Extensional Models for Polymorphism

Val Breazu-Tannen
MIT Laboratory for Computer Science
545 Technology Sq., Cambridge, MA 02139, USA

Thierry Coquand
INRIA
Domaine de Voluceau, 78150 Rocquencourt, France

Abstract. We present a general method for constructing extensional models for the polymorphic lambda calculus—the *polymorphic extensional collapse*. The method yields models that satisfy additional, computationally motivated constraints like having only two polymorphic booleans and having only the numerals as polymorphic integers. Moreover the method yields models that prove that the polymorphic lambda calculus can be *conservatively* added to arbitrary algebraic data type specifications, even with complete transfer of the computational power to the added data types.

1 Introduction

The design of functional and object-oriented programming languages has recently witnessed the widespread adoption of polymorphic type systems. A list of examples that is by no means exhaustive includes, in addition to the archetype ML [Gordon *et al.* 1979], such languages as Miranda [Turner 1985], Poly [Matthews 1985], Amber [Cardelli 1985], polymorphic FQL [Nikhil 1984], Ponder [Fairbairn 1982], and Hope [Burstall *et al.* 1980], while an excellent survey of the field is provided by [Cardelli & Wegner 1985].

We adopt the Girard-Reynolds polymorphic lambda calculus (henceforth denoted $\lambda^\forall$) as a formal setting for studying properties of such languages. Our concern here will be to construct models for $\lambda^\forall$ that satisfy additional constraints computationally motivated. These constraints have to do with the interaction between the type discipline of $\lambda^\forall$ and the data types with which we compute, eg. integers, booleans etc. An example follows.

Consider the representation of the integers in $\lambda^\forall$ where the numerals are taken to be the closed terms of type

$$polyint \stackrel{\text{def}}{=} \forall t.\, (t \to t) \to t \to t\ .$$

The numeral corresponding to the integer n is

$$\tilde{n} \stackrel{\text{def}}{=} \lambda t.\, \lambda f{:}t \to t.\, \lambda x{:}t.\, f^n x\ .$$

The first author was supported in part by an IBM Graduate Fellowship and in part by NSF Grant DCR-8511190

One can then define

$$Add \stackrel{\text{def}}{=} \lambda u{:}\,polyint.\ \lambda v{:}\,polyint.\ \lambda t.\ \lambda f{:}\,t \to t.\ \lambda x{:}\,t.\ utf(vtfx) \quad : polyint \to polyint \to polyint$$

and verify that $\lambda^\forall$ proves

(1) $$Add\,\widetilde{m}\,\widetilde{n} = \widetilde{m+n}\ .$$

The arithmetic functions that are numeralwise representable in the same way addition is represented above are exactly the functions which are provably total recursive in second-order Peano arithmetic [Girard 1972], [Statman 1981], [Fortune *et al.* 1983]. To date, no "natural" examples of total recursive functions that are not in this class are known and one can argue that such computational power is adequate for most purposes [Leivant 1983], [Reynolds 1985]. Therefore it appears that $\lambda^\forall$ can be regarded as a programming language already equipped with a type of integers and, as it turns out, also with one of booleans:

$$polybool \stackrel{\text{def}}{=} \forall t.\ t \to t \to t$$

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

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

as well as many other familiar data types [Reynolds 1985].

There is a problem, though. While we would like to reason about the terms of type *polyint* as if they actually *are* the integers, the pure $\lambda^\forall$ is *not* sufficient for that. For example, by a simple Church-Rosser argument,

(2) $$Add\,u\,v = Add\,v\,u$$

with arbitrary $u, v{:}\,polyint$ is *not* provable in $\lambda^\forall$. But if u and v are *numerals* then the equation (2) follows from (1). Hence the question: is it consistent to assume equation (2) as a further axiom of $\lambda^\forall$? This would follow from the existence of a model in which the *only* elements of type *polyint* are the (denotations of the) numerals. [1]

Such a question (actually for *polybool*) was asked in [Meyer 1986]. A positive answer was obtained in two ways using two different model constructions [Moggi 1986a], [Coquand 1986]. Both constructions used partial equivalence relations to interpret types, suggesting that there might be some relationship between them and, indeed, a common generalization was found [Breazu-Tannen 1986].

We are going to present this general construction, which we call *polymorphic extensional collapse*, that has as particular cases both the HEO-like model construction [Mitchell 1986b] [2] (the one used in [Moggi 1986a]) and the closed type/closed term model construction (the one used in [Coquand 1986]).

Another application of our general construction is to show that arbitrary algebras can be fully and faithfully embedded in models of $\lambda^\forall$. Moreover, such embeddings can be achieved in a manner that connects the computational power of $\lambda^\forall$ to the embedded algebras. The *conservative extension* theorems of [Breazu-Tannen & Meyer 1987] then follow as corollaries from our full and faithful embedding results.

The investigation of these conservative extension situations started from the same concern that suggested the question asked in [Meyer 1986], namely, is it possible to have data types with

[1] This account is inspired from [Meyer *et al.* 1987] where another example, involving *polybool*, is presented.

[2] [Mitchell 1986b] also contains a short history of the idea of interpreting types using partial equivalence relations. We want to add one reference to Mitchell's account, namely [Gandy 1956], which introduced the extensional collapse model construction for simple types, later called the "Gandy hull" in [Statman 1980].

"classical" properties live in a computational framework? In [Breazu-Tannen & Meyer 1987] it is remarked that unrestricted recursion is not compatible with "classical" properties. Then, computation done within the framework of the type discipline of $\lambda^\forall$ is offered as an alternative. However, unlike in the above discussion, one does not use the built-in integers, booleans, etc.; instead one *adds* such data types to $\lambda^\forall$ as algebraic data type specifications. The advantage is that we can postulate for these added data types whatever equations we wish, so that problems like the one with equation (2) do not arise.

The conservative extension theorems assure us that we can continue to reason about algebraic data expressions "classically", i.e., using the data type specification, even when these expressions occur in the computational framework provided by $\lambda^\forall$.

It is possible that there are some connections between the model constructions we describe here and the work reported in [Scedrov 1986]. However, [Scedrov 1986] does not give enough details for us to be able to verify this yet.

