Outline of a proof theory of parametricity

*Harry G. Mairson**
Department of Computer Science
Brandeis University
Waltham, Massachusetts 02254

Abstract

Reynolds' Parametricity Theorem (also known as the Abstraction Theorem), a result concerning the model theory of the second order polymorphic typed λ-calculus (F_2), has recently been used by Wadler to prove some unusual and interesting properties of programs. We present a purely syntactic version of the Parametricity Theorem, showing that it is simply an example of formal theorem proving in second order minimal logic over a first order equivalence theory on λ-terms. We analyze the use of parametricity in proving program equivalences, and show that structural induction is still required: parametricity is not enough.

As in Leivant's transparent presentation of Girard's Representation Theorem for F_2, we show that algorithms can be extracted from the proofs, such that if a λ-term can be proven parametric, we can synthesize from the proof an "equivalent" parametric λ-term that is moreover F_2-typable. Given that Leivant showed how proofs of *termination*, based on inductive data types and structural induction, had computational content, we show that inductive data types are indeed parametric, hence providing a connection between the two approaches.

1 Introduction

In a recent interesting and beguiling paper, "Theorems for free!" Philip Wadler showed how to use Reynolds' Abstraction Theorem [Rey83] to prove program equivalences. "Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies."[Wad89] He gives the example of a function $r \in \forall A.A^* \to A^*$, where A^* denotes the type of lists over elements of type A, and claims for all $f \in X \to Y$ and $\ell \in X^*$, we must have the equality

$$\mathbf{map}_{XY}\ f\ (r_X\ \ell) = r_Y\ (\mathbf{map}_{XY}\ f\ \ell). \tag{1}$$

Intuitive justifications of such an equation are clear: the function r is constrained by its type to rearrange, duplicate, and remove elements of its input in a data-independent manner. However, a more formal means of justification does not appear obvious. What Wadler did was to identify Reynolds' Abstraction Theorem, heretofore used as a means of relating values in different models of programming languages, as a key to showing such equivalences: as such it becomes a tool for program verification and proving properties of programs. When a function is typable in the second-order polymorphic typed λ-calculus invented by Girard and Reynolds [Gir72, Rey74] (henceforth called F_2), Wadler showed that the interpretation of the function in the *frame models* of Bruce, Meyer, and Mitchell [BMM90] satisfies a technical condition called *parametricity.* Parametricity enforces the *denotational* equivalence of terms such as in equation (1) above.

*Supported by NSF Grant CCR-9017125, and grants from Texas Instruments and from the Tyson Foundation.

A variety of questions immediately come to mind concerning this program of deriving "free theorems," presumably so named because much of the work seems to be done effortlessly by the Abstraction Theorem, and because of the surprising absence of induction arguments. First and foremost: do we really need model theory and denotational semantics to prove such equivalences, either from the point of true mathematical necessity, or from the perspective of clarity? There is a lurking suspicion that program equivalences such as (1) above may be compromised by models that are not fully abstract, since λ-terms not β-convertible may be denotationally equated in a model that is not fully abstract. Furthermore, generations of computer scientists have been taught that the *append* function is associative, and that the *reverse* function computes an involution, all without recourse to model theory (see, for example, [BW88]): are these new theorems qualitatively different, and how so? In fact, proofs of the former theorems are standard examples of the use of structural induction. No induction arguments appear in Wadler's free theorems: are they implicitly hidden, unnecessary, or incorrectly absent? Finally, little attention is given by Wadler to the *language* in which programs are written, or at least to the programming *style*: observe F_2-typable functions do not admit unbounded recursion (i.e., a fixpoint operator), an unusual constraint on expressiveness. Wadler concludes his paper with a suggestion that this *terra incognita* be explored further.

In this paper, we show how to derive Wadler's free theorems in a purely syntactic way, without recourse to any model theory. The logical calculus we use is just second order logic over a first-order equational theory of λ-terms. The syntactic view not only does away with unnecessary model theory, it also clarifies the constructive (and indeed, computational) nature of parametricity arguments, as well as explaining what parts of such proofs are mathematical, and what parts are metamathematical. One consequence of this clarification is that we uncover the genuine need for *structural induction* in proving equivalences such as (1) above, underlining part of the hidden cost of free theorems. The proof theory we develop is a straightforward syntactic rendition of *logical relations*, inspired by the Curry-Howard "propositions as types" analogy [How80]. The Curry-Howard analogy has been used in analysis of the constructive nature of logical calculi, where types are propositions, and λ-terms are proofs. What is more or less irrelevant in this development is what the propositions are *about*: in this paper, we show the propositions can be interpreted as statements about how the term may be used.

We relate the syntactic presentation of parametricity to work of Daniel Leivant, who has given particularly transparent and simple explanations and extensions of Girard's Representation Theorem for F_2 [Gir72] as well as Gödel's Dialectica Theorem [Göd58]. In this framework, type inference for a term becomes merely the homomorphism of a proof in second order logic about the use of the term, in other words the familiar work of proving properties of programs. In his paper, "Contracting proofs to programs," Leivant shows that type inference in F_2 can be viewed as termination proofs for a programming language over inductive data types [Lei90].[1] For instance, we can code **map** as

$$\mathbf{map}\ f\ \mathbf{nil} \;=\; \mathbf{nil} \tag{2}$$
$$\mathbf{map}\ f\ (\mathbf{cons}\ x\ \ell) \;=\; \mathbf{cons}\ (f\ x)\ (\mathbf{map}\ f\ \ell) \tag{3}$$

in the usual style, and we demonstrate **map** is a total function by proving the theorem[2]

$$\mathcal{M}(\mathbf{map}) \equiv \forall AB.\forall f.(\forall x.A(x) \to B(fx)) \to \forall \ell.A^*(\ell) \to B^*(\mathbf{map}\ f\ \ell), \tag{4}$$

where for any unary relation T, we define

$$T^*(\ell) \equiv \forall P.(\forall yt.T(y) \to P(t) \to P(\mathbf{cons}\ y\ t)) \to P(\mathbf{nil}) \to P(\ell). \tag{5}$$

Of course, T^* is the familiar axiom of structural induction over lists of "type" T. Leivant showed how from the *proof* of $\mathcal{M}(\mathbf{map})$ we can extract an *algorithm* (λ-term) computing **map** which is

[1] This result also appears in [BB85].

[2] We write $\forall A\ldots$, using capital letters, to denote second-order quantification (respectively, quantification over type variables), and $\forall x\ldots$, using small letters, to denote first-order quantification over terms.

F_2-typable, such that substituting the λ-term for **map** satisfies the recursive equations used as the initial definition of the function, and indeed such equations can be checked constructively by β-reduction. The parametricity of the solutions follows immediately. We therefore argue that in some sense proving *termination* is conceptually prior to proving *parametricity*, in that the latter follows formally and logically from the former in the case of inductive data types. Complementing Leivant's extraction of programs from proofs, we show a similar contraction whereby parametric programs can be extracted from proofs of parametricity.

