Parametricity and Unboxing with Unpointed Types

John Launchbury[1] and Ross Paterson[2]

[1] Oregon Graduate Institute, P.O. Box 91000, Portland, OR 97291-1000, USA
[2] Department of Computing, Imperial College, London SW7, UK

Abstract. In lazy functional languages, $\perp$ is typically an element of every type. While this provides great flexibility, it also comes at a cost. In this paper we explore the consequences of allowing unpointed types in a lazy functional language like Haskell. We use the type (and class) system to keep track of pointedness, and show the consequences for parametricity and for controlling evaluation order and unboxing.

1 Introduction

Ever since Scott and others showed how to use pointed CPOs (i.e. with bottoms) to give meaning to general recursion, both over values (including functions), and over types themselves, functional languages seem to have been wedded to the concept. Languages like Haskell [5] model types by appropriate CPOs and, because non-terminating computations can happen at any type, all the CPOs are pointed. This gives significant flexibility. In particular, values of any type may be defined using recursion.

1.1 Parametricity

There are associated costs, however. When reasoning about programs, one must also allow for the possibility of non-termination, even if a function is in fact total. Similarly, the general parametricity theorems that follow from polymorphic types (popularized by Wadler as "Theorems for Free" [15]) are rather weaker than in the pure simply-typed λ-calculus.

Parametricity was introduced by Reynolds to express the limits on behaviour that polymorphism induces [11]. For example, a function that has the type $\forall \alpha. \alpha \rightarrow \alpha$ can do nothing interesting to its argument: it can only return it. However, in a language like Haskell in which $\perp$ is an element of every type, a function of type $\forall \alpha. \alpha \rightarrow \alpha$ could also ignore its argument, and call itself in infinite recursion, i.e. return $\perp$. To allow for this, parametricity results must be weakened; the usual treatment is to require that all relations be strict (and inductive—if elements of two chains are related, then so are their limits). For example, consider the following polymorphic typings in which the quantification is given explicitly:

$$\begin{array}{l} \mathit{reverse} : \forall \alpha.\, \mathit{List}\ \alpha \rightarrow \mathit{List}\ \alpha \\ \mathit{foldr} : \forall \alpha, \beta.\, (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \mathit{List}\ \alpha \rightarrow \beta \end{array}$$

The parametricity theorem for *reverse*, in the simplified form where the relation is a function a, is

$$reverse \circ map\ a = map\ a \circ reverse \quad \text{for } a \text{ strict}$$

However, this assertion can easily be proved by induction without assuming strictness for a.

Similarly, the parametricity theorem for *foldr* is

$$b\ (f\ x\ y) = f'\ (a\ x)\ (b\ y) \Rightarrow b \circ (foldr\ f\ c) = foldr\ f'\ (b\ c) \circ map\ a$$
$$\text{for } a \text{ and } b \text{ strict}$$

but, again, a direct proof requires only b to be strict. In each case the parametricity theorem has given a significantly weaker result than a direct proof.

1.2 Operational Issues

There are also implementation costs associated with allowing all types to contain a bottom element. In a non-strict language, any expression that could be $\perp$ cannot be evaluated before it is known to be needed. Hence arguments are *boxed*, that is passed as pointers to (delayed) computations. Then, even something as straightforward as addition becomes costly: the arguments have to be evaluated, extracted, added, and reboxed. This compares rather poorly with a single machine instruction in C.

The bottom element has also been used to give the programmer some control over evaluation order, and hence also over space behaviour. It is well known that if a function is strict, an argument whose type is either primitive or an algebraic data type may be safely pre-evaluated. Hence, some languages have a sequencing operator, such as *strictify* or *seq*, intended to force evaluation of one expression before going on to another.

Unfortunately, an operator like *seq* is not as sensible for other argument types, even though they contain $\perp$. In the case of product types, *seq* would require some sort of interleaved evaluation; for function types this would be prohibitively expensive. Furthermore, because the implementation of such an operator depends to some extent on the argument type, it cannot be made polymorphic without weakening parametricity: all relations would have to be strict *and* bottom-reflecting. Optimization techniques that rely on parametricity would be lost.

Efforts to extend this sequencing effect to other types tend to complicate the denotational description. For example, instead of true products, Haskell has lifted tuples whose evaluation may be forced. While products satisfy a simple set of equations, lifted tuples are more difficult to reason about.

1.3 This Paper

In this paper we show how modern domain theory may be used in practice to influence the design of a lazy programming language like Haskell. In particular, we will not accept that $\bot$ is an element of every type, but will allow unpointed domains also. When recursion is used, that fact will be recorded in the type, stating which type parameters must be pointed.

There are two major consequences of all this. First, parametricity is repaired and returns to its former glory. When recursion is not used, there is no strictness side-condition on the theorems. Secondly, unpointed types can be used to control both evaluation order and unboxing of values, subsuming earlier work by Peyton Jones and Launchbury [9].

2 Pointed and Unpointed Domains

Domain theory was developed originally in order to provide solutions to recursive domain equations. Such solutions are needed to model the untyped λ-calculus, but also for recursively-defined data types in languages like Haskell and ML. As these solutions always exist for equations over pointed CPOs, the theory is often presented in exactly that framework.

