Empty Types in Polymorphic Lambda Calculus (Preliminary Report)

Albert R. Meyer*
MIT Lab. for Computer Sci.

John C. Mitchell
AT&T Bell Labs

Eugenio Moggi
Univ. Edinburgh

Richard Statman
Mathematics Dept.
Carnegie-Mellon Univ.

Received 11/4/86

Abstract. The model theory of simply typed and polymorphic (second-order) lambda calculus changes when types are allowed to be empty. For example, the "polymorphic Boolean" type really has *exactly* two elements in a polymorphic model only if the "absurd" type $\forall t.t$ is empty. The standard β-η axioms and equational inference rules which are complete when all types are nonempty are *not complete* for models with empty types. Without a little care about variable elimination, the standard rules are not even *sound* for empty types. We extend the standard system to obtain a complete proof system for models with empty types. The completeness proof is complicated by the fact that equational "term models" are not so easily obtained: in contrast to the nonempty case, not every theory with empty types is the theory of a single model.

1 Why empty types?

Functional languages with polymorphic control constructs and polymorphic data types support an attractive programming style which has been suggested by several authors [5], [6], [8], [9], [13], [14] [16], [17], [19], [20].

For example, Booleans and conditional operators arise directly from polymorphic concepts. Namely, the type

$$\mathit{polybool} ::= \forall t.\, t \to t \to t$$

is often called the type of *polymorphic Booleans.* One closed term of type *polybool* is

*This research was supported by NSF Grant No. 8511190-DCR and by ONR grant No. N00014-83-K-0125.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specfic permission.

© 1987 0-89791-215-2/87/0100-0253 75¢

$$True ::= \lambda t.\, \lambda x{:}t.\, \lambda y{:}t.\, x\,.$$

That is, *True* is the polymorphic projection-on-the-first-argument function, which given any type, t, returns the projection-on-the-first-argument function of type $t \to t \to t$.

Notation. Read "$x{:}t$" as "x is of type t". We reserve t as a variable ranging over types, writing $\lambda t.\, M$ instead of $\lambda t{:}\, Type.\, M$. Full definitions appear in Section 3 below. ■

Another term of type *polybool* is

$$False ::= \lambda t.\, \lambda x{:}t.\, \lambda y{:}t.\, y\,,$$

viz., the polymorphic projection-on-the-second-argument function. Indeed, *True* and *False* are the *only* pure, *i.e.*, constant-free, closed terms of type *polybool*. Defining the polymorphic conditional to be simply application:

$$Cond ::= \lambda t.\, \lambda b{:}\, polybool.\, b\,t,$$

we easily verify that for all $x, y{:}t$

$$Cond\,t\;True\,x\,y = x \tag{1}$$

and likewise

$$Cond\,t\;False\,x\,y = y \tag{2}$$

Thus it seems that Booleans and conditionals need not be added as a separate feature since they already appear as an intrinsic part of a polymorphic language. However, this appearance is misleading. For example, the equation

$$Cond\;t\,b\,y\,y = y \tag{3}$$

does not follow from the definitions above, even though it *does* follow directly from equations (1) and (2) when $b{=}True$ and when $b{=}False$. The problem is that even though *True* and *False* are the only two values of type *polybool* which are *definable* by pure closed terms, there are models with additional polymorphic Boolean elements for which equation (3) fails, *e.g.*, when $b = \bot_{polybool}$ in the usual cpo-based models [22], [4].

Thus we arrive at the kind of question which led us to the present study: is it *consistent* to assume equation (3) as a further axiom of polymorphic calculus? More generally, is it consistent to assume that *True* and *False* are the *only* elements of type *polybool*?

Ingenious model constructions by Moggi [18] and Coquand [7] (see [3], [2]), which we shall not try to develop here, show that the answer is yes.