Finally, since part of the beauty of "free theorems" is the (relative!) absence of *induction*, we present a style of "free programs" which exhibit surprising expressive power in the absence of *recursion*. The style is not new, however we believe that the type-free presentation we give is particularly lucid, and should be completely understandable to functional programming enthusiasts who have not been immersed in the type theory literature.

The preference of the author for proof theory over model theory is merely a taking of sides in a longstanding philosophical debate about the nature and foundations of mathematics: for a short exposition of the two semantic traditions, sometimes ascribed to Tarski and Heyting, see [GLT89]. The reduction we give to *logic* we believe is primary, and has a long tradition including Boole (who reduced "laws of thought" to logic), Russell and Whitehead (who reduced mathematics to logic), and more recently Curry and Howard (who, unbeknownst to them, reduced computation to logic). What we present is firmly in the tradition of the Curry-Howard "propositions as types" analogy.

We wish to emphasize our belief that there is no "right answer" in this debate. As a consequence, the work presented here is not meant to contradict Wadler's results, rather to complement them: we develop a simple and constructive framework for proving equalities about programs. Among the virtues of this constructive framework is that one could easily envision a formal proof checking system in which these equalities could be developed. Not coincidentally, it is this kind of goal which prompted the initial development of ML. What we are trying to explain simply and in a straightforward manner is that programs are just first order equational theories, and polymorphic types are theorems in second order logic about those theories. A not very surprising consequence is that types can be used to prove properties of programs.

As background references, for a description of F_2, we recommend [GLT89, PDM89, Sc90, Rey90]; for relevant ideas and notation from logic, especially sequent calculus and natural deduction, we recommend [vanD79]. A good introduction to logical relations can be found in [Sta85, Mit91].

2 Constructing proofs of parametricity from type inferences

The slogan generally associated with *parametricity* is that "programs evaluated in related environments have related results." We now formalize this notion in a very straightforward manner, using binary relations and second-order logic. Given the derivation of a type judgement $\Gamma \triangleright E{:}\,\sigma$ using the type inference rules of F_2, we can synthesize a proof concerning the parametric properties of E. Said otherwise, type judgements are homomorphic contractions of parametricity proofs.

Definition 2.1 *If E is an (untyped) λ-term, we define E' to be the term derived by replacing each free variable x of E by x'. Similarly, we define E'' by replacing each free variable x of E by x''.*

Definition 2.2 *If σ is an F_2 type and E is a λ-term, we define the proposition $\mathcal{R}^\sigma(E', E'')$ inductively as:*

$$\mathcal{R}^A(E', E'') \equiv A(E', E'') \qquad \text{if } \sigma \equiv A \text{ is a type variable} \tag{6}$$

$$\mathcal{R}^{\sigma\to\tau}(E', E'') \equiv \forall x'x''.\mathcal{R}^\sigma(x', x'') \to \mathcal{R}^\tau((Ex)', (Ex)'') \tag{7}$$

$$\mathcal{R}^{\forall A.\sigma}(E', E'') \equiv \forall A.\mathcal{R}^\sigma(E', E''). \tag{8}$$

The propositions in the above Definition specify relations between terms E' and E'' that are identical except for their free variables. The slogan associated with parametricity now takes on particular meaning in that we can consider such propositions as *hypotheses* about terms (i.e., an environment), and deduce similar propositions about other terms. This perspective motivates the next definition.

Definition 2.3 *If Γ is a type context, we define the set $\tilde{\Gamma}$ of propositions as:*

$$\tilde{\Gamma} = \{\mathcal{R}^{\sigma}(x', x'') \mid x{:}\,\sigma \in \Gamma\}. \tag{9}$$

Each kind of type inference rule can now be interpreted as the contraction of a second order logical deduction. We consider in turn each of the F_2 inference rules.

Rule (*var*). For the type inference rule

$$(var) \qquad \frac{}{\Gamma \cup \{x{:}\,\alpha\} \rhd x{:}\,\alpha}$$

we have the logical rule

$$[hyp] \qquad \frac{}{\tilde{\Gamma} \cup \{\mathcal{R}^{\alpha}(x', x'')\} \vdash \mathcal{R}^{\alpha}(x', x'')}$$

Rule (*app*). For the type inference rule

$$(app) \qquad \frac{\Gamma \rhd M{:}\,\alpha \to \beta \qquad \Gamma \rhd N{:}\,\alpha}{\Gamma \rhd MN{:}\,\beta}$$

we have the logical rule

$$[\to E] \qquad \frac{\tilde{\Gamma} \vdash \forall x'x''.\mathcal{R}^{\alpha}(x', x'') \to \mathcal{R}^{\beta}((Mx)', (Mx)'') \qquad \tilde{\Gamma} \vdash \mathcal{R}^{\alpha}(N', N'')}{\tilde{\Gamma} \vdash \mathcal{R}^{\beta}(M'N', M''N'')}$$

Rule (*abs*). For the type inference rule

$$(abs) \qquad \frac{\Gamma \cup \{x{:}\,\alpha\} \rhd M{:}\,\beta}{\Gamma \rhd \lambda x.M{:}\,\alpha \to \beta}$$

we have the logical rule

$$[\to I] \qquad \frac{\tilde{\Gamma} \cup \{\mathcal{R}^{\alpha}(x', x'')\} \vdash \mathcal{R}^{\beta}(M', M'')}{\tilde{\Gamma} \vdash \forall x'x''.\mathcal{R}^{\alpha}(x', x'') \to \mathcal{R}^{\beta}((\lambda x.M)'x', (\lambda x.M)''x'')}$$

Rule (*inst*). For the type inference rule

$$(inst) \qquad \frac{\Gamma \rhd M{:}\,\forall A.\Phi(A)}{\Gamma \rhd M{:}\,\Phi(\sigma)}$$

we have the logical rule

$$[\forall^2 E] \qquad \frac{\tilde{\Gamma} \vdash \forall A.\mathcal{R}^{\Phi(A)}(M', M'')}{\tilde{\Gamma} \vdash \mathcal{R}^{\Phi(\sigma)}(M', M'')}$$

Rule (*gen*). Finally, for the type inference rule

$$(gen) \qquad \frac{\Gamma \rhd M{:}\,\Phi(A) \qquad [A \notin FV(\Gamma)]}{\Gamma \rhd M{:}\,\forall A.\Phi(A)}$$

we have the logical rule

$$[\forall^2 I] \qquad \frac{\tilde{\Gamma} \vdash \mathcal{R}^{\Phi(A)}(M', M'') \qquad [A \notin FV(\tilde{\Gamma})]}{\tilde{\Gamma} \vdash \mathcal{R}^{\forall A.\Phi(\sigma)}(M', M'')}$$

Comments.

1. Unlike the usual presentation of type inference rules for F_2, we have omitted any type information in the λ-terms. For instance, we write M instead of $M\,[\sigma]$ in the conclusion to rule (*inst*), and M instead of $\Lambda A.M$ in the conclusion to rule (*gen*). The reason is to separate cleanly type information from their associated pure λ-terms, to parallel the logical deductions about such terms.

