

custom-book-3

Written by consumer consumer

Table of Contents

book-1_1_5

book-1_15_20

Type Systems for Programming Languages

Benjamin C. Pierce
bcpierce@cis.upenn.edu

Working draft of January 15, 2000

This is preliminary draft of a book in progress. Comments, suggestions, and corrections are welcome.

Contents

Preface	8
1 Introduction	13
1.1 What is a Type System?	13
1.2 A Brief History of Type	14
1.3 Applications of Type Systems	16
1.4 Related Reading	16
2 Mathematical Preliminaries	18
2.1 Sets and Relations	18
2.2 Induction	18
2.3 Term Rewriting	19
3 Untyped Arithmetic Expressions	20
3.1 Basics	20
Syntax	21
Evaluation	21
3.2 Formalities	21
Syntax	21
Evaluation	25
3.3 Properties	27
3.4 Implementation	27
Syntax	27
Evaluation	28
3.5 Summary	28
3.6 Further Reading	30
4 The Untyped Lambda-Calculus	31
4.1 Basics	32
Syntax	33
Operational Semantics	34
4.2 Programming in the Lambda-Calculus	34

4.3	Is the Lambda-Calculus a Programming Language?	39
4.4	Formalities	39
	Syntax	39
	Substitution	40
	Operational Semantics	43
	Summary	43
4.5	Further Reading	44
5	Implementing the Lambda-Calculus	45
5.1	Nameless Representation of Terms	45
	Syntax	46
	Shifting and Substitution	48
	Evaluation	49
5.2	A Concrete Realization	49
	Syntax	50
	Shifting and Substitution	50
	Evaluation	51
5.3	Ordinary vs. Nameless Representations	51
6	Typed Arithmetic Expressions	52
6.1	Syntax	52
6.2	The Typing Relation	52
6.3	Properties of Typing and Reduction	53
	Typing Derivations	53
	Typechecking	54
	Safety = Preservation + Progress	54
6.4	Implementation	55
6.5	Summary	56
7	Simply Typed Lambda-Calculus	58
7.1	Syntax	58
7.2	The Typing Relation	59
7.3	Summary	61
7.4	Properties of Typing and Reduction	62
	Typechecking	62
	Typing and Substitution	64
	Type Soundness	64
7.5	Implementation	65
7.6	Further Reading	66

8	Extensions	67
8.1	Base Types	67
8.2	Unit type	67
8.3	Let bindings	68
8.4	Records and Tuples	68
8.5	Variants	72
8.6	General recursion	72
8.7	Lists	73
8.8	Lazy records and let-bindings	75
9	References	76
9.1	Further Reading	79
10	Exceptions	80
10.1	Errors	80
10.2	Exceptions	80
11	Type Equivalence	81
12	Definitions	83
12.1	Type Definitions	83
12.2	Term Definitions	85
13	Subtyping	86
13.1	The Subtype Relation	87
	Variance	88
	Summary	89
13.2	Metatheory of Subtyping	91
	Algorithmic Subtyping	91
	Minimal Typing	92
13.3	Implementation	96
13.4	Meets and Joins	97
13.5	Primitive Subtyping	99
13.6	The Bottom Type	99
13.7	Other stuff	99
14	Imperative Objects	100
14.1	Objects	100
14.2	Object Generators	102
14.3	Subtyping	103
14.4	Basic classes	104
14.5	Extending the Internal State	105
14.6	Classes with “Self”	106

15 Recursive Types	109
15.1 Examples	109
Lists	109
Hungry Functions	109
Recursive Values from Recursive Types	110
Untyped Lambda-Calculus, Redux	110
Recursive Objects	112
15.2 Equi-recursive Types	112
ML Implementation	114
15.3 Iso-recursive Types	115
15.4 Subtyping and Recursive Types	116
16 Case Study: Featherweight Java	117
17 Type Reconstruction	118
17.1 Substitution	118
17.2 Universal vs. Existential Type Variables	119
17.3 Constraint-Based Typing	121
17.4 Unification	125
17.5 Principal Typings	128
17.6 Further Reading	129
18 Universal Types	130
18.1 Motivation	130
18.2 Varieties of Polymorphism	131
18.3 Definitions	132
18.4 Examples	135
Warm-ups	135
Polymorphic Lists	136
Impredicative Encodings	136
18.5 Metatheory	139
Soundness	139
Strong Normalization	139
Erasure and Typeability	140
Type Reconstruction	141
18.6 Implementation	141
Nameless Representation of Types	141
ML Code	141
18.7 Further Reading	143