Proposition 1.1 (Moggi, Coquand) *There is a model of the polymorphic lambda calculus containing exactly two elements of type* polybool. *Equation (3) is necessarily valid in such a model.*

The models satisfying Proposition (1.1) contain types with no elements. It turns out that this is inevitable. To see this, consider the "absurd" type $\forall t.t$. Any element, f, of this type chooses, for any type σ, an element $(f\sigma)$ of type σ. Thus, if *any* type is empty, then the absurd type must be empty as well. The simple argument below shows that if there is such a choice function, f, which there trivially will be in any model without empty types, then equation (3) is inconsistent.

Note that under an interpretation in which every type has at most one element, every well-formed equation is valid. By convention, such trivial interpretations are ruled out in the context of equational reasoning, so a *model* of type theory is required to have at least one type with more than one element. Now it is easy to see that in the polymorphic calculus, *True=False* iff every type has at most one element. Indeed, from the equation *True=False*, one can derive any well-formed equation using standard inference rules. Hence, we say a set of equations is *inconsistent* iff the equation *True=False* follows from them. This is then equivalent to saying the set of equations has no model.

Proposition 1.2 *In any model of polymorphic lambda calculus with all types nonempty, equation (3) is not valid. In particular, there must be more than two elements of type* polybool *in such a model.*

Proof. Suppose f: $\forall t.t$ and (3) holds. Let $b ::= (\lambda t.\lambda x{:}t.\lambda y{:}t.(ft))$. Note that b: *polybool*. Now by definition of b, we have b *polybool* $y\,y = (f$ *polybool*$)$, so by definition of *Cond* and equation (3), we have $y = (f$ *polybool*$)$, *i.e.*, all elements y of type *polybool* are equal (to $(f$ *polybool*$)$). In particular, *True=False*. ∎

So empty types are necessary if *polybool* is to model Booleans exactly. Of course, *polybool* is simply one example of a type where one would like and expect the only elements to be the definable ones. For example, a variation of the proof of Proposition (1.2) applies to the type *polyint* $::= \forall t.(t \to t) \to t \to t$ of Church's "polymorphic integers", so that without empty types, one must accept additional "nonstandard" polymorphic integers besides those definable by the familiar Church numerals.

It is an interesting open problem to explain how these propositions generalize to other types.

Conjecture 1.3 *Propositions (1.1) and (1.2) generalize to arbitrary "universal" types u of the form $\forall t_1 \ldots \forall t_n.\sigma$ where σ is an expression built from type variables $t_1, \ldots, t_n$ using $\to$. Namely, there can be a model in which the elements of u are precisely those definable by the pure closed terms of type u (and moreover, no two such terms have the same value) iff models may have empty types.*

Propositions (1.1) and (1.2) make it clear that

- empty types force themselves into consideration in the context of polymorphism, and
- having them significantly changes the theory of various familiar types such as polymorphic Booleans and integers.

2 Problems with empty types

Empty types complicate the relationship between models and lambda-calculus theories. The set of equations valid in a given collection of models is called a (semantic) *theory*. If we assume no type is empty, then every theory is actually the theory of a *single* "generic" model. This fact is significant in proving completeness (with respect to models with nonempty types) of the familiar equational reasoning (*cf.* [10], [15], [4]). It fails to hold when we allow empty types:

Proposition 2.1 *Let b_0 and b_1 be constants denoting types. The theory of the (collection of all models satisfying the) equation*

$$\lambda x{:}\, b_0.\, \lambda y{:}\, b_1.\, \mathit{True} = \lambda x{:}\, b_0.\, \lambda y{:}\, b_1.\, \mathit{False} \tag{4}$$

is not equal to the theory of any single model of the polymorphic lambda calculus.

Proof. If types b_0 and b_1 are both nonempty, then by applying the functions on either side of the equation to arguments of these respective types, we can derive the inconsistency $\mathit{True} = \mathit{False}$. Therefore, in every model in the collection, either b_0 must be empty, or b_1 must be empty. But it is not hard to find models in which only one of these types is empty. So the equation

$$\lambda x{:}\, b_0.\, \mathit{True} = \lambda x{:}\, b_0.\, \mathit{False} \tag{5}$$

which says that b_0 is empty, is not in the theory of the collection, nor is the corresponding equation about b_1

$$\lambda x{:}\, b_1.\, \mathit{True} = \lambda x{:}\, b_1.\, \mathit{False}\,. \tag{6}$$

But in any *single* model in the collection at least one of b_0, b_1 must be empty, so at least one of equations (5), (6) must be in the theory of that model. ∎

In fact, with suitable added axioms for a base-type *Bool*, the above argument applies to the simply typed lambda calculus as well.

