Logic Column 6

Column Editor: Jon G. Riecke
Bell Laboratories, Lucent Technologies
700 Mountain Avenue
Murray Hill, NJ 07974
riecke@bell-labs.com

Editor's note: This month's column, by Peter W. O'Hearn, describes the semantics of data abstraction and its importance in accounting for local state.

Polymorphism, Objects and Abstract Types*

Peter W. O'Hearn
Department of Computer Science
Queen Mary & Westfield College
(ohearn@dcs.qmw.ac.uk)

October 7, 1998

1 Introduction

Abstraction is one of the pillars of computer science. Of course, choosing the right concepts to emphasize, and details to suppress, is crucial in all of science, but in computer science the role of abstraction extends beyond that of sensible methodology. For in computing we have seen the emergence of an array of constructs and methods whose chief purpose is to provide *general mechanisms* for achieving, validating, or enforcing abstraction, rather than being *specific instances* or examples of it. Among these are programming concepts such as objects, procedures, abstract data types and modules, and mathematical methods such as simulation and logical relations.

But what, more precisely, is the "abstraction" achieved by these constructs? In the case of procedural abstraction, a more or less satisfactory explanation can be given in terms of functions. Much more subtle, and novel, is the idea of data abstraction: A collection of programs or operations conspiring together to represent higher level pieces of data.

What kind of mathematical entities are data abstractions? One answer is provided by algebra, and there has been a good deal of development of the theory of algebraic specifications. This theory is important, but it does not provide a comprehensive answer. In particular, it does not cope comfortably with imperative or object-oriented features, with the ability to generate *new* abstractions (as in new objects), and it is limited to first-order functions.

The purpose of this article is to trace a line of development which revolves around a core calculus, the polymorphic λ-calculus of Girard [6] and Reynolds [31]. Just as the λ-calculus is the

quintessential calculus of functions, the polymorphic calculus is a basic calculus in which a wide range of data abstractions can be expressed.

That is not to say that the polymorphic calculus provides the final word on data abstraction. In particular, the extension of polymorphism to dependent types allows for the description of flexible and subtle forms of module [15, 18]. But type dependency and modules fall outside the scope of this article.

The line we trace begins with work on representing abstract data types in the polymorphic calculus, and a semantic theory, relational parametricity, which describes the sense of "abstractness" in the calculus. We then move on to consider the use of polymorphism to account for the hiding aspect of the object form of data abstraction.

But before proceeding with these accounts we will take a closer look at the two forms of data abstraction that concern us.

2 Objects versus Abstract Data Types

An abstract data type (ADT) is defined by giving a representation type, together with operations for acting on it. The representation type is hidden, in that the operations provide the only means of accessing it. With an object, operations are given for manipulating hidden local state, which again may only be accessed through the provided operations.

But though ADT's and objects both provide data abstraction, they are not the same. That this is so can be seen immediately if we consider that the ADT form *requires* types, while the object form does not. For instance, in Scheme we can construct a counter object whose local state is hidden, which as a result can be incremented but never decremented.

```
(define counter
   (let ((state 0))
     (lambda (message)
        (cond ((eq? message 'inc) (set! state (+ state 1)))
              ((eq? message 'val) state)
         ))))
```

Many object-oriented languages include a class concept which involves some aspects of ADT's and, equally, in typed languages such as ML there are often constructs which allow objects of this form to be constructed. But the point, as illustrated by Scheme and pure object-oriented languages such as Actor-based languages, is that types are not *necessary* to the object form of data abstraction. We don't consider absence of types as crucial to investigate this point further, but rather appeal to it as a simple illustration of the different ways that ADT's and objects enforce abstraction. (A more penetrating account of the implications of the differences may be found in [32, 4].)

