Math. Struct. in Comp. Science (1991), *vol.* 1, *pp.* 1–20

Projections for Polymorphic First-Order Strictness Analysis: *

John Hughes and John Launchbury
Department of Computing Science
University of Glasgow

Received August 1990

We apply the categorical properties of polymorphic functions to compile-time analysis, specifically projection-based strictness analysis. First we interpret parameterised types as functors in a suitable category, and show that they preserve monics and epics. Then we define "strong" and "weak" polymorphism, the latter admitting certain projections that are not polymorphic in the usual sense. We prove that, under the right conditions, a weakly polymorphic function is characterised by a single instance. It follows that the strictness analysis of one simple instance of a polymorphic function yields results that apply to all. We show how this theory may be applied. In comparison to earlier polymorphic strictness analysis methods, ours can apply polymorphic information to a particular instance very simply. The categorical approach simplifies our proofs, enabling them to be carried out at a higher level, and making them independent of the precise form of the programming language to be analysed. The major limitation of our results is that they apply only to first-order functions.

1. Introduction

This is not a paper about category theory. Rather it is about using categorical ideas to develop better optimising compilers for functional languages. The categorical fact we use is that every first-order polymorphic function is a natural transformation [28, 36]. This semantic property enables us to develop a semantic analysis method for polymorphic functions; it guides our intuitions and greatly simplifies our proofs, replacing structural inductions over terms by semantic arguments. These categorical proofs are not only at a higher level than the corresponding "syntactic" proofs would be, they are unspecific about the programming language in which polymorphic functions are expressed. The "parametricity" captured by naturality is a strong condition, much stronger than any that could be inferred from many other models of polymorphism. It is this greater strength that makes our approach feasible.

The particular semantic analysis technique we are interested in is projection-based

: * This is a completely revised version of a paper that originally appeared in the Symposium on Category Theory and Computer Science, Manchester 1989. We have added many proofs missing in the original and reworked the existing proofs to be more categorical and, hopefully, more elegant.

strictness analysis. We prove a new result in this paper, that results from this kind of analysis are, in a sense, polymorphic. This confirms an earlier conjecture [19], and shows how the technique can be applied to first-order polymorphic functions.

The paper is organised as follows. In the next section, we review projection-based strictness analysis very briefly. In Section 3 we introduce the types we will be working with: they are the objects of a category. We show that parameterised types are functors, with certain cancellation properties. In Section 4 we define strong and weak polymorphism: polymorphic functions in programming languages are strongly polymorphic, but we will need to use projections with a slightly weaker property. We prove that, under certain conditions, weakly polymorphic functions are characterised by any non-trivial instance. We can therefore analyse one monomorphic instance of a polymorphic function using existing techniques, and apply the results to every instance. In Section 5 we choose a finite set of projections for each type, suitable for use in a practical compiler. We call these specially chosen projections contexts, and we show examples of factorising contexts for compound types in order to facilitate application of the results of Section 4. We give a number of examples of polymorphic strictness analysis. Finally, in Section 6 we discuss related work and draw some conclusions.

2. Projections for Strictness Analysis

Early strictness analysis methods could discover nothing informative about functions on lazy data-structures, and projection based strictness analysis was developed in an attempt to solve this problem. Recall that, in domain theory, a projection is a function α such that $\alpha \circ \alpha = \alpha$ and $\alpha \sqsubseteq ID$ $(= \lambda x.x)$. The essential intuition is that a projection performs a certain amount of evaluation of a lazy data-structure. For example, the projection

$$\begin{array}{lcll} \alpha : Nat \times Nat \to Nat \times Nat & & & \\ \alpha\ (x, y) & = & (\bot, \bot) & \textbf{if } x = \bot \\ & = & (x, y) & \textbf{if } x \neq \bot \end{array}$$

may be thought of as evaluating the first component of a pair, while

$$\begin{array}{lcll} \beta : Nat \times Nat \to Nat \times Nat & & & \\ \beta\ (x, y) & = & (\bot, \bot) & \textbf{if } x = \bot \textbf{ or } y = \bot \\ & = & (x, y) & \textbf{otherwise} \end{array}$$

evaluates both. Now we can regard a function as β-strict—performing as much evaluation as β—if evaluating its argument with β before the call does not change its result. For example, the function

$$+ : Nat \times Nat \to Nat$$

evaluates both its arguments, and so

$$+ = + \circ \beta$$

More generally, there may be parts of a function's argument that are evaluated only if certain parts of its result are evaluated—a function may evaluate more or less of its

argument depending on context. For example,

$$\begin{aligned} &swap : Nat \times Nat \to Nat \times Nat \\ &swap\ (x, y) = (y, x) \end{aligned}$$

is not β-strict, but it is β-strict in a β-strict context since

$$\beta \circ swap = \beta \circ swap \circ \beta$$

Thus, if both components of *swap*'s result will be evaluated, then the components of its argument can be evaluated before the call without changing the meaning. We make the following definition:

Definition 1.
Let f be a function and α and β be projections. We say f is α-strict in a β-strict context if $\beta \circ f = \beta \circ f \circ \alpha$ (or equivalently, $\beta \circ f \sqsubseteq f \circ \alpha$). In this case we write $f : \beta \Rightarrow \alpha$ □

Projections capture the notion of evaluating a component of a data-structure; to capture evaluation of a single value we must embed it in a "data-structure" with a single component, which we can think of as representing an unevaluated closure. Thus we think of a closure of type t as an element of $t_\perp$, and we "evaluate" it with the projection

$$\begin{aligned} &STR_t : t_\perp \to t_\perp \\ &STR_t\ \perp \quad = \quad \perp \\ &STR_t\ (lift\ x) \quad = \quad \perp \qquad \textbf{if } x = \perp \\ &\qquad\qquad\ = \quad lift\ x \qquad \textbf{if } x \neq \perp \end{aligned}$$

(writing the lifted elements in the form *lift* x). Now, any function $f : s \to t$ induces a function $f_\perp : s_\perp \to t_\perp$ which behaves like f on elements of s, but maps the new $\perp$ to $\perp$. It is easy to show that f is strict if and only if $f_\perp : STR_t \Rightarrow STR_s$.

There are four fundamental projections over lifted types which capture various degrees of evaluation. We have already seen ID (no evaluation), and STR (evaluate). In addition there is ABS defined by

$$\begin{aligned} &ABS_t : t_\perp \to t_\perp \\ &ABS_t\ \perp \quad = \quad \perp \\ &ABS_t\ (lift\ x) \quad = \quad lift\ \perp \end{aligned}$$

and the constant bottom function BOT $(= \lambda x.\perp)$. These are discussed in more detail in Section 5.2. For now we note that strictness analysis can be performed by establishing facts of the form $f : \beta \Rightarrow \alpha$.

Usually f and β are given, and we want to find an α such that this property holds. We can always choose α to be ID, but this is uninformative: ID corresponds to performing no evaluation at all. We would like to find the smallest α such that $f : \beta \Rightarrow \alpha$. In general this is equivalent to the halting problem, but [33] gives methods for finding quite small αs, for monomorphic functions. These methods are beyond the scope of this paper. We remark only that they depend crucially on choosing a finite set of projections for each type, so that recursive equations can be solved effectively. The interested reader should consult [33] for a much fuller presentation of projection based strictness analysis.

3. Types and Functors

In order to model the sorts of types that occur in most modern functional languages following ML, we work within the category of Scott domains and continuous functions, which we write as $\mathcal{C}$. A Scott domain D is a bounded-complete, complete partial order possessing a countable basis $K(D)$ of finite elements (notation from [30]). For much of the time we will be working with two sub-categories: $\mathcal{C}_s$ whose morphisms are strict ($\perp$-preserving) functions; and $\mathcal{C}_r$ whose morphisms are both strict and $\perp$-reflecting (i.e. $f\ x = \perp \Leftrightarrow x = \perp$).

We model type constructions as functors. The constructions we will focus on are

$\mathbf{1}$	the one point domain
$X \oplus Y$	where X and Y are types
$X \otimes Y$	where X and Y are types
$X_\perp$	where X is a type
μF	where $F(X)$ is a type, provided that X is

$\mathbf{1}$ is the one point type. $X \oplus Y$ is the coalesced sum, with elements $\perp$, $inl\ x$ and $inr\ y$ (where $x \in X \setminus \{\perp\}$ and $y \in Y \setminus \{\perp\}$). $X \otimes Y$ is the smash product with elements $\perp$ and (x, y) (where $x \in X \setminus \{\perp\}$ and $y \in Y \setminus \{\perp\}$). $X_\perp$ has elements $\perp$ and $lift\ x$ (where $x \in X$).

Notice that we have not provided the function space construction. One reason for this is that currently projection-based strictness analysis is only understood for first order languages. Another is that it is not at all clear how to generalise our techniques to higher order.