Finally, we should mention that detailed proofs of most of the new results mentioned here are also included in [Breazu-Tannen].

2 The polymorphic lambda calculus

2.1 Syntax

We assume we have an infinite set of *type variables*. By convention, t will range over type variables while σ, τ will range over *type expressions* which are defined by the grammar:

$$\tau ::= t \mid \sigma \to \tau \mid \forall t.\, \sigma\ .$$

We identify type expressions that differ only in the names of bound variables. The set of free type variables of σ will be denoted $fv(\sigma)$.

We also assume we have a separate infinite set of *variables*. By convention, x will range over variables while M, N will range over *raw terms* which are defined by the grammar:

$$M ::= x \mid MN \mid \lambda x{:}\sigma.\, M \mid M\sigma \mid \lambda t.\, M\ .$$

Again, we identify raw terms that differ only in the names of bound variables. The set of free variables of the term M will be denoted $FV(M)$ while the set of free *type* variables of M will be denoted $fv(M)$.

Type assignments are partial functions that map variables to type expressions and that have *finite* domain. Alternatively, we can regard type assignments as finite sets of pairs $x{:}\sigma$ such that no x occurs twice. We will use Δ to range over type assignments. When we write $\Delta, x{:}\sigma$ we mean a type assignment Δ' that contains $x{:}\sigma$ and such that $\Delta = \Delta' \setminus \{x{:}\sigma\}$.

Typing judgments have the form $\Delta \vdash M : \sigma$. Here are the type-checking rules:

(*variable introduction*) $\qquad x{:}\sigma \vdash x : \sigma$

(*extension*) $\qquad \dfrac{\Delta \vdash M : \sigma}{\Delta, x{:}\tau \vdash M : \sigma} \quad x \notin dom\Delta$

($\rightarrow$ *introduction*) $$\frac{\Delta, x{:}\sigma \vdash M : \tau}{\Delta \vdash \lambda x{:}\sigma.\, M : \sigma \rightarrow \tau}$$

($\rightarrow$ *elimination*) $$\frac{\Delta \vdash M : \sigma \rightarrow \tau \quad \Delta \vdash N : \sigma}{\Delta \vdash MN : \tau}$$

($\forall$ *introduction*) $$\frac{\Delta \vdash M : \sigma}{\Delta \vdash \lambda t.\, M : \forall t.\, \sigma} \quad t \notin fv(ran\Delta)$$

($\forall$ *elimination*) $$\frac{\Delta \vdash M : \forall t.\, \sigma}{\Delta \vdash M\tau : \sigma[t := \tau]}$$

Equations have the form

$$\Delta \,.\, M = N : \sigma$$

where the role of Δ and σ is to help check that the equational reasoning is type-correct.

The *core* proof system for deriving equations, denoted $\lambda^{\forall}$, consists of

(*extension*) $$\frac{\Delta \,.\, M = N : \sigma}{\Delta, x{:}\tau \vdash M = N : \sigma} \quad x \notin dom\Delta$$

reflexivity, symmetry, transitivity, congruence w.r.t. application,

(ξ) $$\frac{\Delta, x{:}\sigma \,.\, M = N : \tau}{\Delta \,.\, \lambda x{:}\sigma.\, M = \lambda x{:}\sigma.\, N : \sigma \rightarrow \tau}$$

β, η, congruence w.r.t. polymorphic application, *type* ξ,

(*type* β) $$\Delta \,.\, (\lambda t.\, M)\tau = M[t := \tau] : \sigma[t := \tau]$$

where $\Delta \vdash M : \sigma$, $t \notin fv(ran\Delta)$

(*type* η) $$\Delta \,.\, \lambda t.\, Mt = M : \forall t.\, \sigma$$

where $\Delta \vdash M : \forall t.\, \sigma$, $t \notin fv(ran\Delta)$, $t \notin fv(M)$

As usual, conversion can be analyzed by a reduction system. We will call "$\beta\eta$-reduction" the notion of reduction consisting of both the "regular" and the "type" β and η. The definition is omitted here.

2.2 Semantics

An *algebra of polymorphic types*, $\mathcal{T}$, consists of the following:

- a non-empty set T of *types*;
- a binary operation $\rightarrow$ on T;
- a non-empty set $[T \Rightarrow T]$ of functions from T to T;
- a map $\forall$ from $[T \Rightarrow T]$ to T;

such that the following inductive definition of an assignment of meanings in T to type expressions in type environments is possible (we define a *type environment* to be a map from type variables to T and we will use η to range over type environments):

1. $[\![t]\!]\eta = \eta(t)$
2. $[\![\sigma \rightarrow \tau]\!]\eta = [\![\sigma]\!]\eta \rightarrow [\![\tau]\!]\eta$
3. $[\![\forall t.\, \sigma]\!]\eta = \forall(\lambda a \in T.\, [\![\sigma]\!]\eta\{t := a\})$

By "the definition is possible" we understand that each inductive application of step 3 is defined, i.e., $\lambda a \in T.\, [\![\sigma]\!]\eta\{t := a\} \in [T \Rightarrow T]$. Here $\eta\{t := a\}$ is the type environment equal to η everywhere except at t where it takes the value a.