The reader may have noticed that we speak of the "object form" of data abstraction, rather than simply objects; this is to separate the way of providing data abstraction from other aspects of objects such as inheritance, which we do not consider. We are not claiming that procedures plus local state capture the full essence of the object concept, only that they give rise to an instance of the object form of data abstraction, based on operations for acting on hidden local state.

Although ADT's and objects are not the same, there is an undeniable similarity between them, and we should ask whether the way that each provides abstraction is closely related to the other.

Put another way, we may ask if the *hiding aspect* of the ADT and object forms of data abstraction are, in essence, the same.

We will return to this question, after describing the view of abstract types given by the polymorphic λ-calculus.

3 Polymorphism and Abstract Types

The polymorphic λ-calculus extends the simply-typed λ-calculus with types $\forall\alpha\,.\,T(\alpha)$ for polymorphic functions. Officially, the types are given by the grammar

$$A \;::=\; \alpha \mid A \to A \mid \forall\alpha\,.\,A$$

where α ranges over an infinite set of type variables. A polymorphic function p of type $\forall\alpha\,.\,T[\alpha]$ can be supplied with a type argument, in which case the polymorphic instantiation $p[A]$ has type $T[A]$.

A new binder Λ binds type variables in functions, so that, for example,

$$\Lambda\alpha\,\lambda f : \alpha \to \alpha\,.\,\lambda x : \alpha\,.\,f\,f\,x$$

is the polymorphic doubling function of type $\forall\alpha\,.\,(\alpha \to \alpha) \to (\alpha \to \alpha)$.

3.1 Encodings of ADT's

Consider the declaration of an abstract type α with operations x_i of type $t_i(\alpha)$, representation type T, and operations K_i.

abstype α **with** $x_1 : t_1(\alpha), \ldots, x_n : t_n(\alpha)$
is $T \;\; K_1 \ldots K_n$
in M

Reynolds proposed [31] that such a declaration could be regarded as an abbreviation for a polymorphic λ-calculus expression

$$(\Lambda\alpha\,.\,\lambda x_1 \ldots \lambda x_n\,.M)\,[T]\,K_1 \ldots K_n$$

which binds α and x_i to their concrete representations.

The intuition behind this abbreviation is not just syntactic, i.e., how one might evaluate or typecheck an abstype definition. Rather, it is based on the view that the abstract, or hiding, aspect of the ADT corresponds to *parametricity*, a uniformity property of polymorphic functions. Roughly, parametricity means that a polymorphic function does not "look under" a type variable which may be instantiated in a number of different ways. A related statement is that parametric functions "work the same way" at all types. (See [25, 5] for further discussions of this and related notions of parametricity.)

Thus, parametricity requires much more than mere type correctness of polymorphic functions. For instance, we could conceive of a polymorphic function p of type $\forall\alpha\,.\,\alpha \to \alpha \to \alpha$ that returns the first projection $p[int] = \lambda xy\,.\,x$ when instantiated to the set of integers, but that returns the second projection $p[D] = \lambda xy\,.\,y$ for all other types D. This function, while being type correct in an obvious sense, is not parametric, because it works differently for integers than for other types.

In an influential article "Abstract types have existential type" Mitchell and Plotkin proposed a different encoding of ADT's [19]. A value of type $\exists\alpha\,.\,T(\alpha)$ is given by a package, consisting of

a pair $[D, m]$ where D is a type and $m \in T(D)$. The type D in this pair is not supposed to be exposed, however; it is "hidden" by $\exists$. In Mitchell and Plotkin's view an ADT with signature and representation as above determines a package

$$[T, K_1, \ldots, K_n] : \exists\alpha \,.\, t_1(\alpha) \times \cdots \times t_n(\alpha)$$

With this representation we can unpack a value $v \in \exists\alpha \,.\, t_1(\alpha) \times \cdots \times t_n(\alpha)$ as

$$\textit{unpack } v \textit{ as } [\alpha \,,\, x_1, \ldots, x_n] \textbf{ in } M$$

which is equivalent to Reynolds's representation of an abstype declaration, except that it has the additional proviso that α cannot appear freely in the type of M. Further, since an abstract type in Mitchell and Plotkin's representation is a value, it can be passed around, and even used in the branches of a conditional, like any other value.

