Introduction to Part II

5

John C. Reynolds
Carnegie Mellon University

The polymorphic (or second-order) typed lambda calculus was invented by Jean-Yves Girard in 1971 [10, 11], and independently reinvented by myself in 1974 [22]. It is extraordinary that essentially the same programming language was formulated independently by the two of us, especially since we were led to the language by entirely different motivations.

In my own case, I was seeking to extend conventional typed programming languages to permit the definition of "polymorphic" procedures that could accept arguments of a variety of types. I started with the ordinary typed lambda calculus and added the ability to pass types as parameters (an idea that was "in the air" at the time, e.g. [4]).

For example, as in the ordinary typed lambda calculus one can write

$$\lambda f_{\mathbf{int}\to\mathbf{int}}.\ \lambda x_{\mathbf{int}}.\ f(f(x))$$

to denote the "doubling" function for the type **int**, which accepts a function from integers to integers and yields the composition of this function with itself. Similarly, using a type variable t, one can write

$$\lambda f_{t\to t}.\ \lambda x_t.\ f(f(x))$$

to denote the doubling function for t. Then, by abstracting on the type variable, one can define a polymorphic doubling function,

$$\Lambda t.\ \lambda f_{t\to t}.\ \lambda x_t.\ f(f(x)),$$

that can be applied to any type to obtain the doubling function for that type, e.g.,

$$(\Lambda t.\ \lambda f_{t\to t}.\ \lambda x_t.\ f(f(x)))[\mathbf{int}]$$

$$\Longrightarrow \lambda f_{\mathbf{int}\to\mathbf{int}}.\ \lambda x_{\mathbf{int}}.\ f(f(x))$$

or

$$(\Lambda t.\ \lambda f_{t\to t}.\ \lambda x_t.\ f(f(x)))[\mathbf{real}\to\mathbf{real}]$$

$$\Longrightarrow \lambda f_{(\mathbf{real}\to\mathbf{real})\to(\mathbf{real}\to\mathbf{real})}.\ \lambda x_{\mathbf{real}\to\mathbf{real}}.\ f(f(x)).$$

Notice that an upper case Λ and square brackets are used to indicate abstraction and application of types, and that $\Longrightarrow$ denotes a kind of beta reduction for types, in which type expressions are substituted for occurrences of type variables within ordinary expressions.

To accommodate this kind of abstraction and application of types, it is necessary to expand the variety of type expressions to provide types for the polymorphic functions. Somewhat surprisingly, this can be done in such a way that (if the type of every variable binding is given explicitly) type correctness can be determined syntactically (i.e., at compile time). One writes $\Delta t.\ \omega$ (where Δ is a binding operator) to denote the type of polymorphic function that, when applied to a type t, yields a result of type ω. For example, the polymorphic doubling function has type

$$\Delta t.\ (t\to t)\to(t\to t),$$

and the polymorphic identity function,

$$\Lambda t.\ \lambda x_t.\ x,$$

has type

$$\Delta t.\ t\to t.$$

If an expression e has type ω then $\Lambda t.\ e$ has type $\Delta t.\ \omega$, and if an expression e has type $\Delta t.\ \omega$ then $e[\omega']$ has the type obtained from ω by substituting ω' for t. Thus it is straightforward to decide the type of any expression.

The movitation that led Girard to essentially the same language was entirely different; he was seeking to extend an analogy between types and propositions that was originally found by Curry [8, Section 9E] and Howard [12]. Types can be viewed as propositions by regarding the type constructor $\rightarrow$ as the logical connective **implies**. (Similarly, one can regard the Cartesian product constructor $\times$ as the connective **and** and the disjoint union constructor $+$ as the connective **or**.) Then an expression e of type ω becomes an encoding of a proof of the proposition ω in intuitionistic logic.

For example, the doubling function for t encodes the following, rather roundabout proof that (t **implies** t) **implies** (t **implies** t), in which t is some arbitrary proposition and e: indicates that the proof step is encoded by the expression e.

> Assume f: t **implies** t.
>
> Assume x: t.
>
> Since f: t **implies** t and x: t, we have $f(x)$: t.
>
> Since f: t **implies** t and $f(x)$: t, we have $f(f(x))$: t.
>
> Discharging the assumption x, we have $\lambda x.\ f(f(x))$: t **implies** t.
>
> Discharging the assumption f, we have
> $\lambda f.\ \lambda x.\ f(f(x))$: ($t$ **implies** t) **implies** (t **implies** t).