It has long been noted that this makes for an asymmetrical category: domains are required to have a least element, but functions need not preserve it. As a consequence a number of constructions fail, the most important being sums. The theory of pointed CPOs comes equipped with two kinds of sum: coalesced sum, in which the bottom elements of the summands are merged; and separated sum, in which a completely fresh bottom element is introduced. Unfortunately, neither of these is a true (categorical) sum. The first introduces "confusion", the second contains "junk".

However, plain CPOs (that is, CPOs which do not necessarily have bottom elements, of which sets are a special case) do possess a categorical sum, namely the disjoint union of a pair of CPOs. The two summands are independent of each other and no extra elements have been introduced. Notice however that, as all domains are non-empty, a sum domain cannot have a bottom element.

CPOs also combine very well with the rest of domain theory, so long as recursive-domain equations (and recursive values likewise) are restricted. We shall give a brief account here; details may be found in standard texts [12, 2],

The first thing to note is that the category is still cartesian closed. That is, the one-point domain is terminal: for each other domain there is exactly one function from that domain to the one-point domain; products are built by cartesian product as before, with each component of the product completely independent of the other; and function spaces are constructed exactly as before, with currying an isomorphism. All this is the same as in the category of pointed CPOs.

The earlier separated sum, so widely used for data types in languages like Haskell, can be viewed as a categorical sum followed by an additional operation called lifting, which adds a fresh bottom element to a domain. Thus a data type definition like

$$\textbf{data}\ \mathit{Univ} \triangleq \mathit{Ch}\ \mathit{Char} \mid \mathit{Nu}\ \mathit{Int}$$

$$
\begin{aligned}
&() : \mathbb{1}\\
&\mathit{fst} : \forall\, \alpha, \beta. (\alpha \times \beta) \rightarrow \alpha\\
&\mathit{snd} : \forall\, \alpha, \beta. (\alpha \times \beta) \rightarrow \beta\\
&(,) : \forall\, \alpha, \beta. \alpha \rightarrow \beta \rightarrow (\alpha \times \beta)\\
&\mathit{inl} : \forall\, \alpha, \beta. \alpha \rightarrow (\alpha + \beta)\\
&\mathit{inr} : \forall\, \alpha, \beta. \beta \rightarrow (\alpha + \beta)\\
&\mathit{choose} : \forall\, \alpha, \beta, \gamma. (\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha + \beta) \rightarrow \gamma\\
&\mathit{fail} : \forall\, \alpha. \alpha_\perp\\
&\mathit{lift} : \forall\, \alpha. \alpha \rightarrow \alpha_\perp\\
&\mathit{ext} : \forall\, \alpha, \mathit{Pointed}\, \beta. (\alpha \rightarrow \beta) \rightarrow \alpha_\perp \rightarrow \beta\\
&\perp : \forall\, \mathit{Pointed}\, \alpha. \alpha
\end{aligned}
$$

Fig. 1. Constants

will be modelled by the domain (abusing notation slightly):

$$\mathit{Univ} \triangleq (\mathit{Char} + \mathit{Int})_\perp$$

This lifting construction also arises naturally, as the left adjoint of the (implicit) inclusion of the subcategory of pointed CPOs and strict functions.

This is all rather pleasant from a categorical perspective, because the five domain constructions above arise directly from adjunctions. What this means in practice is that the operations satisfy a rich and clean set of algebraic laws, making reasoning about them easier than if the laws were clouded with special cases and side-conditions.

A lazy functional language may be viewed as a form of the meta-language used in denotational semantics to talk about this category of CPOs. In the following, we shall introduce a tiny language of this sort.

3 Pointed and Unpointed Types

We shall consider a typed λ-calculus with the following types:

$$t ::= \mathbb{1} \mid t \times t \mid t \rightarrow t \mid t + t \mid t_\perp \mid \alpha$$

where α and other Greek letters are type variables. We shall not discuss explicit **let** polymorphism here, but the extension to include that case is entirely standard.

The first three type constructors are those of a cartesian closed category. As in the previous section, in the CPO model, the type $\mathbb{1}$ consists of a single value, $\times$ constructs products, and $\rightarrow$ constructs function spaces. The last two type constructors yield disjoint union and lifting.

The constants of our language, and their types, are given in Fig. 1. The type $\alpha_\perp$ includes a primitive non-terminating value *fail*.

Some types are qualified by a *Pointed* condition, indicating that they have a least element $\bot$. We can describe this class, and when the type constructors construct types that belong to it, with the following pseudo-Haskell:

$$\textbf{class}\ \mathit{Pointed}\ \alpha\ \textbf{where}\ \bot : \alpha$$

$$\textbf{instance}\ \mathit{Pointed}\ \mathbb{1}\ \textbf{where}\ \bot \triangleq ()$$

$$\textbf{instance}\ \mathit{Pointed}\ \alpha \wedge \mathit{Pointed}\ \beta \Rightarrow \mathit{Pointed}\ (\alpha \times \beta)\ \textbf{where}\ \bot \triangleq (\bot_\alpha, \bot_\beta)$$

$$\textbf{instance}\ \mathit{Pointed}\ \beta \Rightarrow \mathit{Pointed}\ (\alpha \rightarrow \beta)\ \textbf{where}\ \bot\ x \triangleq \bot_\beta$$

$$\textbf{instance}\ \mathit{Pointed}\ \alpha_\bot\ \textbf{where}\ \bot \triangleq \mathit{fail}$$

Note that a sum type is never *Pointed.* Also note that lifted types and *Pointed* types are not the same thing. All lifted types are *Pointed* but, while products of *Pointed* types are *Pointed,* they are not lifted.