2. Observe that the logical interpretation of the inference rules for F_2 sometimes requires several logical steps. For example, the interpretation of (*app*) requires $\forall^1$-elimination as well as *modus ponens*.

3. Rule (*abs*) in its logical interpretation assumes $(\lambda x.M)'x' = M'$ and $(\lambda x.M)''x'' = M''$, hence an underlying first order theory of equality on λ-terms. We can safely limit this theory to a set of equations $M = N$, where M and N are equivalent modulo at most one β-reduction. (Observe, then, how computation is related to proof.)

Proposition 2.4 *The logical rules given are all sound.*

Lemma 2.5 *(Fundamental Theorem of Logical Relations [Sta85, Mit91]) If $\Gamma \rhd E{:}\,\sigma$ is a derivable type judgement, then $\tilde{\Gamma} \vdash \mathcal{R}^\sigma(E', E'')$ is a derivable sequent.*

Proof. By induction on the inference of the type judgement. ∎

Definition 2.6 *When $\vdash \mathcal{R}^\sigma(E, E)$ is a provable sequent, we say the λ-term E is* parametric. *(We remark that in this case, E must be a closed term, hence $E \equiv E' \equiv E''$.)*

Corollary 2.7 *If $\Gamma \rhd E{:}\,\sigma$ is a derivable type judgement, and E is a closed λ-term, then E is parametric.*

There is nothing particularly significant about binary relations: the above syntactic presentation could be carried out for any arity.

Theorem 2.8 *If $\Gamma \vdash \mathcal{R}^\sigma(E', E'')$ is a provable sequent, then there exists a λ-term P and context Γ_P where $\Gamma_P \rhd P{:}\,\sigma$ is a derivable type judgement.*

Proof. (sketch) We take the proof of $\Gamma \vdash \mathcal{R}^\sigma(E', E'')$ and *contract* it by removing all first-order information, in the style of Leivant [Lei90]. For example, the axiom of induction for integers,

$$\mathsf{Int}(k) \equiv \forall P.(\forall x.P(x) \to P(\mathbf{succ}\ x)) \to P(\mathbf{zero}) \to P(k), \tag{10}$$

contracts to $\forall P.(P \to P) \to P \to P$, the inductive type representation of integers. By attaching labels to the hypotheses in sequents, we derive a type inference for a λ-term P describing the structure of the proof of the sequent. ∎

3 Applications of parametricity and induction to proving program equivalence

Parametricity can be used in a straightforward way to prove properties of programs. Here is a simple example:

Proposition 3.1 *If* $\rhd\ f{:}\forall A.A \to A$ *is a derivable* F_2 *type judgement, then* $\vdash \forall x.fx = x$ *is a provable sequent in minimal second order logic.*

Proof. Observe that f need not be $\lambda x.x$, only that it *act like* the identity function. The intuition is clear: for *any* type A, f must input an x of type A, and output fx of type A. Since f has no information as to what operations equip type A, it can perform no useful computation with x, and is constrained to simply output the input.

Since the type judgement $\rhd\ f{:}\forall A.A \to A$ has no assumptions, f is a closed λ-term. By Corollary 2.7, using the *unary* version of parametricity, we then know $\vdash \forall A.\forall x.A(x) \to A(fx)$ is a derivable sequent. By quantifier elimination and introduction, we derive $\vdash \forall A.A(x) \to A(fx)$, the familiar "Leibniz equality" in second order logic. Instantiating A as $A(z) \equiv z = x$, we have $\vdash x = x \to fx = x$. We derive $\vdash \forall x.fx = x$ by use of the identity axiom, modus ponens, and quantifier introduction. ∎

There are many other identities that we can similarly prove, for instance $\rhd\ k{:}\forall AB.A \to B \to A$ implies $\vdash \forall xy.kxy = x$, or $\rhd\ b{:}\forall A.A \to A \to A$ implies $\vdash \forall xy.bxy = x \vee bxy = y$. In these examples, "implies" indicates a metamathematical statement about proofs in the formal systems. Of course, the former example attests to the "uniqueness" of the K combinator, and the latter characterizes the coding of Boolean values. Again, the intuitions are straightforward, but more interesting is that the logic provides a completely mechanical way of deriving the identities.

More problematic, however, is the mechanization of such proofs when the underlying datatypes are infinite and defined inductively: integers, lists, trees, queues, and so on – in short, the building blocks of abstract data types. Parametricity does not seem to be powerful enough for reasoning about these more complex data types. We now attempt to explain why.

3.1 An example: Wadler's "map theorem"

Wadler gives the following example of the use of parametricity: let **m** and **map** both be F_2-typable terms with type $\forall AB.(A \to B) \to A^* \to B^*$, where for any type T we define $T^* \equiv \forall P.(T \to P \to P) \to P \to P$, and let **id** be the usual identity function with polymorphic type $\forall A.A \to A$. Then for any $f \in X \to Y$ and $\ell \in X^*$, we must have

$$\mathbf{map}\ f\ (\mathbf{m}\ \mathbf{id}\ \ell) = \mathbf{m}\ \mathbf{id}\ (\mathbf{map}\ f\ \ell) \tag{11}$$

$$\mathbf{m}\ f\ \ell = \mathbf{map}\ f\ (\mathbf{m}\ \mathbf{id}\ \ell) \tag{12}$$

In trying to prove these equations, we will assume that **m** and **map** satisfy parametricity conditions dictated by the specified type, and the usual equational axioms for **map**. We first try and reproduce Wadler's derivation of the first equation in second-order logic, observing the logical syntax carefully to analyze why the assumptions are insufficient. We then prove the second equation, using logical assumptions which replace parametricity with a higher-order form of induction. In this analysis, types play the role of predicates, and the first-order terms are program fragments. Thus types are propositions about programs. No "intuition" (semantics) is used about what the fragments mean.

We imagine programs to be defined axiomatically via an equational theory, for example

$$\begin{aligned} \mathbf{map}\ f\ \mathbf{nil} &= \mathbf{nil} && (13)\\ \mathbf{map}\ f\ (\mathbf{cons}\ x\ \ell) &= \mathbf{cons}\ (f\ x)\ (\mathbf{map}\ f\ \ell) && (14)\end{aligned}$$

where boldface denotes *constants* and italic face denotes (universally quantified) *variables*.