A related model-theoretic contrast between the situations with and without empty types is that without empty types there is a *minimum* model which is a "final" object in the space of models, *i.e.*, it is a "homomorphic image of a submodel" of every model. Consequently, its theory is maximum, namely, contains all equations between pure closed lambda terms which are individually consistent [23], [1, A.1.23].

Conjecture 2.2 *There is no maximum pure theory-with-empty-types of simply typed nor of polymorphic lambda calculus.*

3 Typed terms and equations

We now define precisely the two calculi in this paper: simply typed [1, Appendix A] and polymorphic [11], [21] lambda calculus. The types of simply typed lambda calculus are given by

$$\tau ::= \mathbf{a} \mid \tau \to \tau$$

where $\mathbf{a}$ is a constant denoting a type. (There may be more than one type constant.)

Typed lambda terms are usually defined by assuming that there are infinitely many variables $x_1^\tau, x_2^\tau, \ldots$ for each type τ. However, when we assume there is a variable x^τ of type τ, we are in fact assuming that τ is nonempty. This leads us to present the syntax of terms in another form.

The terms, and their types, are defined using the subsidiary notion of type assignment. A *type assignment* A is a finite set of formulas $x{:}\tau$, with no x occurring twice in A. We write $A[x{:}\sigma]$ for the type assignment

$$A[x{:}\sigma] = \{y{:}\tau \in A \mid y \text{ different from } x\} \cup \{x{:}\sigma\}.$$

Terms will be written in the form $A \mapsto M{:}\tau$, which may be read, "under type assignment A, the term M has type τ." The well-typed terms are defined as follows.

$$A \mapsto x{:}\tau \text{ for } x{:}\tau \in A$$

$$\frac{A \mapsto M{:}\sigma \to \tau,\ A \mapsto N{:}\sigma}{A \mapsto MN{:}\tau}$$

$$\frac{A[x{:}\sigma] \mapsto M{:}\tau}{A \mapsto \lambda x{:}\sigma.M{:}\sigma \to \tau}$$

The types of polymorphic lambda calculus are defined by adding two more clauses to the rules for simple types:

$$\tau ::= \mathbf{a} \mid \tau \to \tau \mid t \mid \forall t.\tau$$

Additional term formation rules for these new types are:

$$\frac{A \mapsto M{:}\forall t.\tau}{A \mapsto M\sigma{:}[\sigma/t]\tau}$$

where $[\sigma/t]\tau$ is the result of substituting σ for t in τ, and

$$\frac{A \mapsto M{:}\tau}{A \mapsto \lambda t.M{:}\forall t.\tau}$$

where we assume t is not free in any type occurring in A.

Given this formulation of terms, it is natural to write equations in the form

$$A \mapsto M = N{:}\tau \tag{7}$$

where it is required that $A \mapsto M{:}\tau$ and $A \mapsto N{:}\tau$.

To facilitate reasoning about empty types, it is convenient to add assumptions of the form $\mathit{empty}(\sigma)$ to type assignments. Therefore, an *equation* will be a formula of the form (7) where A is now to be the union of a type assignment A_1 and a set A_2 of formulas $\mathit{empty}(\sigma)$, and also $A_1 \mapsto M{:}\tau$ and $A_1 \mapsto N{:}\tau$.

Note that the emptiness assertions in A are not used to determine the types of terms.

4 Unsoundness of variable elimination

There are no types which are empty in all models. Consequently, no type is provably empty in the pure lambda theory. More generally,

Theorem 4.1 *An equation is valid in all models without empty types iff it is valid in all models with empty types.*

This will follow from Corollary (A.3) below. Thus, it might seem that admitting models with empty types should not have much affect on equational logic of terms.