With these definitions and rules, the *Pointed* restrictions are inferred by the usual Haskell algorithm [6, 16].

Functions that analyse a lifted type are defined using *ext*, and thus have a *Pointed* result. For example, suppose that the type of lifted integers *Int* is defined to be $\mathit{Int}^{\#}{}_\bot$, where $\mathit{Int}^{\#}$ is the type representing the (unlifted) set of integers. Then a primitive addition function, say

$$(+^{\#}) : \mathit{Int}^{\#} \rightarrow \mathit{Int}^{\#} \rightarrow \mathit{Int}^{\#}$$

operating on unlifted integers would be extended to the type of lifted integers as follows:

$$\begin{aligned}&(+) : \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int}\\ &x + y \triangleq \mathit{ext}\ (\lambda\, u.\ \mathit{ext}\ (\lambda\, v.\ \mathit{lift}\ (u +^{\#} v))\ y)\ x\end{aligned}$$

That is, the arguments to $+$ are "evaluated" using *ext*, then their values are extracted and added together by $+^{\#}$. Finally, the result of the addition is lifted, to make the result an element of the type of lifted integers. The type system would prohibit a function which attempted to return an unlifted integer, as the result type of *ext* must be pointed.

Similarly, the Haskell *case* operation on lifted sums is given by

$$\begin{aligned}&\mathit{case} : \forall\, \alpha, \beta, \mathit{Pointed}\ \gamma.\ (\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha + \beta)_\bot \rightarrow \gamma\\ &\mathit{case}\ f\ g\ x \triangleq \mathit{ext}\ (\mathit{choose}\ f\ g)\ x\end{aligned}$$

Of course, the full **case** construct is much richer, allowing nested pattern matching with guards. What we have here corresponds to the simplified version from Haskell's core language.

We can also use *ext* to define a polymorphic *seq* operation, as follows.

$$\begin{aligned}&\mathit{seq} : \forall\, \alpha, \mathit{Pointed}\ \beta.\ \alpha_\bot \rightarrow \beta \rightarrow \beta\\ &\mathit{seq}\ x\ y \triangleq \mathit{ext}\ (\lambda\, u.\, y)\ x\end{aligned}$$

The argument x is evaluated and if it is non-$\bot$ its value is thrown away by the constant function which returns y. Notice the restrictions on the type of *seq*.

$$
\begin{aligned}
&\forall x : \mathbb{1}.\, x = () \\
&\forall \alpha, \beta. \forall x : \alpha,\ y : \beta,\ p : \alpha \times \beta. \\
&\qquad \mathit{fst}\,(x, y) = x \\
&\qquad \mathit{snd}\,(x, y) = y \\
&\qquad p = (\mathit{fst}\, p, \mathit{snd}\, p) \\
&\forall \alpha, \beta, \gamma. \forall f : \alpha \to \gamma,\ g : \beta \to \gamma,\ x : \alpha,\ y : \beta,\ z : \alpha + \beta. \\
&\qquad \mathit{choose}\ f\ g\ (\mathit{inl}\ x) = f\ x \\
&\qquad \mathit{choose}\ f\ g\ (\mathit{inr}\ y) = g\ y \\
&\qquad (\exists x : \alpha.\, z = \mathit{inl}\ x) \vee (\exists y : \beta.\, z = \mathit{inr}\ y) \\
&\forall \alpha, \mathit{Pointed}\, \beta. \forall f : \alpha \to \beta,\ x : \alpha,\ z : \alpha_\perp. \\
&\qquad \mathit{ext}\ f\ \mathit{fail} = \perp \\
&\qquad \mathit{ext}\ f\ (\mathit{lift}\ x) = f\ x \\
&\qquad (z = \mathit{fail}) \vee (\exists x : \alpha.\, z = \mathit{lift}\ x) \\
&\qquad \mathit{fail} < \mathit{lift}\ x
\end{aligned}
$$

Fig. 2. Axioms

First, the x argument must be drawn from a *lifted* type, as produced by a **data** declaration, for example. As noted in Sect. 1.2, it is not sufficient for x merely to be of pointed type. Secondly, the result type must be *Pointed* as the evaluation of the whole expression will not terminate if x is $\perp$.

3.1 Axioms

The axioms of our language are given in Fig. 2. The first two groups, plus the β and η rules for λ-abstraction, are the usual axioms of a well-pointed cartesian closed category (that is, a category in which the axioms can be presented element-wise).

The forms of the axioms demonstrate their origins. While the type constructors $\mathbb{1}$, $\times$ and $\to$ are defined as right adjoints, $+$ and $(\cdot)_\perp$ are defined as left adjoints, and so their axioms have a different structure.