A *polymorphic lambda interpretation* (p.l.i.) consists of the following:

- an algebra of polymorphic types, $\mathcal{T}$;
- a set D_a for each type $a \in T$ (the *domain* of a);
- a binary operation $\cdot_{ab} : D_{a \rightarrow b} \times D_a \longrightarrow D_b$ for each pair of types $a, b \in T$ (functional application);
- a binary operation $\cdot_\phi : D_{\forall(\phi)} \times T \longrightarrow \bigcup\{D_a\}$ for each function $\phi \in [T \Rightarrow T]$, such that $p \cdot_\phi a \in D_{\phi(a)}$ (polymorphic application);

Given a type assignment Δ and a type environment η, we define an *$\Delta\eta$-environment* to be a function ρ that maps $dom\Delta$ to $\bigcup\{D_a\}$ such that $\rho(x) \in D_{[\![\Delta(x)]\!]\eta}$ for each variable $x \in dom\Delta$. As with type assignments, we will regard $\Delta\eta$-environments as finite sets of pairs $x{:}\, d$, extending to them the notational convention $\rho, x{:}\, d$. With this, the final component of a polymorphic lambda interpretation is

- a "meaning" map that assigns to every typing judgment $\Delta \vdash M : \sigma$ that is *derivable* and every type environment η a map $[\![\Delta \vdash M : \sigma]\!]\eta$ from $\Delta\eta$-environments to $D_{[\![\sigma]\!]\eta}$ such that

  1. $[\![x{:}\,\sigma \vdash x : \sigma]\!]\eta\rho = \rho(x)$

  2. $[\![\Delta, x{:}\,\tau \vdash M : \sigma]\!]\eta\rho = [\![\Delta \vdash M : \sigma]\!]\eta\rho'$

     where $x \notin dom\Delta$ and $\rho' \stackrel{\text{def}}{=} \rho\,|_{dom\Delta}$

3. $[\![\Delta \vdash MN : \tau]\!]\eta\rho = [\![\Delta \vdash M : \sigma \to \tau]\!]\eta\rho \cdot_{ab} [\![\Delta \vdash N : \sigma]\!]\eta\rho$

   where $a \stackrel{\text{def}}{=} [\![\sigma]\!]\eta$ and $b \stackrel{\text{def}}{=} [\![\tau]\!]\eta$

4. $[\![\Delta \vdash \lambda x{:}\sigma.\, M : \sigma \to \tau]\!]\eta\rho \cdot_{ab} d = [\![\Delta, x{:}\sigma \vdash M : \tau]\!]\eta\rho'$

   where $a \stackrel{\text{def}}{=} [\![\sigma]\!]\eta$, $b \stackrel{\text{def}}{=} [\![\tau]\!]\eta$, $d \in D_a$ and $\rho' \stackrel{\text{def}}{=} \rho, x{:}d$

5. $[\![\Delta \vdash M\tau : \sigma[t{:=}\tau]]\!]\eta\rho = [\![\Delta \vdash M : \forall t.\, \sigma]\!]\eta\rho \cdot_{\phi} [\![\tau]\!]\eta$

   where $\phi \stackrel{\text{def}}{=} \lambda a \in T.\, [\![\sigma]\!]\eta\{t{:=}a\}$

6. $[\![\Delta \vdash \lambda t.\, M : \forall t.\, \sigma]\!]\eta\rho \cdot_{\phi} a = [\![\Delta \vdash M : \sigma]\!]\eta\{t{:=}a\}\,\rho$

   where $a \in T$ and $\phi \stackrel{\text{def}}{=} \lambda a \in T.\, [\![\sigma]\!]\eta\{t{:=}a\}$

7. if $\forall t \in fv(ran\Delta) \cup fv(M),\ \eta(t) = \eta'(t)$ then $[\![\Delta \vdash M : \sigma]\!]\eta = [\![\Delta \vdash M : \sigma]\!]\eta'$

An equation $\Delta\ .\ M = N{:}\sigma$ is *valid* in a polymorphic lambda interpretation when $[\![\Delta \vdash M{:}\sigma]\!]\eta = [\![\Delta \vdash N : \sigma]\!]\eta$ for every η.

A p.l.i. is a quite general concept. For example, not even basic axioms like β are necessarily valid in arbitrary p.l.i.'s.

A *model* of the polymorphic lambda calculus is a polymorphic lambda interpretation in which functional and polymorphic applications are *extensional*:

$$(\forall d \in D_a, f \cdot_{ab} d = g \cdot_{ab} d) \Longrightarrow f = g \quad f, g \in D_{a \to b}$$

$$(\forall a \in T, p \cdot_{\phi} a = q \cdot_{\phi} a) \Longrightarrow p = q \quad p, q \in D_{\forall(\phi)}\ .$$

This definition is equivalent and, in fact, very close to the one in [Bruce & Meyer 1984].

A model is *trivial* when all its domains have at most one element. It is not hard to check that a model is trivial if and only if it equates *True* and *False* (i.e., $.\,True = False : polybool$ is valid).

It is easy to see that the proof rules of the core system $\lambda^{\forall}$ are sound for this notion of model. As was recently explained in [Meyer *et al.* 1987], completeness is more complicated. One is, of course, interested in the strong kind of completeness, i.e., completeness of reasoning from additional premises. In [Bruce & Meyer 1984] such a result is stated, but it amounts to completeness of the core proof system extended with the rule:

(*discharging*)
$$\frac{\Delta, x{:}\sigma\ .\ M = N : \tau}{\Delta\ .\ M = N : \tau} \qquad x \notin FV(M) \cup FV(N)$$

for the subclass of models with all types non-empty. (Discharging is *not sound*, in general, in models that can have empty type domains.)

The model definition presented here allows empty types and, in fact, if additional constraints like having only two polymorphic booleans or having only the numerals as polymorphic integers are to hold, then some type domains *must* be empty [Meyer *et al.* 1987].

In [Meyer *et al.* 1987] it is stated that the core proof system $\lambda^{\forall}$ (no discharging), while sound, is not complete for the class of all models. The paper then gives an extension of the proof system

that is sound and complete for all models. This extension involves modifying the syntax of the equations to allow "type emptiness" assertions to be added to the type assignments as well as new axioms and inference rules. Our full and faithful embedding constructions (Subsection 4.3) imply the conservative extension results of [Breazu-Tannen & Meyer 1987] w.r.t. to this new extended proof system.

As far as the *model constructions* described in the present paper, we have noted that it does not matter which of the three proof systems we use to construct our closed term interpretations.

Indeed, by Church-Rosser arguments, discharging is a *derived rule* in the pure $\lambda^\forall$ theory or in theories axiomatized by additional equations that can be analyzed with *delta reduction* rules, like the ones we use in Subsection 4.3. Thus, the closed type/closed term constructions of Subsections 4.2 and 4.3 are the same as the ones that would be obtained using the extended proof system of [Meyer *et al.* 1987] or the proof system with discharging.