We could explicitly add types of the form $\exists\alpha \,.\, A$ to the polymorphic calculus, but these types can be encoded in terms of $\forall$ and $\rightarrow$ as follows:

$$\exists\alpha \,.\, T(\alpha) \;\equiv\; \forall\beta \,.\, (\forall\alpha \,.\, T(\alpha) \rightarrow \beta) \rightarrow \beta$$

(We can also encode products, which we used implicitly above in the type $\exists\alpha \,.\, t_1(\alpha) \times \cdots \times t_n(\alpha)$.) For this to work correctly, however, properties beyond bare β and η equality are needed; these are provided by relational parametricity.

3.2 Relational Parametricity

Throughout his work Reynolds has emphasized a connection between parametric polymorphism and *representation independence*, the principle that behaviour is invariant under changes to the concrete representations of types. For example, a client that uses a type of stacks should not be able to distinguish (at a suitable level of abstraction) an implementation based on lists from one based on functions with integer domain.

The basic idea behind relational parametricity is simple. Suppose we have a polymorphic function $p : \forall\alpha \,.\, T(\alpha)$. This function can be instantiated to a variety of types, yielding $p[D] : T(D)$, $p[E] : T(E)$... Relational parametricity says that the different instantiations have the following relationship, which we call the (binary, relational) parametricity condition:

> for any types D and E and any relation $R : D \leftrightarrow E$, there is an induced relation $T(R) : T(D) \leftrightarrow T(E)$, and $(p[D], p[E]) \in T(R)$.

We may regard a relation $R : D \leftrightarrow E$ as relating different representations of α, and $T(R)$ as an invariant relationship that must be maintained. Typically, the relation $T(R)$ is determined in an inductive manner (of logical relations [26]), with the significant caveat that free type variables other than α are mapped to identity relations. The idea is that two pieces of code satisfying invariant $T(R)$ should behave equivalently from the point of view of the "visible" types, types other than α.

Relational parametricity gives rise to a proof principle for abstract type declarations. In terms of Reynolds's encoding, parametricity of $\Lambda\alpha \,.\, \lambda x_i \,.\, M$ means that all relations are preserved. Given two concrete representations $[T]\, K_1 \ldots K_n$ and $[T']\, K'_1 \ldots K'_n$ if we can find a relation $R : T \leftrightarrow T'$ under which each pair K_i, K'_i is invariant (according to the induced relation $t_i(R) : t_i(T) \leftrightarrow t_i(T')$), then the entire definition will respect R. In some cases, such as when α is not free in the type of M, this will imply that the two declarations are equal. In terms of Mitchell and Plotkin's encoding, such a relation shows the equality of elements of existential type [25]. For instance, we can implement stacks of integers using $list[int]$ as the representation type, or a type $(int \rightarrow int) \times int$ where the

int indicates the top of the stack. The relation used to prove equivalence of the representations relates a list to a pair (f, n) such that $f(0), ...f(n)$ is the list.

[This is similar to Hoare's work on data representations [11]. But note that Hoare's abstraction functions worked on states in Simula classes; it is thus more directly connected to the work in subsequent sections, on the object form of data abstraction.]

3.3 Parametricity and Algebra

Let us say that a *parametric model* of polymorphism is one where all elements $p \in \forall\alpha\,.\,T(\alpha)$ satisfy the relational parametricity condition

For all relations $R : D \leftrightarrow E$, $(p[D], p[E]) \in T(R)$.

This condition is stated informally here: Precise treatments include [36, 14, 25, 1, 35, 9]. Parametric models are not easy to find; the first was presented in [2].