Proposition 1. *The axioms of Fig. 2 hold in the (general) CPO model.*

A simple consequence is the following theorem:

$$\forall\, \mathit{Pointed}\, \alpha. \forall x : \alpha.\, \perp \leq x$$

That is, $\perp$ is the least element of the types that have it.

4 Parametricity

So far, we have taken the types of the language and interpreted them as domains in a fairly standard way. However, that's not the only interpretation of types that is interesting. In particular, types may be interpreted as *relations between*

$$\begin{aligned}
\mathbf{1}\ (\overline{x}) &\triangleq \mathit{true} \\
(A \times B)\ (\overline{p}) &\triangleq A\ (\mathit{fst}\,\overline{p}) \wedge B\ (\mathit{snd}\,\overline{p}) \\
(A \rightarrow B)\ (\overline{f}) &\triangleq \forall \overline{x}.\, A\ (\overline{x}) \Rightarrow B\ (\overline{f}\ \overline{x}) \\
(A + B)\ (\overline{x}) &\triangleq (\exists \overline{y}.\, \overline{x} = \mathit{inl}\,\overline{y} \wedge A\ (\overline{y}))\ \vee (\exists \overline{z}.\, \overline{x} = \mathit{inr}\,\overline{z} \wedge B\ (\overline{z})) \\
A_{\perp}\ (\overline{x}) &\triangleq (\overline{x} = \overline{\perp}) \vee (\exists \overline{y}.\, \overline{x} = \mathit{lift}\,\overline{y} \wedge A\ (\overline{y})) \\
&\text{where } (h\ \overline{x})_i \triangleq h\ x_i \text{ and } (\overline{f}\ \overline{x})_i \triangleq f_i\ x_i
\end{aligned}$$

Fig. 3. Actions on relations

domains, i.e. as logical formulas. It is this relational interpretation of types that allows us to derive the parametricity results.

The actions of the various type constructors on relations are given in Fig. 3. The relations are defined inductively over vectors of elements drawn from the corresponding types. If the vectors are all of length 1, then the relations simply define subsets. More commonly, we take the case where the vectors are of length 2, giving binary relations.

The following result is standard.

Proposition 2. *In any model satisfying the axioms of Fig. 2, the actions on relations preserve identity relations.*

An action that preserves identity relations is said to be *unitary* [10], or to have the Identity Extension Property [11].

We can also add other base types, in particular flat integers, etc, with their relational action being an identity relation on that type. Then as a consequence of this proposition, the relational interpretations of the types of primitive monomorphic constants will be theorems. This is not a great surprise: for example, the binary relational interpretation of the type of $+^{\#}$ is

$$\forall x_1, x_2, y_1, y_2 : \mathit{Int}^{\#}.\, x_1 = x_2 \Rightarrow (y_1 = y_2 \Rightarrow x_1 +^{\#} y_1 = x_2 +^{\#} y_2)$$

The real power of parametricity comes from polymorphic types. In the relational interpretations of such types, free type variables[3] may stand for any relation between any tuple of domains. The parametricity theorem [11] states that if all the constants satisfy the relational interpretation, then so do all λ-expressions built from them.

For example, suppose n is a λ-expression with type $\forall \alpha.\, (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. Then the following theorem holds:

$$\forall \overline{\alpha}.\, \forall A : \mathsf{Rel}(\overline{\alpha}).\, \forall \overline{s} : \overline{\alpha \rightarrow \alpha},\ \overline{z} : \overline{\alpha}.\, (\forall \overline{x} : \overline{\alpha}.\, A\ \overline{x} \Rightarrow A\ (\overline{s}\ \overline{x})) \Rightarrow A\ \overline{z} \Rightarrow A\ (n\ \overline{s}\ \overline{z})$$

In other words, from its type alone we can tell that n satisfies some sort of induction principle.

[3] In our notation we have been using universal quantification as a sort of "meta-notation" to emphasize which variables are free.

This much is standard. What is new here is a subclass of types, namely the *Pointed* class. Now $\perp$ has type $\forall\, \mathit{Pointed}\, \alpha.\, \alpha$, and the relational interpretation of this type is

$$\forall\, \overline{\alpha}.\, \forall\, \mathit{Pointed}\, A : \mathsf{Rel}(\overline{\alpha}).\, A(\overline{\perp})$$

To achieve this, we define

$$\mathit{Pointed}\, A \triangleq A(\overline{\perp})$$

That is, applied to relations, the *Pointed* constraint specifies strictness.

The following extension of a standard result is easily verified.

Proposition 3. *The relational interpretations of the types of the constants of Fig. 1 are provable from the axioms of Fig. 2.*

As a consequence, the parametricity theorem holds for any term of this calculus, i.e. any term satisfies the formula that is its relational interpretation.

5 Recursive Functions

To make our language useful, we must allow recursive functions. We introduce a new constant

$$\mathit{fix} : \forall\, \mathit{Pointed}\, \alpha.\, (\alpha \to \alpha) \to \alpha$$

defined by the usual Kleene construction:

$$\mathit{fix}\, f \triangleq \bigsqcup_{k=0}^{\infty} f^k(\perp)$$

This definition makes sense only if the base type has a least element $\perp$, hence the *Pointed* restriction in the type.