- Untyped — programs simply execute flat out; there is no attempt to check “consistency of shapes”
- Typed — some attempt is made, either at compile time or at run-time, to check shape-consistency

Among typed languages, we can break things down further:

	Statically checked	Dynamically checked
Strongly typed	ML, Haskell, Pascal (almost), Java (almost)	Lisp, Scheme
Weakly typed	C, C++	Perl

1.2 A Brief History of Type

The following table presents a (rough and incomplete) chronology of some important high points in the history of type systems in computer science. Related developments in logic are also included (in italics), to give a sense of the importance of this field’s contributions.

late 1800s	<i>Origins of formal logic</i>	[?]
early 1900s	<i>Formalization of mathematics</i>	[WR25]
1930s	<i>Untyped lambda-calculus</i>	[Chu41]
1940s	<i>Simply typed lambda-calculus</i>	[Chu40, CF58]
1950s	Fortran	[Bac81]
1950s	Algol	[N ⁺ 63]
1960s	<i>Automath project</i>	[dB80]
1960s	Simula	[BDMN79]
1970s	<i>Martin-Löf type theory</i>	[Mar73, Mar82, SNP90]
1960s	<i>Curry-Howard isomorphism</i>	[How80]
1970s	<i>System F, F^ω</i>	[Gir72]
1970s	polymorphic lambda-calculus	[Rey74]
1970s	CLU	[LAB ⁺ 81]
1970s	polymorphic type inference	[Mil78, DM82]
1970s	ML	[GMW79]
1970s	<i>intersection types</i>	[CDC78, CDCS79, Pot80]
1980s	NuPRL project	[Con86]
1980s	subtyping	[Rey80, Car84, Mit84a]
1980s	ADTs as existential types	[MP88]
1980s	<i>calculus of constructions</i>	[Coq85, CH88]
1980s	<i>linear logic</i>	[Gir87, GLT89]
1980s	bounded quantification	[CW85, CG92, CMMS94]

1980s	<i>Edinburgh Logical Framework</i>	[HHP92]
1980s	Forsythe	[Rey88]
1980s	<i>pure type systems</i>	[Bar92a]
1980s	dependent types and modularity	[Mac86]
1980s	Quest	[Car91]
1980s	<i>Extended Calculus of Constructions</i>	[Luo90]
1980s	Effect systems	[?, TJ92, TT97]
1980s	row variables and extensible records	[Wan87, Rém89, CM91]
1990s	higher-order subtyping	[Car90, CL91, PT94]
1990s	typed intermediate languages	[TMC ⁺ 96]
1990s	Object Calculus	[AC96]
1990s	translucent types and modarity	[HL94, Ler94]
1990s	typed assembly language	[MWCG98]

In computer science, the earliest type systems, beginning in the 1950s (e.g., FORTRAN), were used to improve efficiency of numerical calculations by distinguishing between natural-number-valued variables and arithmetic expressions and real-valued ones, allowing the compiler to use different representations and generate appropriate machine instructions for arithmetic operations. In the late 1950s and early 1960s (e.g., ALGOL), the classification was extended to structured data (arrays of records, etc.) and higher-order functions. Beginning in the 1970s, these early foundations have been extended in many directions...

- **parametric polymorphism** allows a single term to be used with many different types (e.g., the same sorting routine might be used to sort lists of natural numbers, lists of reals, lists of records, etc.), encouraging code reuse;
- **module systems** support programming in the large by providing a framework for defining (and automatically checking) interfaces between the parts of a large software system;
- **subtyping** and **object types** address the special needs of object-oriented programming styles;
- connections are being developed between the type systems of programming languages, the **specification languages** used in program verification, and the **formal logics** used in theorem proving.

All of these (among many others) are still areas of active research.

I'd like to include here a longer discussion of the historical origins of various ideas in type systems. This is usually how I use the whole first lecture of my graduate course, and it goes down very well, but to put it all in writing will require a bit of research.

1.3 Applications of Type Systems

Beyond their traditional benefits of robustness and efficiency, type systems play an increasingly central role in computer and network security: static typing lies at the core of the security models of Java and JINI, for example, and is the main enabling technology for Proof-Carrying Code. Type systems are used to organize compilers, verify protocols, structure information on the web, and even model natural languages.

Short sketches of some of these diverse applications...