We are interested not so much in the details of a formalization of parametricity as its consequences. As a specific example, consider the type $\forall\alpha\,.\,\alpha \to \alpha$. We can argue that there is only one parametric element p of this type as follows. Consider any type D and any element $d \in D$. Then consider the relation $R : D \leftrightarrow D$ consisting only of the pair $\langle d, d\rangle$. The induced relation $R \to R$ relates two functions $f : D \to D$, $g : D \to D$ just in case f and g preserve R, so that they both must map d to d. Relational parametricity of p says that $p[D]$ must be $(R \to R)$-related to itself, so p must be the denotation of the polymorphic identity function $\Lambda\alpha\,.\,\lambda x : \alpha\,.\,x$.

A wide variety of results of this form follow from relational parametricity. These include encodings of products, sums, initial algebras, and final coalgebras [34, 2, 36, 25, 9].

For instance, the type of polymorphic Church Numerals is

$$\forall\alpha\,.\,(\alpha \to \alpha) \to (\alpha \to \alpha)$$

The polymorphic expresison corresponding to natural number i is $\Lambda\alpha\,.\,\lambda f : \alpha \to \alpha\,.\,\lambda x : \alpha\,.\,f^i x$, where $f^{n+1}x = f(f^n x)$ and $f^0 x = x$. Relational parametricity implies that the type of Church numerals is a (least) solution to the type equation $D \cong D + 1$, where $+$ is coproduct and 1 is a terminal object, which, when the equation is understood in terms of sets, yields (one way of describing) the natural numbers.

We began by saying that parametricity should mean that a polymorphic function does not "look under" a type variable; in that sense, the type variables are abstract. Relational parametricity says, more precisely, that a polymorphic function must be invariant with respect to relations between different instantiations of a type variable. The intuitive connection between the informal and rigorous notions is that, if you were to act on any non-trivial information gathered by looking under a variable, then this should be detected by changing representation. This is why relational parametricity provides a satisfying, if not final, formalization of abstraction.

4 Local State

Now we turn out attention to the object form of data abstraction. As discussed in Section 2, this does not necessarily entail working with an explicitly object-oriented language, but rather considering that data abstractions can be realized using a combination of procedures and local state. A more explicitly object-oriented setup is considered in the next section.

4.1 Parametricity and Local Variables

Local state has been recognized as a thorny problem in traditional denotational semantics. The difficulties stem from the fact that a procedure may gain indirect access to a storage variable that is not even in existence (in the sense of not yet being allocated) at the time when the procedure is declared; this can happen if the variable is passed as an argument, or if it is referenced within the body of a procedure parameter. However, the procedure has no direct way to refer to the variable: Access is limited to that provided to it by arguments. This limited access is not captured by traditional models of imperative languages, or in most program logics [16].

O'Hearn and Tennent suggested that the difficulties in the semantics of local state had essentially to do with data abstraction [21]:

> We propose that a non-local procedure is *independent* of local state in the same way that a polymorphic function is *independent* of types to which it is instantiated.

If this thesis were true, then it would, at least partially, answer the question about the relationship between the object and ADT forms of data abstraction posed in Section 2. For, in Section 3 we outlined the intimate connection between polymorphism and ADT-abstraction, and in Section 2 we saw that (what we referred to as) the object form of data abstraction can be seen as arising from a combination of procedures and local state.

To support their proposal, O'Hearn and Tennent developed a model for a small imperative language, Idealized Algol [33], in which relational parametricity was used to govern the interaction between non-local procedures and local state. The main idea can be understood in terms of a mapping from programs in the imperative language into the polymorphic λ-calculus. We will not give the complete semantics here, but instead consider an extended example, which we hope conveys the essential ideas.

Consider the following program for a counter object

$$\mathbf{new}\, x \,.\, x := 0;\ P(x := x + 1, x)$$

Here, a "client" procedure P is passed two "methods," a command for incrementing the local variable and an expression thunk for reading its value. (If we λ-abstract on P we obtain a counter class, that is, a way to generate counters for arbitrary clients.) Since x is local, the client can never access it directly, but only through using the two arguments.