This definition also requires a $\bigsqcup$ operation on countable chains, i.e. that the base domain is complete. Similarly, this operation must be preserved by relations: if a relation A relates corresponding elements in a tuple of chains, it must relate their lubs. That is, the relation must be inductive. We have chosen to assume that all domains are complete, and so will also assume that all relations are inductive. These properties are preserved by our type constructors and their actions on relations. [4]

Then, by construction, *fix* is parametric, i.e. it satisfies the relational interpretation of its type:

$$\forall\, \overline{\alpha}.\, \forall\, \mathit{Pointed}\, A : \mathsf{Rel}(\overline{\alpha}).\, (\forall\, \overline{x} : \overline{\alpha}.\, A\, \overline{x} \Rightarrow A(\overline{f}\, \overline{x})) \Rightarrow A(\mathit{fix}\, \overline{f})$$

This is just the familiar Scott-de Bakker induction rule.

[4] Alternatively, we could have allowed any domains and relations, and introduced a type class *Complete* for those that must be complete (and in the relational case, inductive), but the additional complexity seems to bring little benefit. However, the effort might be worthwhile in a specification language.

Now when we define functions, *Pointed* constraints are placed only where needed. Returning to the examples from the introduction, let us assume a unary type constructor *List*. We shall discuss recursive type definitions in detail in Sect. 7, but for now it suffices to note that since it is defined as a lifted sum, *List* α is *Pointed* even if α is not. The function *reverse* is the fixed point of a function of type

$$\forall\,\alpha.\,(\mathit{List}\;\alpha \rightarrow \mathit{List}\;\alpha) \rightarrow (\mathit{List}\;\alpha \rightarrow \mathit{List}\;\alpha)$$

Since *List* α is always *Pointed*, so is *List* $\alpha \rightarrow$ *List* α. Hence the inferred type of *reverse* is

$$\mathit{reverse} : \forall\,\alpha.\;\mathit{List}\;\alpha \rightarrow \mathit{List}\;\alpha$$

without any condition on α. Similarly, the type of *foldr* is

$$\mathit{foldr} : \forall\,\alpha,\,\mathit{Pointed}\,\beta.\,(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \mathit{List}\;\alpha \rightarrow \beta$$

which describes the desired property. This time the use of recursion relies on the type β (but not α) being *Pointed*.

6 Operational Implications

None of the earlier material forces any changes to the operational model. The type system guarantees that we *could* implement unlifted types differently from lifted types, but it does not *require* that we do. In effect, we could model an element of an unpointed type by the corresponding element in the corresponding pointed type, and lose nothing. All the reasoning ability from the forgoing is still entirely valid.

However, we will claim that the semantically clean language we have presented is also ideal for the expression of such low-level concerns as sequencing and unboxing. Peyton Jones and Launchbury [9] present a closely related system, with the intention of describing when a value may be passed unboxed. It turns out that our system provides a better vehicle for doing the same, and with a greater degree of flexibility.

6.1 Unboxed Values

A value is said to be boxed if it represented by an indirection into the heap, say, rather than being represented directly by an appropriate bit-pattern. In a language like Haskell, there are three distinct reasons why values are boxed (that is, placed in the heap and passed by reference). First, it may be more efficient to pass around the address of a large data object than the object itself. Secondly, in order to implement a polymorphic function as a single piece of generic code, the values it manipulates must be packaged so that they all look the same. There has been a lot of recent work on minimizing the boxing and unboxing of values that arises in this way [7, 3, 14]. Finally, in lazy languages, arguments are not to be evaluated until it is known that their results are required, so arguments are passed as pointers to computations (so-called call-by-need). In order to use the same function whether the arguments are already evaluated or not (perhaps they were shared by some other computation which forced their evaluation) all arguments must be passed boxed. It is the last of these that we will address here.

6.2 Unpointed Types and Unboxed Values

The semantic notion of unpointed types and the operational notion of unboxed values are closely related: an expression of unpointed type must terminate, and thus may be safely evaluated and represented by a *value* in weak head normal form. Operationally this is just right: the value may be stored unboxed. If the value has already been evaluated, then it cannot possibly be $\perp$, so does not need to live in a pointed domain.

This relationship was first explored by Peyton Jones and Launchbury [9]. They introduce a class of unboxed types corresponding to our unlifted types, some primitive (e.g. unboxed integers), and others defined by the user using unboxed data type declarations. Also as here, they model these types using unpointed domains.

However, the big difference comes in the semantics of functions. Because they do not track the use of recursion, they are forced to model a function whose target is an unboxed type, by a function to the lifted version of that type. So if g has type $g : A \rightarrow B^{\#}$, where $B^{\#}$ is unboxed, the semantics models this by a function $A \rightarrow (B^{\#}{}_{\perp})$. That is, unboxed values are manipulated in a special strict sublanguage. This special treatment of unboxed types complicates the semantics, but they were able to salvage the usual transformations by imposing two restrictions on expressions, to be enforced by a modified type system:

1. An expression of unboxed type appearing as a function argument must be in weak head normal form. Thus if f is a function $f : B^{\#} \rightarrow C$, their language does not permit the expression $f\ (g\ x)$. Rather, $g\ x$ must be explicitly evaluated and bound to a variable, which may then be used as an argument.
2. Ordinary type variables cannot be instantiated to unboxed types.