Let's begin by trying to prove (11), assuming the parametricity conditions $\Psi(\mathbf{map})$ and $\Psi(\mathbf{m})$, where

$$\begin{aligned} \Psi(g) \ \equiv\ & \forall AB.\forall hh'.(\forall xx'.A(x,x') \to B(hx, h'x')) \to \\ & \forall \ell\ell'.\mathcal{R}^{A^*}(\ell,\ell') \to \mathcal{R}^{B^*}(g\ h\ \ell, g\ h'\ \ell'), \end{aligned} \tag{15}$$

and for any binary relation A, we define $\mathcal{R}^{A^*}(\ell,\ell')$ as:

$$\begin{aligned} & \forall P.\forall cc'.(\forall xx'.A(x,x') \to \forall rr'.P(r,r') \to P(c\ x\ r, c'\ x'\ r')) \to \\ & \quad \forall nn'.P(n,n') \to P(\ell\ c\ n, \ell'\ c'\ n'). \end{aligned} \tag{16}$$

(This logical relation comes from the F_2 coding $A^* \equiv \forall P.(A \to P \to P) \to P \to P$ for lists.) Let $A(z,z') \equiv B(z,z') \equiv z' = fz$ and $h = h' = \mathbf{id}$; then from $\Psi(\mathbf{m})$ we derive

$$(\forall xx'.x' = fx \to \mathbf{id}\ x' = f(\mathbf{id}\ x)) \to \forall \ell\ell'.\mathcal{R}^{A^*}(\ell,\ell') \to \mathcal{R}^{B^*}(\mathbf{m}\ \mathbf{id}\ \ell, \mathbf{m}\ \mathbf{id}\ \ell'). \tag{17}$$

By the (added!) assumption $\mathbf{id}\ x = x$, the premise is true; we have

$$\forall \ell\ell'.\mathcal{R}^{A^*}(\ell,\ell') \to \mathcal{R}^{B^*}(\mathbf{m}\ \mathbf{id}\ \ell, \mathbf{m}\ \mathbf{id}\ \ell'). \tag{18}$$

Now let $\ell' = \mathbf{map}\ f\ \ell$: we get

$$\mathcal{R}^{A^*}(\ell, \mathbf{map}\ f\ \ell) \to \mathcal{R}^{B^*}(\mathbf{m}\ \mathbf{id}\ \ell, \mathbf{m}\ \mathbf{id}\ (\mathbf{map}\ f\ \ell)). \tag{19}$$

We can rewrite $\mathcal{R}^{A^*}(\ell, \mathbf{map}\ f\ \ell)$, given our earlier assumption that $A(z,z') \equiv z' = fz$, as

$$\begin{aligned} & \forall P.\forall cc'.(\forall xrr'.P(r,r') \to P(c\ x\ r, c'\ (f\ x)\ r')) \to \\ & \quad \forall nn'.P(n,n') \to P(\ell\ c\ n, (\mathbf{map}\ f\ \ell)\ \ c'\ n'). \end{aligned} \tag{20}$$

Let's ask a simpler question: how could we prove (20) not for *all* ℓ, but for simply $\ell \equiv \mathbf{nil}$? Observe that the first premise is irrelevant; what we really want to prove is (observing $\mathbf{map}\ f\ \mathbf{nil} = \mathbf{nil}$):

$$\forall nn'.P(n,n') \to P(\mathbf{nil}\ c\ n, \mathbf{nil}\ c'\ n'). \tag{21}$$

This is easy as long as we add the assumption

$$\mathbf{nil}\ x\ y \ = \ y. \tag{22}$$

In general, to prove (18) we need an assumption that ℓ is a list of "type" A, i.e., the induction axiom:

$$A^*(\ell) \equiv \forall Q.(\forall yt.A(y) \to Q(t) \to Q(\mathbf{cons}\ y\ t)) \to Q(\mathbf{nil}) \to Q(\ell). \tag{23}$$

We then choose $Q(z) \equiv \mathcal{R}^{A^*}(z, \mathbf{map}\ f\ z)$: note our "simple example" above was proof of the basis $Q(\mathbf{nil})$. For the inductive step, we need to prove $Q(\mathbf{cons}\ y\ t)$ given the additional assumptions

$A(y)$ and $Q(t)$. But $Q(t)$ is

$$\forall P.\forall cc'.(\forall x r r'.P(r,r') \to P(c\ x\ r, c'\ (f\ x)\ r')) \to \\ \forall n n'.P(n,n') \to P(t\ c\ n, (\mathbf{map}\ f\ t)\ c'\ n'), \tag{24}$$

while $Q(\mathbf{cons}\ y\ t)$ is defined as

$$\forall P.\forall cc'.(\forall x r r'.P(r,r') \to P(c\ x\ r, c'\ (f\ x)\ r')) \to \\ \forall n n'.P(n,n') \to P((\mathbf{cons}\ y\ t)\ c\ n, (\mathbf{map}\ f\ (\mathbf{cons}\ y\ t))\ c'\ n'). \tag{25}$$

We now recall that $\mathbf{map}\ f\ (\mathbf{cons}\ y\ t) = \mathbf{cons}\ (f\ y)\ (\mathbf{map}\ f\ t)$; in addition, we add the assumption

$$(\mathbf{cons}\ z\ w)\ x\ y \quad = \quad x\ z\ (w\ x\ y). \tag{26}$$

Observe that using λ-notation, assumptions (22) and (26) can be rewritten as the familiar inductive type definitions

$$\mathbf{nil} \quad \equiv \quad \lambda x.\lambda y.y \tag{27}$$

$$\mathbf{cons} \quad \equiv \quad \lambda z.\lambda w.\lambda x.\lambda y.x\ z\ (w\ x\ y). \tag{28}$$

Adding assumption (26), we observe that the conclusion in (25) can be rewritten as:

$$P(c\ y\ (t\ c\ n), c'\ (f\ y)\ ((\mathbf{map}\ f\ t)\ c'\ n')). \tag{29}$$

Using the premises of (25) in assumption (24), we can derive $P(t\ c\ n, (\mathbf{map}\ f\ t)\ c'\ n')$; we use the assumptions in (25) further to prove (29). From the induction assumption (23) we then derive $\mathcal{R}^{A^*}(\ell, \mathbf{map}\ f\ \ell)$, hence from (19) we get $\mathcal{R}^{B^*}(\mathbf{m}\ \mathbf{id}\ \ell, \mathbf{m}\ \mathbf{id}\ (\mathbf{map}\ f\ \ell))$, namely:

$$\forall P.\forall cc'.(\forall y r r'.P(r,r') \to P(c\ y\ r, c'\ (f\ y)\ r')) \to \\ \forall n n'.P(n,n') \to P(\mathbf{m}\ \mathbf{id}\ \ell\ c\ n, \mathbf{m}\ \mathbf{id}\ (\mathbf{map}\ f\ \ell)\ c'\ n'). \tag{30}$$

If we choose $P(z,z') \equiv z' = \mathbf{map}\ f\ z$ and $c = c' = \mathbf{cons}$, $n = n' = \mathbf{nil}$, then the premises in (30) become

$$r' = \mathbf{map}\ f\ r \to \mathbf{cons}\ (f\ y)\ r' = \mathbf{map}\ f\ (\mathbf{cons}\ y\ r) \tag{31}$$

$$\mathbf{nil} = \mathbf{map}\ f\ \mathbf{nil}, \tag{32}$$

the familiar equational definitions of **map**; we then derive

$$\mathbf{m}\ \mathbf{id}\ (\mathbf{map}\ f\ \ell)\ \mathbf{cons}\ \mathbf{nil} \quad = \quad \mathbf{map}\ f\ (\mathbf{m}\ \mathbf{id}\ \ell\ \mathbf{cons}\ \mathbf{nil}). \tag{33}$$