The semantics works by using the traditional idea of state transformations, but where polymorphism is employed to give more detailed types to the state. The client P is a procedure that accepts a command (or parameterless procedure) and an expression thunk (or parameterless, integer-valued, procedure) as arguments, and produces a command. It corresponds to a polymorphic function

$$p : \forall\beta \,.\, (\alpha \times \beta \to \alpha \times \beta) \times (\alpha \times \beta \to \alpha \times \beta \times \mathbf{nat}) \to (\alpha \times \beta \to \alpha \times \beta)$$

Here, we regard α as the portion of the store that p knows about directly, and β as ranging over pieces of local state in other objects, of which p can only have indirect knowledge. So, we read the type as follows:

> for all pieces of local state β, p accepts a command and an expression that work over α and β, and produces a command over α and β.

The block in which the counter object is declared is then a transformation on the non-local state α

$$\begin{array}{l}\underline{\lambda} s : \alpha \,.\, \mathbf{let}\ [s', n'] \ \mathbf{be}\ p[\mathbf{nat}]\langle id_\alpha \times succ,\ id_\alpha \times copy\rangle\, [s, 0] \\ \qquad \mathbf{in}\ s' \\ \qquad : \alpha \to \alpha\end{array}$$

The key point here is that the β component in the type of p is instantiated with **nat**, the type of the local variable.

This way of arranging the semantics gives a direct connection between parametric polymorphism and local state. The idea that p cannot access the local variable, except through provided methods, is modelled by the relational parametricity property of $\forall$. As with the encodings of ADT's, this gives rise to a collection of relational principles for reasoning about local state.

4.2 Linearity and Polymorphism

The parametricity semantics of state is successful in many respects, but for one problem: It does not account well for the *irreversible* nature of state change, where an assignment statement destroys the state of the store rather than producing a new copy. This problem has been addressed by O'Hearn and Reynolds [20], by moving to a form of polymorphism based on linear logic [7].

As an example, consider a procedure that accepts a command as an argument, and produces a command as a result. In the linear version of the parametricity semantics such a procedure has type

$$\forall \beta \,.\, (\alpha \otimes \beta \multimap \alpha \otimes \beta) \to (\alpha \otimes \beta \multimap \alpha \otimes \beta)$$

For the reader unfamiliar with linear logic, the basic idea is that the linear function type $\multimap$ is for functions that use their arguments exactly once [7, 37]. After a function uses its argument, it can never do so again because that would constitute two uses; this corresponds to the destructive nature of state change. The $\otimes$ is a kind of product where the elements are tightly coupled, where, to use an element of such a product, you must use both components exactly once. In the type we have just given the state is the only part subject to this linearity. The role of $\to$ is to enable the whole command argument, of type $(\alpha \otimes \beta \multimap \alpha \otimes \beta)$, to be used many times, as is common in an imperative language.

This move to linear typing enables a representation theorem, analogous to the characterization of the type of Church Numerals discussed in Section 3. For instance, the polymorphic type just pictured is a (least) solution to the domain equation for deterministic resumptions [27]

$$D \ \cong\ S \multimap S \oplus (S \otimes D)$$

This result is formulated for a model where $\multimap$ is the strict function space construct from domain theory, $\otimes$ is smash product, and $\oplus$ is coalesced sum. In the domain equation, S is a flat cpo which interprets α.

This representation result connects up two very different readings of imperative procedures. The polymorphic type suggests a reading where

> for all possible pieces of local state β, p accepts a command acting on α and β and produces a command acting on the same state sets.

The domain equation suggests a completely different reading, where

> p starts by possibly changing the state. It then either halts, or uses its argument once and resumes.