However, once uses of recursion are recorded in the types, these restrictions vanish. One way of expressing the difference between the systems is that, whereas their semantics introduced lifting to model functions to unboxed types, we require that lifting show up in the source language whenever it is actually needed, so giving finer source-level control.

6.3 An Implementation Scheme

The only way in which a value of lifted type $t_{\perp}$ can be constructed is by using *lift*; the only way it can be scrutinized is by using *ext*. Operationally, *lift* corresponds to a return, leaving an element of type t on the top of the stack or in appropriate registers, depending on the convention of the actual implementation. (If this value is potentially sharable, it must also be copied into the heap, updating a boxed closure.) Similarly, *ext* corresponds to a context switch, going off to evaluate its second argument. If evaluation of the argument terminates, it will have performed a *lift*. Now *ext* can immediately (tail-) call its function argument, passing it the explicit value that had just been returned.

This evaluation scheme is reminiscent of the continuation-passing style that is rather effective for call-by-value computations [1]. More precisely, there is a one-to-one correspondence between elements of a type $t_\perp$ and functions of type $\forall\, \mathit{Pointed}\ \alpha.\, (t \rightarrow \alpha) \rightarrow \alpha$, given by the functions

$$v \mapsto \lambda k.\, \mathit{ext}\ k\ v \qquad\qquad f \mapsto f\ \mathit{lift}$$

To implement *choose* requires some convention about the layout of sums. Perhaps the tag is always the top word on the stack, for example. As *choose* operates on a sum, its argument will already have been evaluated, so *choose* can simply perform a branch on the basis of the tag.

6.4 Source-level Unboxing

Having outlined a possible implementation mechanism, we shall consider how we can take advantage of it in practice. Fortunately there is a lot of direct experience we can draw on here. As our mechanisms subsume the methods of Peyton Jones and Launchbury, we can use all their techniques; techniques which are used in practice every time the Glasgow Haskell compiler is used.

We will take two examples of the forms of optimization that can be achieved. The first concerns removing repeated attempts at evaluation, the second shows how to take advantage of simple strictness information.

At this point it is worth mentioning that it is not our intention that the typical programmer should ever see any of this. Mostly it will be done within the compiler. On the other hand, there are times when the programmer needs control of data layout, particularly when writing library code, or time-critical code. In such cases, the form of code presented here may be written by hand.

Eliminating Repeated Attempts at Evaluation. We may define a doubling function as follows.

$$\begin{aligned} &\mathit{double} : \mathit{Int} \rightarrow \mathit{Int} \\ &\mathit{double}\ x \triangleq x + x \end{aligned}$$

When *double* is called, its argument is represented by a possibly unevaluated heap closure. When the body is evaluated, the plus function is entered. Because + requires its arguments, the first argument is entered, evaluated, and its value extracted. Then the second argument is entered and, because it's already evaluated, its value is returned directly. Clearly the second evaluation is unnecessary, and one might hope that a clever code generator could spot this.

On the other hand, code generators are already rather complex, so offloading extra functionality on to the code generator may not be a good idea. If instead it is possible to express unboxing as a source to source transformation, then we can move such optimizations to an earlier phase in the pipeline.

In the case above, for example, we could unfold the definition of $+$ we gave earlier to obtain

$$\begin{aligned}&\mathit{double} : \mathit{Int} \to \mathit{Int}\\ &\mathit{double}\ x \triangleq \mathit{ext}\ (\lambda\, u.\ \mathit{ext}\ (\lambda\, v.\ \mathit{lift}\ (u +^{\#} v))\ x)\ x\end{aligned}$$

Now we can appeal to a law satisfied by *ext*, namely that

$$\mathit{ext}\ (\lambda\, u.\ \mathit{ext}\ (\lambda\, v.\ f\ u\ v)\ x)\ x = \mathit{ext}\ (\lambda\, u.\ f\ u\ u)\ x$$

That is, repeated scrutiny of a value can be replaced by a single one. Using this law, we obtain

$$\begin{aligned}&\mathit{double} : \mathit{Int} \to \mathit{Int}\\ &\mathit{double}\ x \triangleq \mathit{ext}\ (\lambda\, u.\ \mathit{lift}\ (u +^{\#} u))\ x\end{aligned}$$

Now x is scrutinized just once, and its $\mathit{Int}^{\#}$ component used twice.

Workers and Wrappers. To see how source-level unboxing can be used to take advantage of simple strictness analysis, consider the following iterative version of the factorial function.

$$\begin{aligned}&\mathit{fact} : \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\ &\mathit{fact}\ x\ n \triangleq \textbf{if}\ x == 0\ \textbf{then}\ n\ \textbf{else}\ \mathit{fact}\ (x-1)\ (n * x)\end{aligned}$$

Conventional strictness analysis tells us that *fact* was strict in each of its arguments, that is $\mathit{fact}\ x\ n = \bot$ if either x or n is $\bot$. This statement is not as useful for program transformation as the following equivalent form using lifting:

$$\mathit{fact}\ x\ n = \mathit{ext}\ (\lambda\, u.\ \mathit{ext}\ (\lambda\, v.\mathit{fact}\ (\mathit{lift}\ u)\ (\mathit{lift}\ v))\ n)\ x$$