It then remains to show

$$\mathbf{m}\ \mathbf{id}\ (\mathbf{map}\ f\ \ell)\ \mathbf{cons}\ \mathbf{nil} \quad = \quad \mathbf{m}\ \mathbf{id}\ (\mathbf{map}\ f\ \ell) \tag{34}$$

$$\mathbf{map}\ f\ (\mathbf{m}\ \mathbf{id}\ \ell\ \mathbf{cons}\ \mathbf{nil}) \quad = \quad \mathbf{map}\ f\ (\mathbf{m}\ \mathbf{id}\ \ell) \tag{35}$$

To prove (35) we need to show $\mathbf{m}\ \mathbf{id}\ \ell\ \mathbf{cons}\ \mathbf{nil} = \mathbf{m}\ \mathbf{id}\ \ell$, which we demonstrate by induction on $\mathbf{m}\ \mathbf{id}\ \ell$—but to do so, we need to know that it is indeed a list! Hence we must make the additional assumption that $\mathbf{m}$ is totally defined and maps lists to lists, i.e., $\mathcal{M}(\mathbf{m})$, where

$$\mathcal{M}(g) \equiv \forall AB.\forall f.(\forall x.A(x) \to B(fx)) \to \forall \ell.A^*(\ell) \to B^*(g\ f\ \ell). \tag{36}$$

(Recall (23) for the definition of A^* and B^*.) In the course of the induction proof of (35), the "iterator equalities" (22) and (26) are used again. The proof of (34) is similar.

Retrospective

What added assumptions become apparent in this proof? Recall the slogan that motivated Wadler's theorems: "You give me the type, and I'll give you a theorem." From the *type* of $\mathbf{m}$ came the "free" proof of the *parametricity* of $\mathbf{m}$, but we needed to assume its *totality* as well, namely that $\mathbf{m}$ mapped lists of one "type" to lists of another "type." Induction axioms for lists were also essential; again, assumptions that lists are parametric did not seem strong enough. Also, we required the iterator equalities for **cons** and **nil**.

The induction hypotheses used by Wadler (essentially, the instantiations of A and B) were entirely correct; what we have tried to do is formalize some of the inductive machinery. What we discover is that from assuming $\mathbf{m}$ (and, of course, **map**) is *total* and mapping lists to lists, the assumptions of parametricity and, as we shall see shortly, the iterator equalities are not needed either. The inductions needed to finish a parametricity-based proof, however superfluous they seem to be, depend on a proof that $\mathbf{m}$ is a total function mapping lists to lists. This condition on $\mathbf{m}$ seems stronger than its mere parametricity. The reason is in part given by the following proposition:

Proposition 3.2 $\forall xy.\mathbf{nil}\ x\ y = x, \forall xyzw.(\mathbf{cons}\ z\ w)\ x\ y = x\ z\ (w\ x\ y) \vdash A^*(\ell) \to \mathcal{R}^{A^*}(\ell)$.

Proof. We instantiate the induction axiom $A^*(\ell)$ with a parametricity relation. ∎

We can generalize the proposition to:

Theorem 3.3 *If the constructors of an inductive data type τ satisfy equations of primitive recursion, then the terms of type τ are parametric.*

(The converse does not seems obvious.) Furthermore, the truth of the program equivalences described above depend on the style in which lists are typically encoded in F_2 (see Section 4): the "map theorem" is then dependent on a particular coding.

3.2 Another interesting equality, using a simpler proof method

We now proceed to prove Wadler's second equation. In place of parametricity, the logical assumption made about $\mathbf{m}$ we write as $\mathcal{M}_2(\mathbf{m})$, where $\mathcal{M}_2(g)$ is defined as:

$$\begin{aligned}\mathcal{M}_2(g) \ \equiv\ & \forall AB.\forall hh'.(\forall xx'.A(x,x') \to B(h\ x, h'\ x')) \to \\ & \forall \ell\ell'.\mathcal{S}^{A^*}(\ell,\ell') \to \mathcal{S}^{B^*}(g\ h\ \ell, g\ h'\ \ell'). \qquad (37)\end{aligned}$$

This proposition introduces relations $\mathcal{S}^{A^*}$ and $\mathcal{S}^{B^*}$, where for any binary relation A we define $\mathcal{S}^{A^*}$ as:

$$\begin{aligned}\mathcal{S}^{A^*}(\ell,\ell') \ \equiv\ & \forall P.(\forall yy'tt'.A(y,y') \to P(t,t') \to P(\mathbf{cons}\ y\ t, \mathbf{cons}\ y'\ t')) \to \\ & P(\mathbf{nil},\mathbf{nil}) \to P(\ell,\ell'). \qquad (38)\end{aligned}$$

Hence if $\mathcal{S}^{A^*}(\ell,\ell')$ is provable, ℓ and ℓ' must have the same length, where the elements of the two lists are pairwise related by relation A. It is this logical statement which corresponds to Wadler's informal definition (in [Wad89]) of relations between lists; this informal definition is not precisely captured by the logical relation defined by the type of lists.

Observe that if $\mathcal{M}(g)$ is provable in second order logic, so is $\mathcal{M}_2(g)$. To construct $\mathcal{M}_2(g)$ from $\mathcal{M}(g)$, we replace unary relations by binary relations, and replace each first-order quantified variable by two such variables: the first variable is used in the first coordinate of binary relations, and the second variable in the second coordinate of the relations. The (metamathematical) transformation of the associated proof of $\mathcal{M}(g)$ into a proof of $\mathcal{M}_2(g)$ is similar. Just as parametricity is a general notion which can be expressed over relations of any arity, so we may generalize inductively defined relations to any arity. Since inductively defined relations are fundamentally induction axioms in the logic, programs like $\mathbf{m}$ and $\mathbf{map}$ induce *higher order* inductively defined relations (they map inductions axioms to induction axioms).

Given $\mathcal{M}_2(\mathbf{m})$, let $A(z, z') \equiv z' = z$, $B(z, z') \equiv z' = fz$, $h = \mathbf{id}, h' = f$, and let $\ell' = \ell$; we derive

$$(\forall x x'. x = x' \to f x' = f(\mathbf{id}\ x)) \to \mathcal{S}^{A^*}(\ell, \ell) \to \mathcal{S}^{B^*}(\mathbf{m}\ \mathbf{id}\ \ell, \mathbf{m}\ f\ \ell). \tag{39}$$

The first premise is a tautology; we prove $\mathcal{S}^{A^*}(\ell, \ell)$ by induction on ℓ, recalling the induction axiom

$$A^*(\ell) \equiv \forall Q.(\forall y t. A(y) \to Q(t) \to Q(\mathbf{cons}\ y\ t)) \to Q(\mathbf{nil}) \to Q(\ell), \tag{40}$$

taking $Q(z) \equiv \mathcal{S}^{A^*}(z, z)$. We omit this proof, which is quite straightforward: observe that it does not use iterator equations. We then derive $\mathcal{S}^{B^*}(\mathbf{m}\ \mathbf{id}\ \ell, \mathbf{m}\ f\ \ell)$, namely:

$$\forall P.(\forall y t t'. P(t, t') \to P(\mathbf{cons}\ y\ t, \mathbf{cons}\ (f\ y)\ t')) \to P(\mathbf{nil}, \mathbf{nil}) \to P(\mathbf{m}\ \mathbf{id}\ \ell, \mathbf{m}\ f\ \ell). \tag{41}$$