2.3 Logical relations

The concept of second-order logical relation was introduced in [Mitchell & Meyer 1985]. Here we will review only a particular case of this concept, namely, the case that we need for the polymorphic extensional collapse.

Given a polymorphic lambda interpretation, a *logical relation* on it is a family of binary relations $\mathcal{R} = \{R_a | a \in T, R_a \subseteq D_a \times D_a\}$ such that

$$f\ R_{a\to b}\ g \text{ iff } \forall d \forall e\ d\ R_a\ e \implies f \cdot_{ab} d\ R_b\ g \cdot_{ab} e$$

and

$$p\ R_{\forall(\phi)}\ q \text{ iff } \forall a\ p \cdot_\phi a\ R_{\phi(a)}\ q \cdot_\phi a\ .$$

Proposition 2.1 (Fundamental property of logical relations) *Assume we have a logical relation $\mathcal{R}$ on a polymorphic lambda interpretation. For any derivable typing judgment $\Delta \vdash M : \sigma$, any type environment η and any two $\Delta\eta$-environments ρ_1 and ρ_2, if*

$$\forall x \in dom\Delta,\quad \rho_1(x)\ R_{[\Delta(x)]\eta}\ \rho_2(x)$$

then

$$[\![\Delta \vdash M : \sigma]\!]\eta\rho_1\ R_{[\![\sigma]\!]\eta}\ [\![\Delta \vdash M : \sigma]\!]\eta\rho_2\ .$$

We will make essential use of this property in the polymorphic extensional collapse and, in fact, the definition of the concept of polymorphic lambda interpretation was engineered to consist of the "minimum necessary" to make the proof of Proposition 2.1 work.

3 Polymorphic extensional collapse

If D is a set, let $per(D)$ denote the set of *partial equivalence* (i.e., symmetric and transitive but not necessarily reflexive) *relations* (p.e.r.'s) on D. Let $R \in per(D)$. We denote by $R[d]$ the p.e.r. class of d w.r.t. R. Note that $R[d] \neq \emptyset$ iff $d\ R\ d$. The *quotient set* D/R is the set of all nonempty p.e.r. classes w.r.t R.

3.1 Factoring by a logical partial equivalence relation

Suppose that we have a polymorphic lambda interpretation I together with a logical relation $\mathcal{R}$ on it such that each R_a is a p.e.r. We call $\mathcal{R}$ a *logical p.e.r.* Then, we can construct a new polymorphic lambda interpretation, $I/\mathcal{R}$ (the *quotient* of I by $\mathcal{R}$), which is actually extensional, *i.e.*, a *model.*

The algebra of types will be the same. As domains we take the quotient sets D_a/R_a. Since $\mathcal{R}$ is logical, it is also a congruence w.r.t. functional and polymorphic application therefore application on the quotient sets is defined straightforwardly.

Claim. Both functional and polymorphic application are extensional.

For polymorphic application this is immediate. For functional application it can be seen that

$$\text{if } f\ R_{a\to b}\ f \text{ and } g\ R_{a\to b}\ g \text{ and } (\forall d\ f\cdot_{ab} d\ R_b\ g\cdot_{ab} d) \text{ then } f\ R_{a\to b}\ g\ .$$

To show how to define the meaning function let us fix a derivable typing statement $\Delta \vdash M : \sigma$ and a type environment η. For any $I/\mathcal{R}$-$\Delta\eta$-environment $\hat{\rho}$, choose an I-$\Delta\eta$-environment ρ such that

$$\forall x \in dom\Delta,\ \ \hat{\rho}(x) = R_{[\![\Delta(x)]\!]\eta}[\rho(x)]$$

and then define

$$[\![\Delta \vdash M : \sigma]\!]^{I/\mathcal{R}}\eta\hat{\rho} \stackrel{\text{def}}{=} R_{[\![\sigma]\!]\eta}[[\![\Delta \vdash M : \sigma]\!]^I\eta\rho]\ .$$

Indeed, by Proposition 2.1 the definition does not depend on the choice of ρ and also $[\![\Delta \vdash M : \sigma]\!]^I\eta\rho$ is related to itself hence its p.e.r. class is nonempty.

Any equation valid in I is valid in $I/\mathcal{R}$. The converse is in general not true since there can be pairs of closed terms whose meanings in I are distinct but *related* by $\mathcal{R}$ and therefore whose meanings in $I/\mathcal{R}$ are the same p.e.r. class.

If I is actually a *model* then the relation consisting of the identity on each of its types is actually logical (because of extensionality) and the model I/Id is isomorphic to I. Therefore, the class of models obtained as quotients of arbitrary polymorphic lambda interpretations by logical p.e.r.'s is the same as the class of all models.

3.2 Tagging the types with partial equivalence relations

In order to take advantage of the construction in the previous subsection, assuming that one has a p.l.i., how does one construct a logical p.e.r. on it so that one would then obtain a model by taking the quotient?

The idea is the same as the one behind Girard's "candidats de réductibilité": since we don't know in advance which p.e.r.'s we will need, we will put in all of them!

Starting with an arbitrary polymorphic lambda interpretation I we show how to construct a new one, I^{per}, that has a logical p.e.r. on it and, moreover, the same equations are valid in I and I^{per}.

First, from the algebra of types $\mathcal{T}$ construct $\mathcal{T}^{per}$ as follows:

- the new types, T^{per}, are pairs $<a, R>$ with $a \in T$ and $R \in per(D_a)$;

- $<a, R> \to <b, S> \stackrel{\text{def}}{=} <a \to b, R \to S>$ where $R \to S$ is defined by

$$f\ R \to S\ g \text{ iff } \forall d \forall e\ d\ R\ e \implies f \cdot_{ab} d\ S\ g \cdot_{ab} e$$

and is shown to be a p.e.r. also;

- $[T^{per} \Rightarrow T^{per}]$ consists of the functions determined (one-to-one) by pairs $<\phi, H>$ where $\phi \in [T \Rightarrow T]$ and H is a family of maps

$$H_a : per(D_a) \longrightarrow per(D_{\phi(a)})$$

one for each type $a \in T$;

- $\forall(<\phi, H>) \stackrel{\text{def}}{=} <\forall(\phi), \forall(H)>$ where $\forall(H)$ is defined by

$$p\ \forall(H)\ q \text{ iff } \forall <a, R> \in T^{per}\ \ p \cdot_\phi a\ H_a(R)\ q \cdot_\phi a\ .$$

and is, of course, also a p.e.r.