This suggests a restructuring of *fact* into two functions: a wrapper (still called *fact*) which evaluates and unboxes its arguments, and a worker (called $\mathit{fact}^{\#}$) which received unboxed arguments and does the work. Such functions could be produced entirely mechanically to give the following.

$$\begin{aligned}&\mathit{fact} : \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\ &\mathit{fact}\ x\ n \triangleq \mathit{ext}\ (\lambda\, u.\ \mathit{ext}\ (\lambda\, v.\mathit{fact}^{\#}\ u\ v)\ n)\ x\\ &\mathit{fact}^{\#} : \mathit{Int}^{\#} \to \mathit{Int}^{\#} \to \mathit{Int}\\ &\mathit{fact}^{\#}\ u\ v \triangleq\ \textbf{if}\ \mathit{lift}\ u == 0\ \textbf{then}\ \mathit{lift}\ v\,\text{else}\ \mathit{fact}\ (\mathit{lift}\ u - 1)\ (\mathit{lift}\ v * \mathit{lift}\ u)\end{aligned}$$

In the body of $\mathit{fact}^{\#}$, the boxed versions of the arguments have simply been reconstructed. So far we have gained nothing. But now, let's adopt the principle that all wrappers are to be unfolded. After all, wrappers will be very short, non-recursive functions, that is, they recurse via the worker which we do not intend to unfold. In addition, let's suppose that all the "primitive" operations like $==$, $-$, $*$ and **if** are also defined in terms of workers and wrappers. For example,

$$\begin{aligned}&(==) : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}\\ &m == n \triangleq \mathit{ext}\ (\lambda\, u.\ \mathit{ext}\ (\lambda\, v.\ \mathit{lift}\ (u ==^{\#} v))\ n)\ m\end{aligned}$$

Unfolding the definition of $==$ brings a use of *ext* directly against an explicit use of *lift*. From the axioms earlier the two cancel as follows:

$$ext\ f\ (lift\ u) = f\ u$$

Unfolding all the wrappers (including *fact*, and 0 whose wrapper is of the form $0 = lift\, 0^{\#}$), and cancelling the explicit *lifts* yields the result:

$$\begin{aligned}&fact^{\#} : Int^{\#} \rightarrow Int^{\#} \rightarrow Int\\ &fact^{\#}\ u\ v \triangleq \mathbf{if}^{\#}\ u ==^{\#} 0^{\#}\ \mathbf{then}\ lift\, v\, \mathbf{else}\ fact^{\#}\ (u -^{\#} 1^{\#})\ (v *^{\#} u)\end{aligned}$$

So, on the initial call to *fact*, the arguments are evaluated and unboxed. From then on the computation proceeds without laziness, passing unboxed values to the tail-recursion. Finally, once the function terminates, a lifted integer is returned. If course the function may not terminate as x could have been negative. In this case the result is $\perp$. If we had tried to avoid the final lifting on the result, the type checker would object as we are using recursion—the result type must be *Pointed*.

6.5 Projection-based Strictness Analysis

Our explicit treatment of lifting is also suitable for exploiting the results of projection-based strictness analysis [17]. For example, the image of the head-strict projection on lists of integers is exactly the lists of unboxed integers. With explicit lifting, this may be formalized by factoring projections as embedding-projection pairs, as is done in a related paper [8].

7 Recursive Type Definitions

To complete the picture of the interaction between *Pointed* constraints and the features of a typical functional language, we now consider recursively defined types. The semantics of these types is customarily described using a colimit construction [13]. That is, to construct the fixed point of a type constructor F, one constructs the sequence of domains corresponding to $\mathbb{1}, F\ \mathbb{1}, F(F\ \mathbb{1}), \ldots$ with each domain embedded in the next. Usually, one assumes that all domains are pointed, but to obtain these embeddings it suffices that F preserve *Pointed*ness. Then the categorical colimit construction yields a *Pointed* type μ_F, with a pair of isomorphisms

$$\begin{aligned}in_F &: F\ \mu_F \rightarrow \mu_F\\ out_F &: \mu_F \rightarrow F\ \mu_F\end{aligned}$$

The interesting case is where the recursive type has parameters. The following treatment follows Pitts [10]. However, instead of assuming all types and relations are pointed, we shall use the *Pointed* constraint to keep track of exactly which types are required to be pointed in Pitts's proofs.

To sketch the general situation, we shall assume a single recursive type with one parameter. The extension to mutual recursion and more parameters is straightforward. To obtain the most general types [10], we first separate negative and positive occurrences of each variable, replacing each with a pair of variables. Suppose $F\ \alpha^-\ \alpha^+\ \beta^-\ \beta^+$ is a type constructor, with α^- and β^- occurring negatively, and α^+ and β^+ occurring positively. The recursive type will be $\mu_F\ \alpha^-\ \alpha^+$. As this type is constructed by iterating F and taking the colimit, we need a condition $\mathcal{C}$ (either *Pointed* or nothing) such that

$$\mathcal{C}\ \alpha^+ \wedge \mathit{Pointed}\ \beta^+ \Rightarrow \mathit{Pointed}\ (F\ \alpha^-\ \alpha^+\ \beta^-\ \beta^+)$$