The polymorphic reading is "internal," in that it mentions the local states, but with parametricity governing the way that local state is accessed. The resumption reading, on the other hand, is "external," in that local states are not mentioned at all. We may regard the representation result connecting the "internal" and "external" readings as providing evidence in favour of the thesis concerning parametricity and local variables stated at the beginning of this section.

5 Object Encodings

The polymorphic λ-calculus has played a central role in research on the foundations of object-oriented languages. In this section we look at two object encodings from that work.

5.1 Two Encodings

The first encoding is "objects as packages," as put forth by Pierce and Turner [24]. They describe the basic idea thus:

> Both the state of an object and the methods acting upon it are visible... with the existential type protecting the state from external access.

For example, a counter object would be given the type

$$\exists\alpha\,.\,\alpha \times (\alpha \to \mathbf{nat} \times \alpha)$$

Here, the leftmost α component of $\alpha \times (\alpha \to \mathbf{nat} \times \alpha)$ is where the state resides, and the $(\alpha \to \mathbf{nat} \times \alpha)$ component breaks down into a function of type $\alpha \to \mathbf{nat}$ for extracting the current value of a counter and a function of type $\alpha \to \alpha$ for incrementing the local state.

This encoding of objects appears to rest on a similar conception of local state to that discussed in the last section, except that $\exists$ is used instead of $\forall$. It was presented by Pierce and Turner via examples, and then further analyzed over a period of time.

First, Hofmann and Pierce [12] provided a source language for the encodings, an extension of the polymorphic λ-calculus with a new type-forming operator

$$\mathbf{Object}\,\alpha\,.\,T\alpha$$

For the counter type $T\alpha$ is $\mathbf{nat} \times \alpha$. It gives the "interface type information" for a counter.

In addition to the package encoding, Hofmann and Pierce provided a different encoding, based on recursive records, which (they indicated) was a synthesis of ideas implicit in previous works.

Objects as Packages

$$P[\![\mathbf{Object}\,\alpha\,.\,T\alpha]\!] \;=\; \exists\alpha\,.\,\alpha \times\; (\alpha \to T\alpha)$$

Objects as Recursive Records

$$RR[\![\mathbf{Object}\,\alpha\,.\,T\alpha]\!] \;=\; \mu\,\alpha\,.\,T\alpha$$

Here, $\mu\alpha\,.\,T\alpha$ is a (least) solution to the equation $D \cong TD$.

It is useful to revisit the type of a counter object, from the point of view of recursive records.

$$\mu\alpha\,.\,\mathbf{nat} \times \alpha$$

This gives rise to the following reading:

If p is an object of counter type, you can read its current value $\pi_0 p$, or you can apply the increment method and obtain an updated object $\pi_1 p$.

We call this view "external" because it characterizes an object behaviour in terms of a pattern of interactions with it, instead of in terms of a hidden internal state. Similarly, we call the packages view "internal."

These encodings were compared recently by Bruce, Cardelli and Pierce [3]. They considered two other encodings as well, and made their comparison along a number of dimensions, including suitability for inheritance, method update, and binary methods. For our present concerns, however, one of their observations is especially important. They remarked that, if recursion is present in the language, the two encodings give different results. And one certainly does want recursion, particularly since this is the means of modelling the "self" construct [30].

The authors of [3] seem annoyed. The reason could be that computational intuition suggests that the internal and external views, given by the package and recursive record encodings, should amount two ways of describing the same thing. So the disagreement of the two encodings is disappointing.

However, as with the work on local state, the situation can be improved by using a linear polymorphic λ-calculus as the target of the encodings.

5.2 Objects as Packages, Linearly

We can define a modified package encoding, as follows.