To check that this is an algebra of polymorphic types, one further shows that the inductive definition of meaning is possible. Moreover

$$proj_1([\![\sigma]\!]^{T^{per}} \eta) = [\![\sigma]\!]^T (proj_1 \circ \eta)\ .$$

The rest of the definition of the polymorphic lambda interpretation is by "taking the first projection of the types":

- $D_{<a,R>} \stackrel{\text{def}}{=} D_a$;
- $f \cdot_{<a,R><b,S>} d \stackrel{\text{def}}{=} f \cdot_{ab} d$;
- $p \cdot_{<\phi,H>} a \stackrel{\text{def}}{=} p \cdot_\phi a$;
- $[\![\Delta \vdash M : \sigma]\!]^{I^{per}} \eta \stackrel{\text{def}}{=} [\![\Delta \vdash M : \sigma]\!]^I (proj_1 \circ \eta)$.

Clearly, any equation valid in I is valid in I^{per}. The converse is also true since any T-type environment is the first projection of some T^{per}-type environment.

The benefit of all this is that now we have at each type $<a, R>$ a p.e.r., namely R, and, more importantly, this collection of p.e.r.'s, call it $\mathcal{R}^{per}$, is a *logical relation* on I^{per} as one can readily check.

We define the *polymorphic extensional collapse* of I to be the model $I^{per}/\mathcal{R}^{per}$.

The models obtained as polymorphic extensional collapses of arbitrary polymorphic lambda interpretations are not arbitrary at all. For example, it is not hard to see that in all of them the type $\forall t.\, t$ is *empty*. This implies the validity of certain non-trivial equations, for example

$$True\ \forall t.\, t = False\ \forall t.\, t\ .$$

An interesting open question is to obtain a characterization of the set of equations that are valid in all such models.

4 Applications

4.1 HEO-like models

These models are obtained by applying the polymorphic extensional collapse to "erase-types" polymorphic lambda interpretations, i.e., ones in which the terms are interpreted by first erasing the type information and then interpreting the resulting untyped terms in, say, some combinatory algebra (via the usual translation into combinatory terms; see [Barendregt 1984]).

We consider untyped lambda terms built from the same variables used for the polymorphic lambda terms. Again, we identify terms that differ only in the name of the bound variables.

An *untyped lambda interpretation* (called *pseudo-λ-structure* in [Hindley & Longo 1980]) , $\mathcal{U}$, consists of the following:

- a non-empty set D;
- a binary operation $\cdot$ on D (application);
- a "meaning" map that assigns to every untyped lambda term M and every D-environment π an element $[\![M]\!]\pi$ of D (we define a *D-environment* to be a map from variables to D and we will use π to range over D-environments) such that:

  1. $[\![x]\!]\pi = \pi(x)$
  2. $[\![MN]\!]\pi = [\![M]\!]\pi \cdot [\![N]\!]\pi$
  3. $[\![\lambda x.\, M]\!]\pi \cdot d = [\![M]\!]\pi\{x := d\}$ where $d \in D$
  4. if $\forall x \in FV(M),\ \pi_1(x) = \pi_2(x)$ then $[\![M]\!]\pi_1 = [\![M]\!]\pi_2$

If application is also extensional we get the usual concept of *model of the untyped λβη-calculus* [Hindley & Longo 1980], [Meyer 1982], [Barendregt 1984].

Any combinatory algebra yields an untyped lambda interpretation. Namely, the meaning map $[\![M]\!]\pi$ is defined by first translating M into a combinatory term [Barendregt 1984] and then interpreting the result in the algebra.

Now, starting from an arbitrary algebra of polymorphic types $\mathcal{T}$ and an arbitrary untyped lambda interpretation $\mathcal{U}$, we construct the "erase-types" polymorphic lambda interpretation , $\mathcal{I}^{\mathcal{T},\mathcal{U}}$ as follows:

The algebra of types is, of course, $\mathcal{T}$. The domains of all types are equal to D, the domain of $\mathcal{U}$. Functional application is given by the application in $\mathcal{U}$. Polymorphic application simply erases the type:

$$p \cdot_\phi a \stackrel{\text{def}}{=} p\ .$$

Finally, the meaning map is defined by

$$[\![\Delta \vdash M : \sigma]\!]^{\mathcal{I}^{\mathcal{U}}}\eta\rho \stackrel{\text{def}}{=} [\![Erase(M)]\!]^{\mathcal{U}}\pi$$

where π is some D-environment that takes the same values as ρ on $FV(M)$ (by (4) above, only these values matter) and $Erase(\lambda x{:}\sigma.\, M) = \lambda x.\, Erase(M)$, $Erase(M\sigma) = Erase(M)$, $Erase(\lambda t.\, M) = Erase(M)$, etc.

We then define the *HEO-like model* with parameters $\mathcal{T}$ and $\mathcal{U}$ to be the polymorphic extensional collapse of $\mathcal{I}^{\mathcal{T},\mathcal{U}}$.

The construction can be further generalized to start from *partial* combinatory algebras, just like the original HEO_2 model [3] [Girard 1972], [Troelstra (ed.) 1973], as was discovered by independently by Gordon Plotkin and Eugenio Moggi.

Moggi's ingenious construction [Moggi 1986a] of a model that has exactly two elements of type *polybool* amounts to—in the terminology of this paper—the HEO-like model whose parameters are the trivial algebra of types (just one type) and the open term model of the untyped lambda calculus.

Such HEO-like models whose second parameters are actually untyped lambda models (untyped application is already extensional) are particularly intriguing. This is because the corresponding "erase-types" polymorphic lambda interpretations are *already* models: functional application is extensional because the untyped application is while polymorphic application is always (trivially) extensional! However, these "erase-types" models are much too coarse: all elements have all types! The point of continuing with a polymorphic extensional collapse—as suggested by Moggi's idea—seems therefore to be the "pruning" of the model, while preserving extensionality.