Girard extended the Curry-Howard analogy by regarding the binding operator $\Delta t.$ as a universal quantifier of a propositional variable, i.e., as "For all propositions t". (He also introduced an analogous existential quantifier.) Thus the polymorphic doubling function encodes a proof that

($\forall t$) (t **implies** t) **implies** (t **implies** t).

Notice that there is a circularity or "impredicativity" here, since such a quantified proposition belongs to the set of propositions being quantified over. (This circularity is also present in the Coquand-Huet Calculus of Constructions, which includes the polymorphic calculus as a sublanguage, but not in the types-as-propositions formalisms of Martin-Löf [17] or Constable [5].)

Despite this circularity, Girard showed that every expression of the polymorphic typed lambda calculus possesses a normal form, i.e., that every expression can be reduced by some finite sequence of beta reductions to a form that cannot be reduced further. (This result was strengthened by Prawitz [21, p. 256] to show that every expression is strongly normalizable, i.e., that no expression is amenable to any infinite sequence of beta reductions.) Proof-theoretically, this means that every proof can be transformed into a "cut-free" proof. Computationally, it means that every expression describes a terminating computation.

This is extraordinary. For any language in which every expression describes a terminating computation, there must be computable functions that cannot be expressed; indeed we are used to taking this fact as evidence that such languages are uninteresting for practical computation. Yet the polymorphic typed lambda calculus is just such a language, in which one can express "almost everything" that one might actually want to compute. Indeed, Girard has shown that every function from natural numbers to natural numbers that can be proved total by using second-order arithmetic can be expressed in the calculus. This includes not only primitive recursive functions, but also Ackermann's function as well as far more esoteric (and rapidly growing) functions.

This result depends upon a particular way of encoding the natural numbers called "Church numerals". In his early work on the untyped lambda calculus, Church used the encoding

$$
\begin{aligned}
0&: \quad \lambda f.\, \lambda x.\, x\,, \\
1&: \quad \lambda f.\, \lambda x.\, f(x)\,, \\
2&: \quad \lambda f.\, \lambda x.\, f(f(x))\,, \\
&\dots
\end{aligned}
$$

The obvious analogue for the polymorphic calculus is

$$
\begin{aligned}
0&: \quad \Lambda t.\, \lambda f_{t\to t}.\, \lambda x_t.\, x\,, \\
1&: \quad \Lambda t.\, \lambda f_{t\to t}.\, \lambda x_t.\, f(x)\,, \\
2&: \quad \Lambda t.\, \lambda f_{t\to t}.\, \lambda x_t.\, f(f(x))\,, \\
&\dots
\end{aligned}
$$

In both cases, n is encoded by a higher-order function mapping f into f^n, reflecting the idea that the fundamental use of a natural number n is to iterate something n times. (For example, 2 is encoded by the doubling function.) But in the polymorphic typed case, there is a particular type

$$\mathbf{nat} \stackrel{\text{def}}{=} \Delta t.\, (t \to t) \to (t \to t)$$

that is possessed by every encoding of a natural number. Moreover, every closed (and constant-free) expression of this type is equivalent (via beta and eta reduction) to such an encoding. Thus it is reasonable to regard **nat** as the type of natural numbers.

Using this encoding, one can program arithmetic functions such as

$$succ \stackrel{\text{def}}{=} \lambda n_{\mathbf{nat}}.\, \Lambda t.\, \lambda f_{t\to t}.\, \lambda x_t.\, f(n[t](f)(x)),$$

$$add \stackrel{\text{def}}{=} \lambda m_{\mathbf{nat}}.\, \lambda n_{\mathbf{nat}}.\, \Lambda t.\, \lambda f_{t\to t}.\, \lambda x_t.\, m[t](f)(n[t](f)(x)).$$

Notice that these are functions that accept and produce polymorphic functions. (Such functions go beyond the kind of implicit polymorphism provided by ML.)

Other fundamental sets can be encoded in a similar spirit. For example, the type

$$\mathbf{bool} \stackrel{\text{def}}{=} \Delta t.\, t \to (t \to t)$$

is possessed by the two "choice" functions

$$\Lambda t.\, \lambda x_t.\, \lambda y_t.\, x \quad \text{and} \quad \Lambda t.\, \lambda x_t.\, \lambda y_t.\, y,$$

and every closed expression of this type beta-reduces to one of these functions. Thus it is reasonable to regard **bool** as the type of Boolean values, reflecting the idea that the fundamental use of a Boolean is to make binary choices.