Objects as Packages

$$P[\![\mathbf{Object}\,.\,\alpha\,.\,T\alpha]\!] \;=\; \exists\alpha\,.\,\alpha\otimes !(\alpha\multimap T\alpha)$$

The reader not familiar with linear logic might enjoy the following, somewhat fanciful, reading of the new package interpretation. As before, an object has some state, which is protected from access by an existential type. But now, when using an object the state *must* be given to one of the methods, since the occurrence of α to the left of $\otimes$ has no "!". The methods, however, need to be available for reuse on subsequent invocations; that's the reason for the "!" in $!(\alpha\multimap T\alpha)$.

As a consequence of general observations of Plotkin [28], the linear package encoding is equivalent to that for recursive records. More precisely, the two encodings will be isomorphic, if we presume a relational parametricity property for $\forall$, and hence $\exists$ (and if T is positive in α, as required by Hofmann and Pierce). This seems to be a satisfying result, connecting the "internal" objects-as-packages and "external" objects-as-recursive-records views. Just as was the case with local state, it provides support for Pierce and Turner's viewpoint, as expressed in the quote at the beginning of this section.

6 Conclusion

Throughout the course of the paper we have tried to give an idea of how the polymorphic λ-calculus, and relational parametricity, provide a powerful account of data abstraction.

Along the way we also collected evidence in favour of the idea that the hiding aspect of the object and ADT forms of data abstraction are based on the same underlying semantic mechanism. However, despite this positive evidence, the jury is still out on whether they are actually the same.

Consider the following block, where P is a non-local procedure that accepts a storage cell as an argument and Q is a parameterless non-local procedure.

$$\mathbf{new}\, x\,.P(x);\ x := 1;\ Q;\ \mathbf{if}\ (x = 1)\ \mathbf{then\ diverge\ else}\ x := 1$$

Does this block diverge if control gets beyond Q? It depends. If the language does not allow the cell denoted by x (as opposed to its contents) to be stored by P, then Q will have no access to x and the conditional test will succeed. But if P can store x, say in a global variable z, then Q could subsequently read x out of z and set its contents to 2. In this case, the test would fail.

The latter case is a form of the phenomenon that Milner has labelled *extrusion* [17]. Extrusion occurrs when a locally declared entity is passed out to a non-local procedure or process, which remembers the local entity, in order to use it again in the future. This phenomenon arises in the presence of storable procedures, as in Scheme or ML, π-calculus channels, Pascal-style pointers, or storable object identities. Extrusion is so fundamental in object-oriented programming that it forms the basis for Hewitt's Actor model [10], and even shows up at an early stage in beginner's Java programs (as in the procedure call `AddActionListener(this)`).

The problem, as far as our "evidence" is concerned, is that it is not evident how to model this form of extrusion satisfactorily in polymorphic λ-calculus. In terms of the semantics of local state from Section 4, the type of P quantifies over all possible pieces of local state, and these are regarded as being completely independent of non-local state. In contrast, if P can store x then there is a potential dependency between the local and non-local state. Similar remarks apply to the objects-as-packages encoding from Section 5.

One reaction to this situation is simply that the answer to our question of Section 2 is no: The hiding aspects of the ADT and object forms of data abstraction are different. There does not appear to be any way to achieve the kind of behaviour exhibited by extrusion with ADT's, unless one includes extra primitives (such as pointers) that go beyond the ADT concept.

Another reaction is that it does not constitute a difference; rather, extrusion breaks the abstractness of local state. In the program block above, notice how knowledge of x passed from P to Q, behind the scenes so to speak. In the worst case, a locally declared reference can leak through an entire system, whereby it ceases to be local in any reasonable sense. This second reaction is summed up well by Hogg's [13] colorful declaration that "The big lie of object-oriented programming is that objects provide encapsulation."

I do not know which of these reactions, if either, is right. But I do believe that we would benefit from a better understanding of data abstraction, in the presence of pointers.

Acknowledgements. This note is based on a lecture "Objects, Local State and Linear Polymorphism" I gave at the Foundations of Object-Oriented Languages workshop in San Diego, in January 1998. Thanks to Jon Riecke for helpful comments on the presentation. The support by the UK EPSRC is gratefully acknowledged.