Take $P(z, z') \equiv z' = \mathbf{map}\ f\ z$; we get

$$\begin{aligned} &(\forall y t. \mathbf{cons}\ (f\ y)\ (\mathbf{map}\ f\ t) = \mathbf{map}\ f\ (\mathbf{cons}\ y\ t)) \to \\ &\qquad \mathbf{nil} = \mathbf{map}\ f\ \mathbf{nil} \to \mathbf{m}\ f\ \ell = \mathbf{map}\ f\ (\mathbf{m}\ \mathbf{id}\ \ell). \end{aligned} \tag{42}$$

We then prove $\mathbf{m}\ f\ \ell = \mathbf{map}\ f\ (\mathbf{m}\ \mathbf{id}\ \ell)$ using the equational definition of $\mathbf{map}$ and *modus ponens*.

3.3 Comparison of the two proof methods

In what sense can our proof methods be considered simpler? The key insight of Wadler was that the parametricity of $\mathbf{m}$ is *almost* a higher-order form of induction, and the second order instantiations in his proofs are essentially induction hypotheses. It may seem that the substitutions used in both proofs are without motivation, but one should remember that induction indeed requires insight. However, the substitutions we have used can be derived almost mechanically by unification of parts of the "goal" equation with the terms of the final atomic formula in $\mathcal{M}_2(\mathbf{m})$ (respectively, $\Psi(\mathbf{m})$).

We observe that Wadler leaves out some of the logical machinery needed to push through these inductive assertions, and this machinery seems to be the difference between parametric and inductive assumptions. While these proofs can be carried out by assuming that *lists* are parametric when viewed as primitive recursive *iterators*, an additional assumption is then needed at the end that lists are *inductive*–not at all surprising, since lists are a standard example of inductive data types. Our simplification is to largely dispense with parametricity, and work directly with induction axioms.

However, it should be remarked that parametricity defines a logical relation which can be generated from *any* type inference for a closed term, while our "inductively defined relations" seem to make sense for only a restricted subset of terms (those inductively defined). Furthermore, the higher order inductively defined relations (like $\mathcal{M}_2(\mathbf{m})$) require some ingenuity to prove, and do not follow mechanically from type inferences. In this sense, parametricity seems to be a more general notion, but not necessarily more powerful. There are indeed bizarre terms which are parametric, for example, the simulation of Turing Machines in [HM91], employing a programming style completely orthogonal to the inductive-type style illustrated, for example, in [PDM89]; nonetheless, it seems unclear what program equivalences can be proven from them using parametricity. In the case of programming over inductively defined types, we have shown that logical machinery based on inductively defined relations is a much straighter path to the goal.

3.4 Proof theory and cut elimination

There is an added benefit from the approach we have advocated: the "map theorems" we have proven are not restricted to the second-order polymorphic lambda calculus. More specifically, our derived equalities hold for any programming language, including ones having a fixpoint operator, where we can prove $\mathcal{M}(\mathbf{m})$, $\mathcal{M}(\mathbf{map})$, and $A^*(\ell)$. Without loss of generality, we can let the first-order terms be over an algebra with variables, constants (**cons**, **nil**, **m**, **map**, etc.), and a first order theory which is not about β-reduction, but *any* equational theory. For instance, we can describe **map** by the standard equational presentation, with similar definitions for **m**. As long as we can prove the relevant higher order induction propositions about the relevant programs, the result holds. Equivalents of the "map theorem" can easily be stated for other inductive datatypes, where (for example) we replace lists with trees: what changes is the form of the induction axioms.

What, then, does any of this have to do with F_2, the second-order polymorphic lambda calculus? Leivant has shown that by analyzing the *proofs* of $\mathcal{M}(\mathbf{m})$ and $\mathcal{M}(\mathbf{map})$, we can extract F_2-typable *algorithms* for computing these functions, where the "map theorem" holds for these algorithms. In the case of **map**, the algorithm we synthesize is:

$$\mathbf{map} = \lambda f.\lambda\ell.\ell\ (\lambda x.\mathbf{cons}\ (f\ x))\ \mathbf{nil}. \tag{43}$$

Written with F_2 type information, this can be rewritten:

$$\begin{aligned}\mathbf{map} \;=\; & \Lambda A.\Lambda B.\lambda f\colon A\to B.\\ & \quad \lambda\ell\colon\forall P.(A\to P\to P)\to P\to P.\\ & \qquad \ell\ [\forall Q.(B\to Q\to Q)\to Q\to Q]\\ & \qquad\quad (\lambda x\colon A.\mathbf{cons}[B]\ (f\ x))\ \mathbf{nil}[B],\end{aligned} \tag{44}$$

where

$$\begin{aligned}\mathbf{cons} \;=\; & \Lambda A.\lambda x\colon A.\lambda\ell\colon\forall P.(A\to P\to P)\to P\to P.\\ & \quad \Lambda Q.\lambda c\colon A\to Q\to Q.\lambda n\colon Q.\\ & \qquad c\ x\ (\ell[Q]\ c\ n)\end{aligned} \tag{45}$$

$$\mathbf{nil} \;=\; \Lambda A.\Lambda Q.\lambda c\colon A\to Q\to Q.\lambda n\colon Q.n. \tag{46}$$

Written in "logical" form, the totality of the **map** function could be written as:

$$\begin{aligned}\mathbf{map} \;=\; & \Lambda A.\Lambda B.\lambda f\colon\forall x.A(x)\to B(fx).\\ & \quad \lambda\ell\colon\forall P.(\forall yt.A(y)\to P(t)\to P(\mathbf{cons}\ x\ t))\to P(\mathbf{nil})\to P(\ell).\\ & \qquad \ell\ [\forall Q.(\forall yt.B(y)\to Q(t)\to Q(\mathbf{cons}\ x\ t)\to Q(\mathbf{nil})\to Q(\ell)]\\ & \qquad\quad (\lambda x\colon A(x).\mathbf{cons}\ (f\ x))\ \mathbf{nil}\colon Q(\mathbf{nil}).\end{aligned} \tag{47}$$

While the "logical" notation above is a bit fanciful, it is meant to suggest the Curry-Howard isomorphism: namely, that λ-bound variables are the names of *parcels* of logical hypotheses, where function application is deduction.

Finally, we mention that our proofs of the "map theorems" have an additional computational content: not only do they prove equalities of certain terms, but they ensure that these terms are convertible, and moreover encode the conversion. Syntactic equalities ensure truth in all models, hence in the term model, but we can in addition "provably" convert both sides of the equalities (for fixed f and ℓ, with proper substitutions for the second-order variables) by performing cut elimination on the resultant proof.