Another connection is with the *PER models* of [Mitchell 1986b]. PER models are, in general, *not* obtained by (what we defined as) polymorphic extensional collapse (for example, one can construct PER models that have all types non-empty). Nonetheless, we have developed enough terminology to be able to say what they are.

Such models have three parameters: an algebra of polymorphic types $\mathcal{T}$, an untyped lambda interpretation $\mathcal{U}$ [4] and a logical p.e.r. $\mathcal{R}$ on the "erase-types" polymorphic lambda interpretation $\mathcal{I}^{\mathcal{T},\mathcal{U}}$. The model is then defined as the quotient $\mathcal{I}^{\mathcal{T},\mathcal{U}}/\mathcal{R}$.

Any HEO-like model is a PER-model. Indeed, given an HEO-like model with parameters $\mathcal{T}$ and $\mathcal{U}$, we note that $(\mathcal{I}^{\mathcal{T},\mathcal{U}})^{per} \equiv \mathcal{I}^{\mathcal{T}^{per},\mathcal{U}}$. Therefore this model is the same as the PER model with parameters $\mathcal{T}^{per}$, $\mathcal{U}$ and $\mathcal{R}^{per}$.

A characterization of the theories of PER models is given in [Mitchell 1986b]. It essentially says that the only additional equations that hold in PER models—compared to general models— are those obtained by equating terms that look the same when types are erased. It is an open question whether these are also the only additional equations that hold in HEO-like models—compared to general models obtained by polymorphic extensional collapse.

4.2 The closed type/closed term model construction

We start with the observation that the closed type expressions form an algebra of polymorphic types. Indeed, we can take $[T \Rightarrow T]$ to be the set of functions $\omega \longmapsto \sigma[t := \omega]$ determined by the closed polymorphic type expressions $\forall t.\, \sigma$.

Then, for each closed type expression ω we have an equivalence relation on the set of closed terms of type ω:

$$M \cong_\omega N \ \textit{iff}\ \lambda^\forall \vdash \ .\, M = N : \omega\ .$$

We take the domain of ω to consist of equivalence classes of closed terms of type ω modulo $\cong_\omega$. We can also think of this domain as the set of all closed normal forms of type ω. The definitions of application and the definition of the meaning map by substitution are straightforward. It is

[3] HEO_2 starts from the partial combinatory algebra of integers and "Kleene brackets" application

[4] Actually, Mitchell requires $\mathcal{U}$ to be a lambda model but the construction goes through for interpretations

easy to check that we get a polymorphic lambda interpretation (call it the *closed type/closed term polymorphic lambda interpretation*). Moreover, the elements of type *polyint*, for example, are in one to one correspondence with the numerals. Unfortunately, extensionality fails since, for example, there are no elements of type $\forall t.\ t$ but there are two *distinct* elements of type $(\forall t.t) \to (\forall t.t) \to (\forall t.t)$ namely (the equivalence classes of) $\lambda x{:}\forall t.t.\lambda y{:}\forall t.t.x$ and $\lambda x{:}\forall t.t.\lambda y{:}\forall t.t.y$ while by extensionality there should be at most one.

Hence the idea of [Coquand 1986] of combining the closed type/closed term construction with factoring by p.e.r.'s and thus achieving extensionality. This amounts to taking the polymorphic extensional collapse of the closed type/closed term polymorphic lambda interpretation. Let us call the resulting model $\mathcal{C}$. In [Coquand 1986] it is shown that in $\mathcal{C}$ there are exactly two elements of type *polybool* namely (the meanings of) *True* and *False*. In particular, $\mathcal{C}$ is non-trivial. Here we will show that in $\mathcal{C}$ the elements of type *polyint* are exactly the numerals. This will provide a positive answer to the question we asked in the introduction, as well as another proof that $\mathcal{C}$ is non-trivial.

Since the numerals are the only closed normal forms of type *polyint* we need only show that no further identifications between numerals take place.

Suppose that $\vdash \widetilde{m} = \tilde{n} : \mathit{polyint}$ is valid in $\mathcal{C}$. Then, for any closed type expression ω and for any p.e.r. R on the set of equivalence classes modulo $\cong_\omega$ we must have (recall that $\cong_\omega[M]$ is the equivalence class of M modulo $\cong_\omega$)

$$\cong_{(\omega\to\omega)\to\omega\to\omega}[\widetilde{m}\,\omega]\ \ (R\to R)\to R\to R\ \ \cong_{(\omega\to\omega)\to\omega\to\omega}[\tilde{n}\,\omega]\ .$$

Define

$$\mathit{Succ} \stackrel{\text{def}}{=} \lambda u{:}\mathit{polyint}.\,\mathit{Add}\,u\,\tilde{1}\ :\ \mathit{polyint}\to\mathit{polyint}\ .$$

Taking $\omega = \mathit{polyint}$ and R to be the identity we have

$$\mathit{Succ}\ R\to R\ \mathit{Succ}\quad\text{and}\quad \tilde{0}\ R\ \tilde{0}$$

hence

$$\widetilde{m}\ \mathit{polyint}\ \mathit{Succ}\,\tilde{0}\ \cong_{\mathit{polyint}}\ \tilde{n}\ \mathit{polyint}\ \mathit{Succ}\,\tilde{0}\ .$$

But

$$\widetilde{m}\ \mathit{polyint}\ \mathit{Succ}\,\tilde{0} = \mathit{Succ}^m\,\tilde{0} = \widetilde{m}$$

hence $\widetilde{m} \cong_{\mathit{polyint}} \tilde{n}$ which implies $m = n$.

Let ω be a closed type. A model is said to be *canonical* at ω if its elements of type ω are in one-to-one correspondence with the closed normal forms of type ω. So far, we have seen that $\mathcal{C}$ is canonical at *polybool* and *polyint*. As it was pointed out to us by Albert Meyer, it follows from a result in [Statman 1983] that this generalizes to all "such" types.