Less trivially, the closed expressions of type

$$\mathbf{list}(s) \stackrel{\text{def}}{=} \Delta t.\, (s \to (t \to t)) \to (t \to t)$$

have normal forms of the form

$$\Lambda t.\, \lambda f_{s\to(t\to t)}.\, \lambda x_t.\, f(e_1)(\ldots(f(e_n)(x))\ldots),$$

where $e_1, \ldots, e_n$ are subexpressions of type s. Thus **list**(s) can be regarded as the type of lists with elements of type s, reflecting the idea that the fundamental use of a list is to reduce the list (in the sense of APL).

These encodings are all special cases of a general result, discovered independently by Böhm [2] and Leivant [15], and anticipated in the work of Takeuti [28, Proposition 3.15.18]. For any many-sorted algebraic signature without laws, there is a set of polymorphic types (one for each sort) whose closed normal forms constitute an initial algebra. Moreover, the operations of this algebra can be expressed as functions among these types. Thus the polymorphic calculus encompasses algebraic data types as well as number-theoretic computations. Several examples of the kind of programming that is entailed are given in [25].

In summary, the polymorphic typed lambda calculus is far more than an extension of the simply typed lambda calculus that permits polymorphism. It is a language that guarantees the termination of all programs, while providing a surprising degree of expressiveness for computations over a rich variety of data types. In "Computable Values Can Be Classical" (in this volume), Val Breazu-Tannen and Albert Meyer argue that the guarantee of termination substantially simplifies reasoning about programs by permitting the conservation of classical data type specifications. In "Polymorphism Is Conservative over Simple Types" (also in this volume), the same authors further substantiate this argument by showing that polymorphism can be superimposed on familiar programming language features without changing their behavior.

However, the practicality of this language is far from proven. To say that any reasonable function can be expressed by some program is not to say that it can be expressed by the most reasonable program. It is clear that the language requires a novel programming style. Moreover, it is likely that certain important functions cannot be expressed by their most efficient algorithms. Also, the guarantee of termination precludes interesting computations that never terminate, such as those involving lazy computation with infinite data structures. (These reservations apply to the pure polymorphic calculus; if a fixed-point operator is added to provide general recursion, the language expands to include conventional functional programming, including lazy computation, but the guarantee of termination is lost.)

The known semantic models of the polymorphic typed lambda calculus can be divided into two species. In the first, the meaning of a type is (the set of equivalence classes of) a partial equivalence relation on a model of the untyped lambda calculus. This view characterizes the earliest models [11, 29], as well as recent work [16, 20, 9, 14, 3, and in this volume, John Mitchell's "A Type-Inference Approach to Reduction Properties and Semantics of Polymorphic Expressions"] that embeds such models in the natural setting of the "effective topos" [13]. (The connection between this kind of model and the effective topos, or equivalently, the "realizability universe", seems to have been first noted by Moggi.)

In the second kind of model, the meaning of a type is a Scott domain. In the earliest of these models [19], these domains were sets of fixed points of closures of a universal domain, where a closure of a domain is an idempotent continuous function from the domain to itself that extends the identity function. Two facts permit this concept to serve as a model of the polymorphic calculus:

> There is a universal domain U such that $U \to U$, the domain of continuous functions from U to U, is isomorphic to the set of fixed points of a

closure of U.

The set of closures of U, which can be regarded as meanings of types, is isomorphic to the set of fixed points of a closure of $U \to U$.

Similar models have been developed in which the concept of closure is replaced by that of finitary retraction [18] or of finitary projection [1]. More recently, Girard has devised a model based on the use of qualitative domains and stable functions, which is described in his paper "The System F of Variable Types, Fifteen Years Later" (in this volume). Other models of this kind are described in [7, 6].

The domain-based models describe not only the pure calculus but also the extension obtained by adding fixed-point operators. Thus they fail to capture the fact that all expressions denote terminating programs and represent proofs of their type interpreted as a proposition. A vivid consequence of this failure is that the type $\Delta t.\ t$, which is clearly false when interpreted as a proposition (and which is not the type of any expression in the pure language), denotes a nonempty domain. Whether such types have empty denotations is a pivotal question about semantic models, whose implications are described in "Empty Types in Polymorphic λ-Calculus" (in this volume), by Meyer, Mitchell, Moggi, and Statman.

