Abstract. This paper forms the substance of a course of lectures given at the International Summer School in Computer Programming at Copenhagen in August, 1967. [...] 

Strachey’s lectures give a broad overview of the state of the art in programming and programming language design as of 1967. Here we are only interested in his discussion of polymorphism (§3.6). He describes a polymorphic function simply as one that can operate at multiple different types. He distinguishes between ad-hoc and parametric polymorphism: an ad-hoc polymorphic function might behave in unrelated ways at different types, whereas a parametrically polymorphic function has, in some sense, a uniform behavior across all types.

Strachey’s significance in our theme is simply his introduction of the informal notion of parametric polymorphism, now referred to as Strachey parametricity. He gives no formal criteria to distinguish ad-hoc polymorphic functions from parametric functions, nor any way to say when a language guarantees that all definable polymorphic functions will be parametric.

Abstract. We explore the thesis that type structure is a syntactic discipline for maintaining levels of abstraction. Traditionally, this view has been formalized algebraically, but the algebraic approach fails to encompass higher-order functions. For this purpose, it is necessary to generalize homomorphic functions to relations; the result is an “abstraction” theorem that is applicable to the typed lambda calculus and various extensions, including user-defined types.

Finally, we consider polymorphic functions, and show that the abstraction theorem captures Strachey’s concept of parametric, as opposed to ad hoc, polymorphism.
In this paper Reynolds formalizes the idea of data abstraction: what does it mean for a client of a data type to be insensitive to the type’s choice of internal representation? An abstract data type exposes a type name and a set of operations on the type, and a client program respects the abstraction if it cannot make a concrete observation to distinguish different choices of representation for the abstract type. This is formalized, roughly, by requiring that when a program is run on two related implementations of the abstract type, the two results are (1) identical where they do not involve the abstract type, and (2) related where they do.

Reynolds’ starting point for modeling abstract data types is abstract algebra. Operations in traditional algebra are first order, so he bridges the gap to higher-order operations by generalizing homomorphisms from functions to relations. Homomorphic relations guide his definition of a relational semantics (a “logical relation” in modern terminology) for a simply typed $\lambda$-calculus extended with type variables. He then proves an abstraction theorem (the “basic lemma” for the logical relation) stating that the meaning of every term maps related environments to related results. In particular, the theorem establishes that no program definable in the language can violate data abstraction.

The remainder of the paper discusses various generalizations of the calculus and relational semantics that also satisfy the abstraction theorem. The main generalization introduces the polymorphic $\lambda$-calculus (independently of Girard) for defining functions that are parametrically polymorphic in the sense of Strachey. Reynolds extends the relational semantics to polymorphic types, reestablishes the abstraction theorem, and argues that the theorem captures Strachey’s informal notion of parametricity.

Reynolds’ relational semantics and polymorphic calculus have been the most lasting contributions to our understanding of parametricity. His abstraction theorem provides the formal notion of relational parametricity, the canonical interpretation of Strachey’s informal notion, and his polymorphic calculus is our prototype for languages with parametric polymorphism. Reynolds’ stated goal was to characterize data abstraction, but along the way he also discovered its intimate relationship with parametric polymorphism, so his paper served as a launching point for both lines of research.

Reynolds’ paper does not explore the consequences of his abstraction theorem—what have we gained by showing that our polymorphic functions behave uniformly for all types? He alludes to a few consequences in passing but stays focused on the fundamental issues.


*Abstract.* From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds’ abstraction theorem for the polymorphic lambda calculus.
Wadler shows how Reynolds’ relational parametricity can be used to reason about functional programs with polymorphism. After a direct and clear explanation of the relational semantics, he rephrases the abstraction theorem and dubs it the *parametricity theorem*. He then spends the majority of the paper illustrating how the theorem, when applied to various common polymorphic functions, gives useful equations for reasoning about the functions. Moreover, since the relational semantics is defined inductively over types, these equations can be given just from the type of a function, without knowing how the function is actually defined. Wadler concludes the paper with a straightforward redevelopment of Reynolds’ relational semantics in terms of the *frame models* of Bruce, Meyer, and Mitchell instead of the set-based models used in [Rey83]; Reynolds’ set-theoretic semantics contained the right ideas but was technically incoherent, and Reynolds himself showed a year later that no such set-theoretic semantics for polymorphism exists.

Wadler’s main contribution, as he essentially states, is to digest Reynolds’ work for the lay functional programmer and illustrate how it can be used to derive certain kinds of equations about polymorphic functions. Throughout he uses a limited form of the theorem to derive equations like:

If \( f : \forall \alpha. \text{list} \alpha \to \text{list} \alpha \) and \( g : t \to u \)

then \( \text{map} \ g \circ f_t = f_u \circ \text{map} \ g \)

These equations say, roughly, that a polymorphic function commutes with all functions that operate “under” its type structure—in this case, functions under the map of a list. Even though Wadler doesn’t use the full power of the abstraction theorem, his “free theorems” are useful in practice and justify a certain kind of reasoning that many functional programmers use when working with polymorphic functions.


Abstract. In this paper we introduce a logic for parametric polymorphism. Just as LCF is a logic for the simply-typed \( \lambda \)-calculus with recursion and arithmetic, our logic is a logic for System F. The logic permits the formal presentation and use of relational parametricity. Parametricity yields—for example—encodings of initial algebras, final co-algebras and abstract datatypes, with corresponding proof principles of induction, co-induction and simulation.

Plotkin and Abadi define a logic for formalizing arguments that rely on relational parametricity. The system is a standard second-order logic over System F terms, extended with an axiom schema expressing relational parametricity. By introducing the logic they hope to: (1) clarify the assumption imposed by relational parametricity and simplify its use, and (2) extend the reach of LCF-like program logics to languages with abstract or polymorphic types. They illustrate reasoning within the logic by formally comparing various notions of parametricity, and by deriving familiar logical properties of inductive types, coinductive types, and abstract types encoded in System F.

Plotkin and Abadi’s essential contribution is to show that the use of relational parametricity is available within a standard second-order program logic. For the purposes of our theme, their
paper provides clear and insightful examples, via the logic, of how relational parametricity can be used to reason about polymorphic programs—in this case, encodings of various datatypes and their extensionality properties. The logic also clearly expresses the assumption of relational parametricity: a family of axioms, defined inductively over types, requiring all terms to map “related environments to related results”.

\[ \forall x : (\forall \alpha. t) . \forall \beta, \gamma . \forall \rho \subseteq \beta \times \gamma . x_\beta t[\rho/\alpha] x_\gamma \]