However, when we reason from equational *hypotheses*, the valid consequences are very different depending on whether empty types are allowed.

For example, we have remarked that the equation (5), taken with the empty type assignment, is true in a model iff the type b_0 is empty. In particular, from (5), we can certainly conclude

$$x{:}\,b_0 \mapsto (\lambda x{:}\,b_0.\ \mathit{True})x = (\lambda x{:}\,b_0.\ \mathit{False})x{:}\,(b_0 \to \mathit{polybool}),$$

so by β-reduction, we have

$$x{:}\,b_0 \mapsto \mathit{True} = \mathit{False}{:}\,\mathit{polybool}. \tag{8}$$

Now if every type is nonempty, then the rule

$$(\mathit{nonempty}) \qquad \frac{A \cup \{x{:}\,\sigma\} \mapsto M = N{:}\,\tau}{A \mapsto M = N{:}\,\tau}\ x \text{ not free in } M, N$$

is sound, since assuming something about the type of an irrelevant variable x has no effect on the validity of an equation. (This rule is usually not stated in proof systems for typed lambda calculus, since it is implicit in the usual formulation for terms over models without empty types.) Then from (8) and (*nonempty*), we deduce the inconsistency

$$\emptyset \mapsto \mathit{True} = \mathit{False}{:}\,\mathit{polybool}. \tag{9}$$

This is the formal confirmation of the obvious fact that the assertion that type b_0 is empty is inconsistent with the assumption that all types are nonempty. However, equation (5) is not inconsistent if we allow models with b_0 interpreted as empty.

Thus, an equation between terms which follows under some type assignment, does *not* necessarily follow under an assignment involving *only* the free variables of the terms. The problem is that a type assignment $x{:}\,\sigma$ implies σ is nonempty, and such an assumption cannot be discharged without justification when empty types are possible. In short, rule *nonempty* is *not sound* in when empty types are allowed. (A similar kind of unsoundness was already observed for many-sorted algebras in [12].)

So extra care will be needed in manipulating type assignments when empty types occur.

5 Completeness

Our proof systems will differ from the usual ones (for models without empty types) in two respects: we carefully specify the rules for variable elimination, and we add an axiom scheme and inference rule for reasoning about empty types.

The rule for empty types is needed because even when the technical problem of variable discharge noted above is repaired to yield a sound lambda theory for empty types, the remaining standard systems are still not complete.

Theorem 5.1 *The standard axioms and equational inference rules which are complete for simply typed [10] or polymorphic [4] lambda calculus when all types are nonempty, are* not complete *for proving semantic consequences of equations over models with empty types.*

The proof is by a proof-theoretic analysis which we omit in this summary.

For example, let $\pi_1 ::= \lambda x{:}\, b_0.\, \lambda y{:}\, b_0.\, x$, $\pi_2 ::= \lambda x{:}\, b_0.\, \lambda y{:}\, b_0.\, y$, and $\mathbf{f}$ be a constant of suitable type. Then the equation

$$\lambda x{:}\, b_0.(\mathbf{f}\pi_1) = \lambda x{:}\, b_0.(\mathbf{f}\pi_2)$$

implies

$$(\mathbf{f}\pi_1) = (\mathbf{f}\pi_2).$$

This follows by the same reasoning which led to (9) if b_0 is not empty, and if b_0 is empty, then $\pi_1 = \pi_2$, and again it follows trivially.

However, argument by cases like this cannot be formalized without new inference rules. This is what distinguishes the proof systems we describe below from previous proof systems for equality in models without empty types [4]. In particular as noted above, our equations use the additional formulas $empty(\sigma)$ in type assignments.

In the appendix we give a complete proof system for equations with empty types (Theorem A.4). The proof, which we omit in this preliminary report, follows the usual proof of equational completeness by construction of term models, with an added twist resembling "model completion" in a Henkin-style completeness proof for predicate logic. The twist is brought on by the nonexistence of generic models noted in Proposition (2.1).