Another shortcoming of the domain-based models is their failure to capture the notion of "parametricity". When Christopher Strachey first coined the word "polymorphism" [27], he distinguished between ad hoc polymorphic functions, which can have arbitrarily different meanings for different types, and parametric polymorphic functions, which must behave similarly for different types. Intuitively, only parametric polymorphic functions can be defined in the polymorphic calculus, but the domains denoted by polymorphic types in the domain-based models also contain ad hoc functions.

It is not known whether any of the partial-equivalence-relation models enforce parametricity (except, in a trival sense, the collapsed term model of [3]). Indeed, at present there is no general agreement on how to define parametricity precisely and generally, although a first attempt in this direction was given in [23], and a more recent approach appears in this volume in "Functional Polymorphism" by Bainbridge, Freyd, Scedrov, and Scott.

The fact that all expressions are strongly normalizable, and that certain types correspond to initial algebras, make it plausible that there should be a model extending the naive set-theoretic model of the simply typed lambda calculus, in which types denote sets and $S \to S'$ denotes the set of all functions from S to S'. Indeed, I made such a conjecture in 1983 [23]. Then in the following year —to my embarassment— I proved the conjecture false [24]. (This proof uses a cardinality argument that can be made in classical, but not constructive, logic. Indeed, as shown in [20] and [16], set-theoretic models can be found in a constructive metatheory.) Soon thereafter, Gordon Plotkin generalized my proof, showing that it is based upon a general property of functors (on the Cartesian closed category underlying an arbitrary model) that can be expressed in the calculus. This generalization is described in this volume in "On Functors Expressible in the Polymorphic Typed Lambda Calculus."

Beneath all these specific models lies the question of what, in general, constitutes a model of the language, which is discussed by Kim Bruce, Albert Meyer, and John Mitchell in "The Semantics of Second-Order Lambda Calculus" (in this volume). A more abstract answer to this question, using category-theoretic concepts, has been given by Seely [26].

The polymorphic lambda calculus also raises the problem of type inference. Although type checking is straightforward for the explicitly typed form of the calculus, the explicit statement of types whenever a variable is bound is a serious burden for the programmer. Ideally, one would like an algorithm that could examine an expression of the untyped lambda calculus and decide whether there is any assignment of types to variables that makes the expression well-typed. However, despite considerable efforts, the existence of such an algorithm for the polymorphic calculus remains an open question.

Current research on this question is described in this volume in "Polymorphic Type Inference and Containment" by John Mitchell. In "A Type-Inference Approach to Reduction Properties and Semantics of Polymorphic Expressions", also in this volume, the same author applies type inference to the study of the calculus itself, obtaining a simplified proof of the strong normalization property and a proof of completeness for a class of partial-equivalence-relation models.

The author wishes to thank Val Breazu-Tannen, Kim Bruce, Carl Gunter, Giuseppe Longo, Albert Meyer, John Mitchell, Eugenio Moggi, Gordon Plotkin, Andre Scedrov, and Rick Statman, all of whom have contributed comments that have improved the accuracy and generality of this introduction.

References

[1] Amadio, R., Bruce, K. B., and Longo, G. "The finitary projection model for second order lambda calculus and solutions to higher order domain equations". In *Proceedings Symposium on Logic in Computer Science*, pp. 122–130, 1986.

[2] Böhm, C. and Berarducci, A. "Automatic synthesis of typed Λ-programs on term algebras". *Theoretical Computer Science 39* (1985), pp. 135–154.

[3] Breazu-Tannen, V. and Coquand, T. "Extensional models for polymorphism". *Theoretical Computer Science 59* 1–2 (July 1988), pp. 85–114.

[4] Cheatham, T. E. Jr., Fischer, A., and Jorrand, P. "On the basis for ELF — an extensible language facility". *Proceedings AFIPS 1968 Fall Joint Computer Conference*, vol. 33, part 2, pp. 937–948. Thompson Book Company, Washington, D. C., 1968.

[5] Constable, R. L. et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, N.J., 1986.

[6] Coquand, T., Gunter, C. A., and Winskel. G. "Domain theoretic models of polymorphism". *Information and Computation 81*, 2 (May 1989), pp. 123–167.