The construction μF constructs the smallest type satisfying the recursive domain equation $\mu F = F(\mu F)$. It is given as the limit of the diagram

$$\mathbf{1} \xleftarrow{\iota_{F(\mathbf{1})}} F(\mathbf{1}) \xleftarrow{F(\iota_{F(\mathbf{1})})} F^2(\mathbf{1}) \longleftarrow \cdots$$

where, for all types t, we write $\iota_t : t \to \mathbf{1}$ for the unique arrow into the terminal object (the one point domain) given by $\iota_t\ x = \perp_{\mathbf{1}}$. We will generally omit the subscript from ι.

In languages with ML-like type systems, there are types of the form $X + Y$ (separated sum) and $X \times Y$ (where X and Y are types). These are related to the operators above as follows: for any types X and Y, it is the case that $X + Y \cong X_\perp \oplus Y_\perp$ and $(X \times Y)_\perp \cong X_\perp \otimes Y_\perp$. The reason we choose to use smash sum and product is that it is convenient for the constructions in Section 5. Furthermore, types usually regarded as base types may be defined in terms of these constructions. For example, the type of natural numbers is given by $Nat = \mu n\ .\ \mathbf{1}_\perp \oplus n$.

Lifting, coalesced sum and smash product are all functors over the strict sub-category $\mathcal{C}_s$. For any strict functions $f : t \to u$ and $g : v \to w$ we define

$$\begin{array}{lcl} f_\perp : t_\perp \to u_\perp & & \\ f_\perp\ \perp & = & \perp \\ f_\perp\ (lift\ a) & = & lift\ (f\ a) \end{array}$$

$$
\begin{array}{lcll}
f \oplus g : t \oplus v \to u \oplus w & & & \\
f \oplus g \quad \bot & = & \bot & \\
f \oplus g \ (inl\ a) & = & \bot & \textbf{if } f\ a = \bot \\
 & = & inl\ (f\ a) & \textbf{otherwise} \\
f \oplus g \ (inr\ c) & = & \bot & \textbf{if } g\ c = \bot \\
 & = & inr\ (g\ c) & \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{lcll}
f \otimes g : t \otimes v \to u \otimes w & & & \\
f \otimes g \quad \bot & = & \bot & \\
f \otimes g \ (a, c) & = & \bot & \textbf{if } f\ a = \bot \ \textbf{ or } g\ c = \bot \\
 & = & (f\ a, g\ c) & \textbf{otherwise}
\end{array}
$$

Then $\cdot_\bot : \mathcal{C}_s \to \mathcal{C}_s$ and $\oplus, \otimes : \mathcal{C}_s \times \mathcal{C}_s \to \mathcal{C}_s$. Note that the restriction to $\mathcal{C}_s$ is essential to ensure that $\oplus$ and $\otimes$ distribute over compositions: that is,

$$
\begin{aligned}
(f \circ g) \oplus (h \circ k) &= (f \oplus h) \circ (g \oplus k) \\
(f \circ g) \otimes (h \circ k) &= (f \otimes h) \circ (g \otimes k)
\end{aligned}
$$

There are counter-examples to both of these equations if f, g, h and k are allowed to be non-strict.

We note two properties that are immediate from the definitions: these functors are continuous on arrows, in the sense that they preserve the order and directed least upper bounds of arrows (for example, $f \sqsubseteq g \Rightarrow f_\bot \sqsubseteq g_\bot$), and they may also be viewed as functors over $\mathcal{C}_r$ by restriction (i.e. if both f and g are strict and $\bot$-reflecting then so are $f_\bot$, $f \oplus g$ and $f \otimes g$).

3.1. Parameterised Types

In ML, the type of a polymorphic function may be written as

$$f : F('a, 'b, \ldots) \to G('a, 'b, \ldots)$$

The $'a$ parameters act as type variables and the types $F('a, 'b, \ldots)$ and $G('a, 'b, \ldots)$ may be thought of as functions from the values of these variables. We model such parameterised types as functors from tuples of types[†] to types.

In order to interpret compound parameterised types such as $F('a, 'b) \oplus G('a, 'b)$ we overload the functors above to operate on parameterised types as well as monomorphic types, that is, on functors as well as objects. The following definition captures this.