References

[1] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121(1-2):9–58, December 1993.

[2] S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(10):35–64, 1990. *Corrigendum* in 71(3):431, 1990.

[3] K.B. Bruce, L. Cardelli, and B.C. Pierce. Comparing object encodings. In *Invited lecture at Third Workshop on Foundations of Object Oriented Languages (FOOL 3)*, July 1996. Available electronically in informal workshop proceedings.

[4] W. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, Berlin, 1991.

[5] M. P. Fiore, A. Jung, E. Moggi, P. O'Hearn, J. Riecke, G. Rosolini, and I. Stark. Domains and denotational semantics: History, accomplishments and open problems. In number 59 of *Bulletin of the European Association for Theoretical Computer Science*, pager 227–256, 1996.

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

[7] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987.

[8] D. Gries, editor. *Programming Methodology, A Collection of Articles by IFIP WG 2.3*. Springer-Verlag, New York, 1978.

[9] R. Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4(1):71–109, March 1994.

[10] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):107–118, 1977.

[11] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972. Reprinted in [8], pages 269-281.

[12] M. Hofmann and B. Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995.

[13] J. Hogg. Islands: aliasing protection in object-oriented languages. *OOPSLA 91 Proceedings*, 1991.

[14] QingMing Ma and J. C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In S. Brookes et al., editors, *Mathematical Foundations of Programming Semantics, Proceedings of the 7th International Conference*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40. Springer-Verlag, Berlin, 1992, Pittsburgh, PA, March 1991.

[15] D. B. MacQueen. Using dependent types to express modular structure. In *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg Beach, Florida, 1986.

[16] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In POPL [29], pages 191–203.

[17] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[18] J. C. Mitchell and R. Harper. The essence of ML. In POPL [29], pages 28–46.

[19] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Trans. Programming Languages and Systems*, 10(3):470–502, 1988.

[20] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *J. ACM*, 1998. To appear.

[21] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, May 1995. Also in [22], pages 109–164.

[22] P. W. O'Hearn and R. D. Tennent, editors. *Algol-like Languages*, volume 2. Birkhauser, Boston, 1997.

[23] P. W. O'Hearn and R. D. Tennent, editors. *Algol-like Languages*, volume 1. Birkhauser, Boston, 1997.

[24] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[25] G. Plotkin and M. Abadi. A logic for parametric polymorphism. In M. Bezen and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375, Utrecht, The Netherlands, March 1993. Springer-Verlag, Berlin.

[26] G. D. Plotkin. Lambda-definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, 1980.

[27] G. D. Plotkin. *Domains*. Lecture notes. Available from `ftp://ftp.dcs.ed.ac.uk/pub/gdp/dom.ps.Z`, 1983.

[28] G.D. Plotkin. Type theory and recursion. Slides from ScottFest talk, 1993.

[29] *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, 1988. ACM, New York.

[30] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 289–297, Snowbird, Utah, jul 1988.

[31] J. C. Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[32] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In S. A. Schuman, editor, *New Advances in Algorithmic Languages 1975*, pages 157–168. Inst. de Reserche d'Informatique et d'Automatique, Rocquencourt, France, 1975. Reprinted in [8], pages 309-317.

[33] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, October 1981. North-Holland, Amsterdam. Also in [23], pages 67-88.

[34] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North Holland, Amsterdam, 1983.

[35] E. P. Robinson and G. Rosolini. Reflexive graphs and parametric polymorphism. In *Proceedings, 9th Annual IEEE Symposium on Logic in Computer Science*, Paris, 1994. IEEE Computer Society Press, Los Alamitos, California.

[36] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359, 4th International Symposium, Imperial College, London, September 1989. ACM, New York.

[37] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC-2 Working Conference on Programming Concepts and Methods*, pages 347–359, Sea of Galilee, Israel, April 1990. North Holland, Amsterdam.