- *In programming in the large (module systems, interface definition languages, etc.)*
- *In compiling and optimization (static analyses, typed intermediate languages, typed assembly languages, etc.)*
- *In “self-certification” of untrusted code (so-called “proof-carrying code” [NL96, Nec97, NL98])*
- *In security*
- *In theorem proving*
- *In databases*
- *In linguistics (categorical grammar [Ben95, vBM97, etc.] , and maybe something seminal by Lambek)*
- *In Y2K conversion tools*
- *DTDs and other “web metadata” (note from Henry Thompson: DTDs were originally designed for SGML because of the expense of cancelling huge typesetting runs due to errors in the markup!)*

1.4 Related Reading

While this book attempts to be self contained, it is far from comprehensive: the area is too large, and can be approached from too many angles, to do it justice in one book. Here are a few other good entry points:

- Handbook articles by Cardelli [Car96] and Mitchell [Mit90] offer quick introductions to the area. Barendregt’s article [Bar92b] is for the more mathematically inclined.
- Mitchell’s massive textbook on programming languages [Mit96] covers basic lambda calculus, a range of type systems, and many aspects of semantics.

- Abadi and Cardelli's *A Theory of Objects* [AC96] develops much of the same material as this present book, de-emphasizing implementation aspects and concentrating instead on the application of these ideas in a foundation treatment of object-oriented programming. Kim Bruce's forthcoming *Foundations of Object-Oriented Programming Languages* will cover similar ground. Introductory material on object-oriented type systems can also be found in [PS94, Cas97].
- Reynolds [Rey98] *Theories of Programming Languages*, a graduate-level survey of the theory of programming languages, includes beautiful expositions of polymorphic typing and intersection types.
- Girard's *Proofs and Types* [GLT89] treats logical aspects of type systems (the Curry-Howard isomorphism, etc.) thoroughly. It also includes a description of System F from its creator, and an appendix introducing linear logic.
- *The Structure of Typed Programming Languages*, by Schmidt [?], develops core concepts of type systems in the context of programming language design, including several chapters on conventional imperative languages. Simon Thompson's *Type Theory and Functional Programming* [Tho91] focuses on connections between functional programming (in the "pure functional programming" sense of Haskell or Miranda) and constructive type theory, viewed from a logical perspective.
- Semantic foundations for both untyped and typed languages are covered in depth in textbooks by Gunter [Gun92] and Winskel [Win93].
- Hindley's monograph *Basic Simple Type Theory* [Hin97] is a wonderful compendium of results about the simply typed lambda-calculus and closely related systems. Its coverage is deep rather than broad.

If you want a single book besides the one you're holding, I'd recommend either Mitchell or Abadi and Cardelli.

Chapter 2

Mathematical Preliminaries

This chapter mostly still needs to be written. I do not intend to go into a great deal of detail (a student that needs a real introduction to these topics is going to be lost in a couple of chapters anyway) — just remind the reader of basic concepts and notations.

Before getting started, we need to establish some common notation and state a few basic mathematical facts. Most readers should be able to skim this chapter and refer back to it as necessary.

2.1 Sets and Relations

2.2 Induction

2.2.1 Definition: A partially ordered set S is said to be **well founded** if it contains no infinite decreasing chains—that is, if there is no infinite sequence s_1, s_2, s_3, \dots of elements of S such that each s_{i+1} is strictly less than s_i . \square

2.2.2 Theorem [Principle of well-founded induction]: Suppose that the set S is well founded and that P is some predicate on the elements of S . If we can show, for each $s : S$, that $(\forall s' < s. P(s'))$ implies $P(s)$, then we may conclude that $P(s)$ holds for every $s : S$. \square

2.2.3 Corollary [Principle of induction on the natural numbers]: Suppose that P is some predicate on the natural numbers. If we can show, for each m , that $(\forall i < m. P(i))$ implies $P(m)$, then we may conclude that $P(n)$ holds for every n . \square

Proof: The set of natural numbers is well founded. \square

2.2.4 Corollary [Principle of lexicographic induction]: Define the following “dictionary ordering” on pairs of natural numbers: $(m, n) < (m', n')$ iff $m < m'$ or $m = m'$ and $n < n'$.

Now, suppose that P is some predicate on pairs of natural numbers. If we can show, for each (m, n) , that $(\forall (m', n') < (m, n). P(m', n'))$ implies $P(m, n)$, then we may conclude that $P(m, n)$ holds for every pair (m, n) .

(A similar principle holds for lexicographically ordered triples, quadruples, etc.) □

Proof: The lexicographic ordering on pairs of numbers is well founded. □

2.3 Term Rewriting

(or maybe this material should be folded into the next chapter...)