3.5 An analysis of Reynolds' isomorphism

Using parametricity, we can as well give a syntactic explanation of an isomorphism discussed by Reynolds in [Rey83] and later by Wadler. The syntactic analysis is interesting because it makes explicit an encoding of existential quantification. Let $\mathbf{i} \equiv \lambda x.\lambda g.gx$ and $\mathbf{j} \equiv \lambda h.h(\lambda x.x)$, with F_2 typings $\mathbf{i}: A \rightarrow \tilde{A}$ and $\mathbf{j}: \tilde{A} \rightarrow A$, where $\tilde{A} \equiv \forall X.(A \rightarrow X) \rightarrow X$. By pure equational reasoning (i.e., using β-reduction only) one can easily prove $\forall x.\mathbf{j}(\mathbf{i}\,x) = x$, but not $\forall h.\mathbf{i}(\mathbf{j}\,h) = h$: by equational reasoning, the best one can do is $\mathbf{i}(\mathbf{j}\,h) = \lambda g.g(h(\lambda x.x))$. However, if $\mathbf{i}(\mathbf{j}\,h)$ is F_2-typable, then $h: \tilde{A}$, so that we can indeed prove $\forall h.\mathcal{R}^{\tilde{A}}(h) \rightarrow \mathbf{i}(\mathbf{j}\,h) = h$ in second order logic. Recall that the *unary* relation $\mathcal{R}^{\tilde{A}}(h)$ is defined as

$$\mathcal{R}^{\tilde{A}}(h) \equiv \forall X.\forall g.(\forall a.A(a) \rightarrow X(ga)) \rightarrow X(hg); \tag{48}$$

instantiating X as $X(z) \equiv \exists b.z = gb \wedge A(b)$, from a proof of $\mathcal{R}^{\tilde{A}}(h)$ one can then derive

$$(\forall a.A(a) \rightarrow \exists b.ga = gb \wedge A(b)) \rightarrow \exists b.hg = gb \wedge A(b). \tag{49}$$

The premise is clearly provable, and from $hg = gb$ we can deduce $\lambda g.hg = \lambda g.gb$; assuming a λ-theory with η-equality, we then have $\exists b.h = \lambda g.gb \wedge A(b)$. Note that while our goal is to prove an equality *for all* h, there is really only one *interesting* h (modulo the value b). By equational reasoning we then can prove

$$\exists b.\mathbf{i}(\mathbf{j}\,h) = \mathbf{i}\,b = \lambda g.gb = h \wedge A(b). \tag{50}$$

The fact that we instantiate X using existential quantifiers should come as no surprise, since in second order logic we can prove

$$\vdash \exists y.A(y) \longleftrightarrow \forall X.(\forall y.A(y) \rightarrow X) \rightarrow X; \tag{51}$$

see ([vanD79], pp. 160–162). Ignoring first-order information, we have the propositional theorem $\vdash A \leftrightarrow \forall X.(A \rightarrow X) \rightarrow X$. The functions $\mathbf{i}$ and $\mathbf{j}$ code, in the style of the Curry-Howard correspondence, an introduction and elimination of existential quantifiers: viewed from a propositional perspective; they code the "iff" of the above propositional tautology.

4 Programs for free!

Since the subject of this paper has been parametricity, induction, and the synthesis of F_2-typable terms from parametricity and induction proofs, we provide a brief compendium of the functional programmer's favorite functions, written in a form which admits F_2 typing. The method is not new (see, for instance, [PDM89]), but *how* these functions work is easier to see for the first time if we ignore typing information. We call these "free programs" because they do not use recursion explicitly, but rather a form of *primitive recursion*, although (as in Wadler's analysis) they *do* hide a form of (structural) induction. The examples assume the usual λ-calculus coding of Boolean values, pairing, and projection.

4.1 Integers

Integers are represented by the familiar Church numerals, where the integer k is given by $\lambda s.\lambda z.s^k z$.

$$\begin{aligned}
\textbf{zero} &= \lambda s.\lambda z.z && (52)\\
\textbf{succ} &= \lambda n.\lambda s.\lambda z.s\ (n\ s\ z) && (53)\\
\textbf{plus} &= \lambda m.\lambda n.m\ \textbf{succ}\ n && (54)\\
\textbf{times} &= \lambda m.\lambda n.m\ (\textbf{plus}\ n)\ \textbf{zero} && (55)\\
\textbf{expt} &= \lambda b.\lambda e.e\ (\textbf{times}\ b)\ (\textbf{succ zero}) && (56)\\
\textbf{pred} &= \lambda n.\pi_2\ (n\ (\lambda p.\langle \textbf{succ}\ (\pi_1\ p), \pi_1\ p\rangle)\ \langle \textbf{zero}, \textbf{zero}\rangle) && (57)\\
\textbf{minus} &= \lambda m.\lambda n.n\ \textbf{pred}\ m && (58)\\
\textbf{zero?} &= \lambda n.n(\lambda x.\textbf{false})\ \textbf{true} && (59)\\
\textbf{equal?} &= \lambda m.\lambda n.\textbf{and}\ (\textbf{zero?}\ (\textbf{minus}\ m\ n))\ (\textbf{zero?}\ (\textbf{minus}\ n\ m)) && (60)
\end{aligned}$$

4.2 Lists

Church numerals are very familiar, but note that the following presentation of lists can be thought of as "Church lists," where lists are represented as:

$$[x_1, x_2, \ldots, x_k] \equiv \lambda c.\lambda n.c\ x_1\ (c\ x_2\ \cdots\ (c\ x_k\ n)\cdots).$$

Observe how **nil** is like **zero**, **cons** is like **succ**, **tail** is like **pred**, and **append** is like **plus**. Church numerals are just lists where the cells in the list cannot contain any information. Observe as well how lists abstract control structure.

$$\begin{aligned}
\textbf{nil} &= \lambda c.\lambda n.n && (61)\\
\textbf{cons} &= \lambda x.\lambda \ell.\lambda c.\lambda n.c\ x\ (\ell\ c\ n) && (62)\\
\textbf{head} &= \lambda \ell.\ell\ (\lambda x.\lambda y.x)\ \textbf{nil} && (63)\\
\textbf{tail} &= \lambda \ell.\pi_2\ (\ell\ (\lambda x.\lambda p.\langle \textbf{cons}\ x\ (\pi_1\ p), \pi_1\ p\rangle)\ \langle \textbf{nil}, \textbf{nil}\rangle) && (64)\\
\textbf{append} &= \lambda \ell_1.\lambda \ell_2.\ell_1\ \textbf{cons}\ \ell_2 && (65)\\
\textbf{append-lists} &= \lambda L.L\ \textbf{append nil} && (66)\\
\textbf{map} &= \lambda f.\lambda \ell.\ell\ (\lambda x.\textbf{cons}\ (f\ x))\ \textbf{nil} && (67)\\
\textbf{length} &= \lambda \ell.\ell\ (\lambda x.\textbf{succ})\ \textbf{zero} && (68)\\
\textbf{tack} &= \lambda x.\lambda \ell.\ell\ \textbf{cons}\ (\textbf{cons}\ x\ \textbf{nil}) && (69)\\
\textbf{reverse} &= \lambda \ell.\ell\ \textbf{tack nil} && (70)\\
\textbf{filter} &= \lambda \ell.\lambda \mathit{test}.\ell\ (\lambda x.(\mathit{test}\ x)\ (\textbf{cons}\ x)\ (\lambda y.y))\ \textbf{nil} && (71)
\end{aligned}$$

The interested reader should try to code insertion of an integer into a sorted list of integers: insertion sort can then be coded as $\textbf{sort} = \lambda \ell.\ell\ \textbf{insert nil}$.

