Undecidable. Benjamin C. Pierce

Part V

Inheritance

Chapter 13

Two Semantic Models of Object-Oriented Languages.

Sam Kamin and Uday Reddy

Chapter 14

Inheritance is Not Subtyping.

William R. Cook, Walter L. Hill and Peter S. Canning

In P. Hudak editor: Principles of Programming Languages, 1990, pages 125-135.

Chapter 15

Toward a Typed Foundation for Method Specialization and Inheritance. John C. Mitchell

In P. Hudak editor: Principles of Programming Languages, 1990, pages 109-124.