[7] Coquand, T., Gunter, C. A, and Winskel, G. "DI-domains as a model of polymorphism". *Mathematical Foundations of Programming Language Semantics* (Proceedings, 1987), M. Main, A. Melton, M. Mislove, and D. Schmidt, eds., pp. 344–363. Lecture Notes in Computer Science, vol. 298. Springer-Verlag, Berlin, 1987.

[8] Curry, H. B. and Feys, R. *Combinatory Logic, Volume I* (second edition). North-Holland, Amsterdam, 1968.

[9] Freyd, P. and Scedrov, A. "Some semantic aspects of polymorphic lambda calculus". *Proceedings Symposium on Logic in Computer Science*, pp. 315–319, 1987.

[10] Girard, J.-Y. "Une extension de l'interpretation de Gödel à l'analyse, et son application a l'elimination des coupures dans l'analyse et la théorie des types". *Proceedings of the Second Scandinavian Logic Symposium*, J. E. Fenstad, ed., pp. 63–92. North-Holland, Amsterdam, 1971.

[11] Girard, J.-Y. *Interprétation Fonctionnelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, 1972.

[12] Howard, W. A. "The formulae-as-types notion of construction". *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin and J. R. Hindley, eds., pp. 479–490. Academic Press, London, 1980.

[13] Hyland, J. M. E. "The effective topos". *The L. E. J. Brouwer Centenary Symposium*, A. S. Troelstra and D. Van Dalen, eds., pp. 165–216. North-Holland, Amsterdam, 1982.

[14] Hyland, J. M. E. "A small complete category". *Annals of Pure and Applied Logic 40*, 2 (November 1988), pp. 135–165.

[15] Leivant, D. "Reasoning about functional programs and complexity classes associated with type disciplines". *24th Annual Symposium on Foundations of Computer Science*, pp. 460–469, 1983.

[16] Longo, G. and Moggi, E. "Constructive natural deduction and its 'modest' interpretation". To appear in *Semantics of Natural and Computer Languages*, J. Meseguer, ed. MIT Press, Cambridge, Mass.

[17] Martin-Löf, P. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

[18] McCracken, N. J. "A finitary retract model for the polymorphic lambda-calculus". To appear in *Information and Control*.

[19] McCracken, N. J. *An Investigation of a Programming Language with a Polymorphic*

Type Structure. Ph. D. dissertation, Syracuse University, June 1979.

[20] Pitts, A. M. "Polymorphism is set theoretic, constructively". *Category Theory and Computer Science*, D. H. Pitt, A. Poigné, and D. E. Rydeheard, eds., pp. 12–39. Lecture Notes in Computer Science, vol. 283. Springer-Verlag, Berlin, 1987.

[21] Prawitz, D. "Ideas and results in proof theory". *Proceedings of the Second Scandinavian Logic Symposium*, J. E. Fenstad, ed., pp. 235–307. North-Holland, Amsterdam, 1971.

[22] Reynolds, J. C. "Towards a theory of type structure". *Proceedings, Colloque sur la Programmation*, pp. 408–425. Lecture Notes in Computer Science, vol. 19. Springer-Verlag, Berlin, 1974.

[23] Reynolds, J. C. "Types, abstraction and parametric polymorphism". *Information Processing 83*, R. E. A. Mason, ed., pp. 513–523, Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1983.

[24] Reynolds, J. C. "Polymorphism is not set-theoretic". *Semantics of Data Types*, G. Kahn, D. B. MacQueen, and G. D. Plotkin, eds., pp. 145–156. Lecture Notes in Computer Science, vol. 173. Springer-Verlag, Berlin, 1984.

[25] Reynolds, J. C. "Three approaches to type structure". *Mathematical Foundations of Software Development*, H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, eds., pp. 97–138. Lecture Notes in Computer Science, vol. 185. Springer-Verlag, Berlin, 1985.

[26] Seely, R. A. G. "Categorical semantics for higher order polymorphic lambda calculus". *Journal of Symbolic Logic 52*, 4 (December 1987), pp. 969–989.

[27] Strachey, C. "Fundamental concepts in programming languages". August 1967.

[28] Takeuti, G. *Proof Theory*. North-Holland, Amsterdam, 1975.

[29] Troelstra, A. S. (editor). *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics*, vol. 344. Springer-Verlag, Berlin, 1973.