Note that negative arguments play no role in these constraints, as they are ignored in the rule making $\rightarrow$ an instance of *Pointed.* The proofs of Pitts [10] are easily extended to establish that for any such F and for any α^+ satisfying $\mathcal{C}$, the recursive type $\mu_F\ \alpha^-\ \alpha^+$ exists and is *Pointed.* Moreover μ_F is unitary, and there is a domain interpretation for a pair of constants

$$\begin{aligned} \mathit{in}_F &: \forall\, \alpha^-, \mathcal{C}\ \alpha^+ .\, F\ \alpha^-\ \alpha^+\ (\mu_F\ \alpha^+\ \alpha^-)\ (\mu_F\ \alpha^-\ \alpha^+) \rightarrow \mu_F\ \alpha^-\ \alpha^+ \\ \mathit{out}_F &: \forall\, \alpha^-, \mathcal{C}\ \alpha^+ .\, \mu_F\ \alpha^-\ \alpha^+ \rightarrow F\ \alpha^-\ \alpha^+\ (\mu_F\ \alpha^+\ \alpha^-)\ (\mu_F\ \alpha^-\ \alpha^+) \end{aligned}$$

Further, these functions constitute an isomorphism pair, and satisfy the relational interpretations of their types (this is a form of structural induction).

For example, a language might allow definitions like

$$\begin{aligned} &\mathbf{type}\, \mathit{Pointed}\ \alpha \Rightarrow \mathit{Seq}\ \alpha \triangleq \alpha \times \mathit{Seq}\ \alpha \\ &\mathbf{type}\, \mathit{List}\ \alpha \triangleq (\mathbb{1} + \alpha \times \mathit{List}\ \alpha)_\perp \end{aligned}$$

describing infinite sequences and lists. In the former, the *Pointed* condition is required in order to make the whole type pointed. On the other hand, the type of lists is lifted, and is thus pointed without any condition on α.

Another example is the recursive type used in the definition of the *fix* function as Curry's Y combinator:

$$\begin{aligned} &\mathbf{type}\, \mathit{Pointed}\ \alpha^+ \Rightarrow A\ \alpha^-\ \alpha^+\ \beta^-\ \beta^+ \triangleq \beta^- \rightarrow \alpha^+ \\ &\mathit{fix} : \forall\, \mathit{Pointed}\ \alpha .\, (\alpha \rightarrow \alpha) \rightarrow \alpha \\ &\mathit{fix}\ f \triangleq\ z\ (\mathit{in}_A\ z)\, \mathbf{where}\, z \triangleq \lambda x .\, f\ (\mathit{out}_A\ x\ x) \end{aligned}$$

Here α^+ must be *Pointed* in order to make the recursive type *Pointed*, forcing the constraint on the type of *fix*. We could avoid the need for the *Pointed* constraint in the type definition by lifting it, but then *fix* must be defined using *ext*, and ends up with the same type as before. Both definitions yield fixed points. Together with the parametricity property of this type, this uniquely determines *fix*, so these definitions are equivalent to Kleene's (see section 5).

8 Acknowledgements

The fundamental ideas described in this paper have been around for a while. Some years ago, Phil Wadler suggested that something like the system in this paper might work.

We have benefited from discussions with Erik Meijer, and the paper has been improved by feedback from Tim Sheard, Andrew Tolmack and Andrew Moran.

After writing this paper, we became aware of the work of Brian Howard [4], who uses an equivalent treatment of lifting and pointed types to describe a language in which initial, final and retractive types co-exist.

References

1. Andrew A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
2. Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.
3. Fritz Henglein and J. Jørgensen. Formally optimal boxing. In *21st ACM Symp. on Principles of Programming Languages*, pages 213–226, Portland, OR, January 1994.
4. Brian T. Howard. Inductive, projective, and pointed types. In *ACM Int. Conf. on Functional Programming*, Philadelphia, May 1996.
5. Paul Hudak, Simon Peyton Jones, Philip Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language (Version 1.2). *SIGPLAN Notices*, 27(5), March 1992.
6. Stefan Kaes. Parametric overloading in polymorphic programming languages. In *2nd European Symp. on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer, 1988.
7. Xavier Leroy. Unboxed objects and polymorphic typing. In *19th ACM Symp. on Principles of Programming Languages*, pages 177–188, Albuquerque, NM, January 1992.
8. Ross Paterson. Compiling laziness using projections, October 1995. Draft.
9. Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Conf. on Functional Programming Languages and Computer Architecture*, pages 636–666, Cambridge, MA, 1991.
10. Andrew M. Pitts. Relational properties of domains. *Information and Computation*, to appear, 1996.
11. John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier, 1983.
12. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.
13. Mike B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, 1982.
14. Peter J. Thiemann. Unboxed values and polymorphic typing revisited. In *Conf. on Functional Programming Languages and Computer Architecture'95*, pages 24–35, June 1995.
15. Philip Wadler. Theorems for free! In *4th Conf. on Functional Programming Languages and Computer Architecture*, pages 347–359. IFIP, 1989.
16. Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *16th ACM Symp. on Principles of Programming Languages*, pages 60–76, 1989.
17. Philip Wadler and John Hughes. Projections for strictness analysis. In *Conf. on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, Portland, OR, 1987.