4.3 Binary trees

$$\begin{aligned}
\textbf{leaf} &= \lambda x.\lambda n.\lambda \ell.\ell\ x && (72)\\
\textbf{node} &= \lambda L.\lambda R.\lambda n.\lambda \ell.n\ (L\ n\ \ell)\ (R\ n\ \ell) && (73)\\
\textbf{flatten} &= \lambda t.t\ \textbf{append}\ (\lambda x.\textbf{cons}\ x\ \textbf{nil}) && (74)\\
\textbf{sum-leaves} &= \lambda t.t\ \textbf{plus}\ (\lambda x.x) && (75)\\
\textbf{map-leaves} &= \lambda t.\lambda f.t\ \textbf{node}\ (\lambda x.\textbf{leaf}\ (f\ x)) && (76)
\end{aligned}$$

5 Conclusions

Polymorphic types give a great deal of information about λ-terms. They guarantee strong normalization, and can also prove termination in a stricter sense, i.e., ensuring particular inductively defined relations between (inductive type) inputs and outputs. Polymorphic types also encode parametricity. Given such properties as a consequence of having a polymorphic type, it should come as no surprise that type inference is so computationally difficult!

While it might have seemed that parametricity allows us to 'read off' unusual program equivalences, we have shown that proofs of these equivalences still seem to require structural induction, as well as stronger assumptions than parametricity. What Wadler's analysis does suggest, however, is a kind of structural induction argument over arbitrary relations, as in axiom $\mathcal{S}^{A^*}(\ell, \ell')$ for lists. Theorem provers should consequently rest assured that theorems are not free: proving properties of programs requires induction. The use of "unary" parametric assumptions, on the other hand, seems appropriate for proving that there is "only one" λ-term of a given type.

We have put the case quite strongly for looking to syntactic methods for understanding parametricity and proving program equivalences, but really this is just a matter of taste. What this perspective forces on the analyst is to make (near all) matters of proof explicit and constructive. By way of analogy, consider Gödel's theorem that the consistency of the Peano axioms cannot be demonstrated by constructive, finitary means—yet there is a naive proof of exactly that consistency by considering a plausible model (the integers as we know them), and reflection that the Peano axioms are indeed true in the model, and hence are consistent. While this example is far more sophisticated than anything presented in this paper, there is a powerful analogy in that Wadler's analysis, by focusing on model theory, neglected some of the hard work (or boring drudgery?) in formally carrying out what are essentially induction hypotheses.

Several questions merit further attention. We have seen that closed, typable λ-terms are parametric; is the converse true? Are parametric data types inductive? Can the syntactic presentation of parametricity shown in Section 2 be used to give a term model for F_2 that is not incestuous, but of genuine interest? Does the perspective on data types presented here give any further insight into the complexity of type inference? These will be topics of future research.

Acknowledgements. I would like to thank Daniel Leivant for his patience and kindness far beyond the call of duty, in his answering a seemingly interminable barrage of questions about his paper "Contracting Proofs to Programs," as well as providing further encouragement and advice on the current work. Phil Wadler has also been extremely generous with his time and energy, explaining a lot to me that I didn't understand. I also want to thank Luca Cardelli, Qingming Ma, John Reynolds, Don Smith, and Mitch Wand for their encouragement of this research, and comments on earlier versions of the paper.

References

[BW88] R. Bird and P. Wadler. **Introduction to Functional Programming.** Prentice-Hall, 1988.

[BB85] C. Böhm and A. Berarducci. *Automatic synthesis of typed λ-programs on term algebras.* **Theoretical Computer Science** 39, pp. 135–154.

[BMM90] K. Bruce, J. C. Mitchell, and A. R. Meyer. *The semantics of second-order lambda-calculus.* In **Logical Foundations of Functional Programming**, ed. G. Huet, pp. 213–272. Addison Wesley, 1990.

[vanD79] D. van Daalen. **Logic and Structure.** Springer-Verlag, 1979.

[Gir72] J.-Y. Girard, *Interprétation Fonctionelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur.* Thèse de Doctorat d'Etat, Université de Paris VII, 1972.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. **Proofs and Types.** Cambridge University Press, 1989.

[Göd58] K. Gödel. *Über eine bisher noch nict benüte Erweiterung des finiten Standpunktes.* **Dialectica** 12 (1958), pp. 280–287. Republished with English translation and explanatory notes by A. S. Troelstra in **Kurt Gödel: Collected Works** (Oxford University Press, 1990), vol. II, ed. S. Feferman.

[How80] W. Howard. *The formulae-as-types notion of construction.* In **To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism**, ed. J. Seldin and R. Hindley, pp. 479–490. Academic Press, 1980.

[Lei90] D. Leivant. *Contracting proofs to programs.* In **Logic and Computer Science**, ed. P. Odifreddi, pp. 279–328. Academic Press, 1990.

[HM91] F. Henglein and H. Mairson. *The complexity of type inference for higher-order typed lambda calculi.* **Proceedings of the 18-th ACM Symposium on the Principles of Programming Languages**, January 1991, pp. 119–130.

[Mit91] J. C. Mitchell. *Type systems for programming languages.* **Handbook of Theoretical Computer Science**, van Leeuwen et al., eds. North-Holland, 1990, pp. 365–458.

[PDM89] B. Pierce, S. Dietzen, and S. Michaylov. *Programming in higher-order typed lambda calculi.* Technical Report CMU-CS-89-111, Carnegie Mellon University, March 1989.

[Rey74] J. C. Reynolds. *Towards a theory of type structure.* In **Proceedings of the Paris Colloquium on Programming**, Lecture Notes in Computer Science 19, Springer Verlag, pages 408–425, 1974.

[Rey90] J. C. Reynolds. *Introduction to polymorphic lambda-calculus.* In **Logical Foundations of Functional Programming**, ed. G. Huet, pp. 77–86. Addison Wesley, 1990.

[Rey83] J. C. Reynolds. *Types, abstraction, and parametric polymorphism.* In **Information Processing 83**, ed. R. E. A. Mason, pp. 513–523. Elsevier, 1983.

[Sc90] A. Scedrov. *A guide to polymorphic types.* In **Logic and Computer Science**, ed. P. Odifreddi, pp. 387–420. Academic Press, 1990.

[Sta85] R. Statman. *Logical relations and the typed lambda calculus.* **Information and Control** 65 (1985), pp. 85–97.

[Wad89] P. Wadler. *Theorems for free!* In **4th International Symposium on Functional Programming Languages and Computer Architecture**, London, September 1989.