custom-book-3

Written by consumer consumer

Table of Contents

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

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

1980s	*Edinburgh Logical Framework*	[HHP92]
1980s	Forsythe	[Rey88]
1980s	*pure type systems*	[Bar92a]
1980s	dependent types and modularity	[Mac86]
1980s	Quest	[Car91]
1980s	*Extended Calculus of Constructions*	[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 (categorial 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 $s_1, s_2, s_3, \ldots$ of elements of S such that each s_{i+1} is strictly less than s_i. □

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 $s : S$. □

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. □

Proof: The set of natural numbers is well founded. □

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.) $\square$

Proof: The lexicographic ordering on pairs of numbers is well founded. $\square$

2.3 Term Rewriting

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