More precisely, let us define an *ML-polymorphic* type to be a closed type of the form $\forall t_1 \ldots \forall t_n.\sigma$ where σ is $\forall$-free [5]. As expected, the "combinatorics" of terms of ML-polymorphic type is essentially that of simply typed terms. The following is a result about simple lambda terms from [Statman 1983], rephrased for ML-polymorphic terms.

Statman's Typical Ambiguity Theorem *Let ω be an ML-polymorphic type and M, N two closed terms of type ω. If M and N are not $\beta\eta$-convertible then there exists a closed term $L : \omega \to \mathit{polybool}$ such that $L\,M = \mathit{True}$ and $L\,N = \mathit{False}$.*

Since $\mathcal{C}$ is non-trivial:

[5] These types correspond to the limited kind of polymorphism allowed in the language ML [Gordon *et al.* 1979]

Corollary 4.1 *The model $\mathcal{C}$ is canonical at all ML-polymorphic types.*

An interesting open problem is to characterize the theory of $\mathcal{C}$. A possibility, suggested by Rick Statman [Moggi 1986b], is that $\mathcal{C}$ is a *minimal* model for $\lambda^\forall$, i.e., that any equation that can be consistently added to $\lambda^\forall$ is valid in $\mathcal{C}$.

4.3 Full and faithful embeddings of algebras

This subsection is a spin-off from [Breazu-Tannen & Meyer 1987]. The results presented here can be thought of as model-theoretical versions of the conservative extension theorems stated there.

For the conservative extension theorem that corresponds to (and follows from) Theorem 4.2 below, an alternative, purely syntactic, proof is given in [Breazu-Tannen & Meyer 1987]. We know of no such syntactic proof for the conservative extension theorem corresponding to Theorem 4.3. In fact, [Breazu-Tannen & Meyer 1987] refers to Theorem 4.3 in this paper for a model-theoretical proof of the corresponding result.

Let A be an algebra with (say, just for simplicity) just one sort and a binary operation f. Given a model of $\lambda^\forall$, we say that (A, f) is *fully and faithfully embedded* in it if there exists a type alg and an element $f' \in D_{alg \to alg \to alg}$ such that when regarding f' as a binary operation on D_{alg} the resulting algebra (D_{alg}, f') is isomorphic to (A, f).

Theorem 4.2 *Any many-sorted algebra can be fully and faithfully embedded in a a model of the polymorphic lambda calculus.*

We are going to explain briefly how this is done. Given (again, for simplicity) an algebra (A, f) with one sort and a binary operation f, we construct first an extension of $\lambda^\forall$ by adding:

1. a type constant alg,
2. for each $a \in A$, a constant q_a and a type-checking axiom $\vdash q_a : alg$,
3. a constant f with $\vdash f : alg \to alg \to alg$,
4. for each $a, b \in A$, an equational axiom $\vdash f\ q_a\ q_b = q_{f(a,b)} : alg$.

Then, we construct the closed type/closed term polymorphic lambda interpretation of this extension. We claim that (A, f) is fully and faithfully embedded here at type alg. For this it is sufficient to see that any closed term of type alg converts to a term of the form q_a (true because the $\beta\eta$-normal forms of type alg are algebraic terms which then convert via the axioms (4) above to terms of the desired form) and that no two distinct terms of the form q_a are convertible to each other (true because the axioms (4) seen as reduction rules from left to right form together with β and η a Church-Rosser system—these are delta rules, see [Klop 1980]).

Next, we can see that the full and faithful embedding "survives" the subsequent polymorphic extensional collapse. This is because in the final model we have the liberty to choose a p.e.r for the interpretation of the type constant alg and we choose the *identity* p.e.r. Therefore, we have a full and and faithful embedding of (A, f) at type $<alg, identity>$.

In [Mitchell 1986b] it is stated that PER models can be used to obtain faithful but not full embeddings of algebras into models of the polymorphic lambda calculus. Subsequently, John

Mitchell and Eugenio Moggi have discovered how to do faithful *and full* embeddings of algebras into PER models that have empty types (actually HEO-like models) and, if the algebra has all sorts non-empty, even into PER models that have all types nonempty (these *cannot* be HEO-like models)[Mitchell 1986a]. The construction that we have presented here always yields models with empty types and had been obtained by us previously and independently.

While Theorem 4.2 is providing us complete information about the "pure" interaction between the polymorphic type discipline and algebraic data type specifications, its setting is not entirely satisfactory from the computer scientist's point of view. Indeed, the enormous computational power that $\lambda^\forall$ has over the numerals is only superficially used over the added data types. However, as shown in [Breazu-Tannen & Meyer 1987], there are *extensions* of $\lambda^\forall$ in which a strong connection between the computational power of $\lambda^\forall$ and the added data types can be set up and, moreover, the equational theory of the resulting language is still a conservative extension of the data type specifications. Theorem 4.3 below is a model-theoretic version of this result, showing that both full and faithful embedding and (semantic) computational power over the added data types can be simultaneously achieved. The corresponding conservative extension result from [Breazu-Tannen & Meyer 1987] follows as a corollary of the *proof* of Theorem 4.3.

We will assume that the added data type has a set of *observables*—say, character strings or lists or trees—and that we care only about computational behavior on the observables. We will also assume that there is some standard way to enumerate the observables with each observable appearing exactly once in the enumeration. This in turn will yield a one-to-one correspondence between functions on the integers and functions on the observables.

Theorem 4.3 *Let $\mathcal{A}$ be a many-sorted algebra. Let $c_0, c_1, c_2, \ldots$ be a sequence of distinct elements of some sort* obs *of $\mathcal{A}$, called* observables. *Then, there exists a model $\mathcal{E}$ of the polymorphic lambda calculus such that*

(i) $\mathcal{A}$ is fully and faithfully embedded in $\mathcal{E}$, and

(ii) every function on observables which is provably total recursive in second-order Peano arithmetic is in $\mathcal{E}$.