A Appendix: proof rules

A.1 Without empty types

For ordinary typed lambda terms, we have the usual axioms.

(α_1) $$A \mapsto \lambda x{:}\,\sigma.M = \lambda y{:}\,\sigma[y/x]M{:}\,\sigma \to \tau \quad \text{provided } y \notin FV(M)$$

(β_1) $$A \mapsto (\lambda x{:}\,\sigma.M)N = [N/x]M{:}\,\tau$$

(η_1) $$A \mapsto \lambda x{:}\sigma.Mx = M{:}\sigma \to \tau \quad \text{provided } x \notin FV(M)$$

For polymorphic lambda calculus, we need additional versions of each of these rules.

(α_2) $$A \mapsto \lambda t.M = \lambda s.[s/t]M{:}\forall t.\sigma \quad \text{provided } s \notin FV(A \mapsto M{:}\sigma)$$

(β_2) $$A \mapsto (\lambda t.M)\sigma = [\sigma/t]M{:}\tau$$

(η_2) $$A \mapsto \lambda t.Mt = M{:}\forall t.\sigma \quad \text{provided } t \notin FV(M)$$

The inference rules which are sound for all models are symmetry and transitivity

(sym) $$\frac{A \mapsto M = N{:}\sigma}{A \mapsto N = M{:}\sigma}$$

$(trans)$ $$\frac{A \mapsto M = N{:}\sigma,\ A \mapsto N = P{:}\sigma}{A \mapsto M = P{:}\sigma}$$

congruence rules, and a rule for adding additional hypotheses to type assignments. The congruence rules for ordinary typed lambda calculus are

$(cong_1)$ $$\frac{A \mapsto M_1 = M_2{:}\sigma \to \tau,\ A \mapsto N_1 = N_2{:}\sigma}{A \mapsto M_1N_1 = M_2N_2{:}\tau}$$

(ξ_1) $$\frac{A[x{:}\sigma] \mapsto M = N{:}\tau}{A \mapsto \lambda x{:}\sigma.M = \lambda x{:}\sigma.N{:}\sigma \to \tau}$$

The additional rules for polymorphic terms are

$(cong_2)$ $$\frac{A \mapsto M_1 = M_2{:}\forall t.\sigma,\ \tau_1 = \tau_2}{A \mapsto M_1\tau_1 = M_2\tau_2{:}[\tau_1/t]\sigma}$$

(ξ_2) $$\frac{A \mapsto M = N{:}\sigma}{A \mapsto \lambda t.M = \lambda t.N{:}\forall t.\sigma} \quad t \text{ not free in } A.$$

Since type assignments are explicitly included in equations, we also need the rule

$(add\ hyp)$ $$\frac{A \mapsto M = N{:}\sigma}{B \mapsto M = N{:}\sigma},\quad A \subseteq B$$

for adding additional typing hypotheses.

We write $\vdash_1$ for provability using the axiom schemes (α_1), (β_1), (η_1), and the inference rules $(sym), (trans), (cong_1), (\xi_1)$, and $(add\ hyp)$, and write $\vdash_1^{nonempty}$ for provability using *(nonempty)* in addition. We write $\vdash_2$ and $\vdash_2^{nonempty}$ for the corresponding proof systems for polymorphic lambda calculus. We omit subscripts to refer ambiguously to either (or both) proof systems.

Lemma A.1 *Without equational hypotheses, rule (nonempty) is a derived rule,* i.e., *for any equation* E

$$\vdash E \quad \text{iff} \quad \vdash^{\text{nonempty}} E.$$

Theorem A.2 [4] *The rules for* $\vdash^{\text{nonempty}}$ *are sound and complete for deducing semantic consequences of equations over the class of models in which every type is nonempty.*

Corollary A.3 *The rules for* $\vdash$ *are sound and complete for deducing the equations which hold in all models.*

A.2 With empty types

The proof system for reasoning about empty types makes use of the formulas $empty(\sigma)$ in equations. We have an axiom scheme for introducing equations that use emptiness assertions

$$empty(\sigma), x{:}\sigma \mapsto True = False{:}\, polybool$$

and an inference rule which lets us use emptiness assertions to reason by cases

$$\frac{A \cup \{x{:}\sigma\} \mapsto M = N{:}\tau,\ A \cup \{empty(\sigma)\} \mapsto M = N{:}\tau}{A \mapsto M = N{:}\tau} \quad x \text{ not free in } M, N$$

We write $\vdash^{empty}$ for provability using $\vdash$ and the axiom and inference rule for empty types.