The construction of $\mathcal{E}$ is a refinement of the one we use for Theorem 4.2. Like there, we add a constant type for each sort of $\mathcal{A}$, constants for the elements and operations of $\mathcal{A}$ and axioms corresponding to the "tables" of these operations (items (1)–(4)). Moreover, here we also add the following items:

5. a constant In with $\vdash In : obs \to polyint$,
6. a constant Out with $\vdash Out : polyint \to obs$,
7. for each observable c_n, an axiom $. In\ q_{c_n} = \tilde{n} : polyint$,
8. for each element a of sort obs that is *not* an observable, an axiom $. In\ q_a = \tilde{0} : polyint$,
9. for each integer n, an axiom $. Out\ \tilde{n} = q_{c_n} : obs$.

Again, we construct the closed type/closed term polymorphic lambda interpretation of this extension. We claim that $\mathcal{A}$ is fully and faithfully embedded here and moreover the elements of type *polyint* are exactly the numerals.

To see this, note that conversion in the extended calculus can be analyzed with the reduction system consisting of β, η and the delta rules obtained by orienting the axioms, (4), (7), (8) and (9) from left to right. This system is strongly normalizable [Mitchell 1986b] and Church-Rosser [Klop 1980] and, using this, we can argue in the same spirit as in the proof of Theorem 4.2.

Like before, the full and faithful embedding survives the subsequent polymorphic extensional collapse. But, equally important, so do *In* and *Out*. Moreover, no identifications between distinct numerals take place (see the previous subsection). Since the collapse preserves the validity of axioms (7) and (9), we end up in the final model with two maps that establish a bijection between the observables and the numerals. This immediately implies the second part of the theorem.

Acknowledgments

We are grateful to John Mitchell for reading an earlier version and suggesting some corrections and improvements. Any remaining errors are, of course, our responsibility. The first author also benefited from a discussion on full and faithful embeddings with John Mitchell and Eugenio Moggi.

References

[Barendregt 1984] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, Amsterdam, second edition, 1984.

[Breazu-Tannen] V. Breazu-Tannen. Ph.D. thesis, MIT. Expected Feb.1987.

[Breazu-Tannen & Meyer 1987] V. Breazu-Tannen and A. R. Meyer. Computable values can be classical. In *Proceedings of the 14th Symposium on Principles of Programming Languages*, ACM, 1987. To appear.

[Breazu-Tannen 1986] V. Breazu-Tannen. Communication in the TYPES electronic forum (xx.lcs.mit.edu), July 29th. 1986. Unpublished.

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

[Burstall *et al.* 1980] R. M. Burstall, D.B. MacQueen, and D.T. Sanella. Hope: an experimental applicative language. In *LISP Conference*, pages 136–143, Stanford University Computer Science Department, 1980.

[Cardelli & Wegner 1985] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Cardelli 1985] L. Cardelli. Amber. In *Combinators and functional programming languages, Proceedings of the 13th Summer School of the LITP*, Le Val D'Ajol, Vosges, France, May 1985.

[Coquand 1986] T. Coquand. Communication in the TYPES electronic forum (xx.lcs.mit.edu), April 14th. 1986. Unpublished.

[Fairbairn 1982] J. Fairbairn. *Ponder and its type system.* Tech. Rep. 31, Computer Laboratory, Univ. of Cambridge, Cambridge, England, November 1982.

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

[Gandy 1956] R. O. Gandy. On the axiom of extensionality—Part I. *Journal of Symbolic Logic*, 21, 1956.

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

[Gordon *et al.* 1979] M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1979.

[Hindley & Longo 1980] R. Hindley and G. Longo. Lambda-calculus models and extensionality. *Zeitschrift für Mathematische Logic und Grundlagen der Mathematik*, 26:289–310, 1980.

[Klop 1980] J. W. Klop. *Combinatory reduction systems.* Tract 129, Mathematical Center, Amsterdam, 1980.

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

[Matthews 1985] D. C. J. Matthews. *Poly manual.* Tech. Rep. 63, Computer Laboratory, Univ. of Cambridge, Cambridge, England, 1985.

[Meyer *et al.* 1987] A. R. Meyer, J. C. Mitchell, E. Moggi, and R. Statman. Empty types in polymorphic λ-calculus. In *Proceedings of the 14th Symposium on Principles of Programming Languages*, ACM, 1987. To appear.

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

[Meyer 1986] A. R. Meyer. Communication in the TYPES electronic forum (xx.lcs.mit.edu), February 7th. 1986. Unpublished.

[Mitchell & Meyer 1985] J. C. Mitchell and A.R. Meyer. Second-order logical relations (extended abstract). In R. Parikh, editor, *Logics of Programs*, pages 225–236, Springer-Verlag, Berlin, June 1985.

[Mitchell 1986a] J. C. Mitchell. Personal communication, August. 1986. Unpublished.

[Mitchell 1986b] J. C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions. In *LISP Conference*, pages 308–319, ACM, New York, August 1986.

[Moggi 1986a] E. Moggi. Communication in the TYPES electronic forum (xx.lcs.mit.edu), February 10th. 1986. Unpublished.

[Moggi 1986b] E. Moggi. Communication in the TYPES electronic forum (xx.lcs.mit.edu), July 23rd. 1986. Unpublished.

[Nikhil 1984] R. S. Nikhil. *An incremental, strongly typed database query language.* Ph.D. thesis, Univ. of Pennsylvania, Philadelphia, August 1984. Available as tech. rep. MS-CIS-85-02.

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

[Scedrov 1986] A. Scedrov. Semantical methods for polymorphism. 1986. Unpublished manuscript, Univ. of Pennsylvania, July 1986.

[Statman 1980] R. Statman. On the existence of closed terms in the typed λ-calculus. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 511–534, Academic Press, New York, 1980.

[Statman 1981] R. Statman. Number theoretic functions computable by polymorphic programs. In *22nd Symposium on Foundations of Computer Science*, pages 279–282, IEEE, 1981.

[Statman 1983] R. Statman. λ-definable functionals and $\beta\eta$ conversion. *Arch. math. Logik*, 23:21–26, 1983.

[Troelstra (ed.) 1973] A. S. Troelstra (ed.). *Metamathematical investigation of intuitionistic arithmetic and analysis.* Volume 344 of *Lecture Notes in Mathematics*, Springer-Verlag, 1973.

[Turner 1985] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Functional programming languages and computer architecture*, pages 1–16, Springer-Verlag, Berlin, September 1985.