The semantics of polymorphic models, in particular the meaning of satisfaction, $\models$, follows [4], except of course that types may be empty. Our main result is that $\vdash^{empty}$ is a sound and complete proof system for deducing semantic consequences of equations:

Theorem A.4 *Let Γ be a set of equations and E be any equation. Then*

$$\Gamma \vdash^{\text{empty}} E \quad \text{iff} \quad \Gamma \models^{\text{empty}} E.$$

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in Logic*, North-Holland, 1981. Revised Edition, 1984.

[2] V. Breazu-Tannen and T. Coquand. Extensional models for polymorphism. 1986. To appear, TAPSOFT '87 – CFLP.

[3] V. Breazu-Tannen and A. R. Meyer. Computable values can be classical. 1986. These Proceedings.

[4] K. B. Bruce and A. R. Meyer. The semantics of second order polymorphic lambda calculus. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 131–144, Springer-Verlag, Berlin, June 1984.

[5] L. Cardelli. *A polymorphic calculus with type:type.* Technical Report, DEC System Research Center, 1985.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, December 1985.

[7] T. Coquand. Communication in the TYPES electronic forum (types@xx.lcs.mit.edu). 1986. April 14^{th}.

[8] T. Coquand and G. Huet. *Constructions: A Higher Order Proof System for Mechanizing Mathematics.* Rapport de Recherche 401, INRIA, Domaine de Voluceau, 78150 Rocquencourt, France, May 1985. Presented at EUROCAL 85, Linz, Austria.

[9] S. Fortune, D. Leivant, and M. O'Donnell. The expressiveness of simple and second-order type structures. *J. ACM*, 30(1):151–185, Jan. 1983.

[10] H. Friedman. Equality between functionals. In R. Parikh, editor, *Logic Colloqium, '73*, pages 22–37, Springer-Verlag, 1975.

[11] J. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieure.* PhD thesis, Université Paris VII, 1972.

[12] J. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *SIGPLAN Notices*, 17:9–17, 1982.

[13] P. Martin-Löf. An intuitionistic theory of types: predicative part. In F. Rose and J. Sheperdson, editors, *Logic Colloquium III*, pages 73–118, North-Holland, Amsterdam, July 1973.

[14] N. J. McCracken. *An Investigation of a Programming Language with a Polymorphic Type Structure.* PhD thesis, Syracuse University, Syracuse, New York, June 1979.

[15] A. R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87–122, Jan. 1982.

[16] A. R. Meyer and M. B. Reinhold. 'Type' is not a type: preliminary report. In *Conf. Record Thirteenth Ann. Symp. Principles of Programming Languages*, pages 287–295, ACM, Jan. 1986.

[17] J. C. Mitchell. *Lambda Calculus Models of Typed Programming Languages.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, Aug. 1984.

[18] E. Moggi. Communication in the TYPES electronic forum (types@xx.lcs.mit.edu). 1986. February 10^{th}.

[19] C. Mohring. Algorithm development in the theory of constructions. In *Symp. Logic in Computer Science*, pages 84–91, IEEE, 1986.

[20] J. C. Reynolds. Three approaches to type structure. In *TAPSOFT advanced seminar on the role of semantics in software development*, Springer-Verlag, Berlin, 1985.

[21] J. C. Reynolds. Towards a theory of type structure. In *Coll. sur la programmation*, pages 408–423, Springer-Verlag, 1974.

[22] D. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.

[23] R. Statman. Completeness, invariance and lambda-definability. *J. Symbolic Logic*, 47:17–26, 1982.