Definition 2.
Let $\mathcal{D}$ be any category, and $F, G : \mathcal{D} \to \mathcal{C}_s$ and $H : \mathcal{D} \times \mathcal{C}_s \to \mathcal{C}_s$ be functors. Let t and

: † that is, objects of the product category $\mathcal{C}_s^n$.

u be any objects in $\mathcal{D}$ and $f : t \to u$ any arrow. We define

$$\begin{array}{l} \mathbf{1} : \mathcal{D} \to \mathcal{C}_s \\ \mathbf{1}\ t \;=\; \mathbf{1} \\ \mathbf{1}\ f \;=\; \iota_{\mathbf{1}} \end{array}$$

$$\begin{array}{l} F_\perp : \mathcal{D} \to \mathcal{C}_s \\ (F_\perp)\ t \;=\; (F\ t)_\perp \\ (F_\perp)\ f \;=\; (F\ f)_\perp \end{array}$$

$$\begin{array}{l} F \oplus G : \mathcal{D} \to \mathcal{C}_s \\ (F \oplus G)\ t \;=\; F\ t \oplus G\ t \\ (F \oplus G)\ f \;=\; F\ f \oplus G\ f \end{array}$$

$$\begin{array}{l} F \otimes G : \mathcal{D} \to \mathcal{C}_s \\ (F \otimes G)\ t \;=\; F\ t \otimes G\ t \\ (F \otimes G)\ f \;=\; F\ f \otimes G\ f \end{array}$$

□

The only constant functor we need is the constant functor $\mathbf{1}$, as the other "basic" functors we might want (such as booleans or integers) may be defined using $\mathbf{1}$, the operations above, and recursion (below).

In practice we expect the $\mathcal{D}$ used above to be $\mathcal{C}_s^n$ (for some n) representing the parameters of the recursive type. The category $\mathcal{C}_s^n$ is the product category whose objects $\vec{t}$ are n-tuples of domains[‡] and whose morphisms are likewise n-tuples of strict continuous functions. We write $sel_1, \ldots, sel_n$ for the selection functors, and given functors $H_1 : \mathcal{D} \to \mathcal{C}_s, \ldots, H_m : \mathcal{D} \to \mathcal{C}_s$ we can construct the functor $< H_1, \ldots, H_m > : \mathcal{D} \to \mathcal{C}_s^m$.

Recursive types are constructed using μ, where μH is a functor $\mu H : \mathcal{D} \to \mathcal{C}_s$. Its action on an object t of $\mathcal{D}$, namely $\mu H(t)$, is given by the limit of the diagram,

$$\mathbf{1} \xleftarrow{\iota_{H(t,\mathbf{1})}} H(t,\mathbf{1}) \xleftarrow{H(id_t,\, \iota_{H(t,\mathbf{1})})} H(t,H(t,\mathbf{1})) \xleftarrow{} \cdots$$

and $\mu H(f)$ is the unique mediating morphism from $\mu H(t)$ to $\mu H(u)$ shown in Figure 1. Its existence and uniqueness are guaranteed because $\mu H(u)$ is a limit (note that each of the rectangles commutes, so $\mu H(t)$ is a cone for the diagram for which $\mu H(u)$ is the limit). It is a standard domain-theoretic result that μH is continuous if H is, and if H preserves $\perp$-reflecting functions then so does μH (see [32] for details).

Having defined various means whereby new parameterised types (functors) may be constructed from old, we now focus our attention on those types which may be entirely constructed using only the operations above. This includes all ground types definable in ML-like languages.

‡ We use the notation $\vec{t}$ to emphasise the fact that the object is a tuple of types.

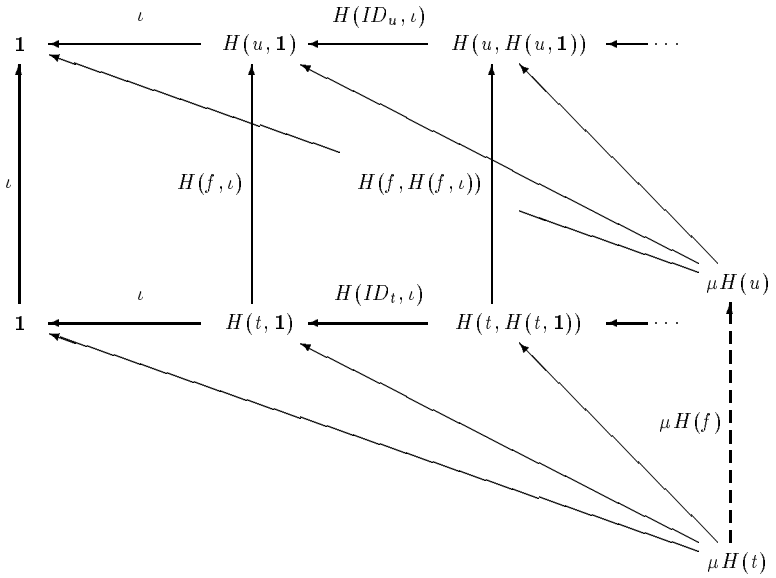


Fig. 1. Definition of μH

Definition 3.
A *type functor* is a functor constructed using $\mathbf{1}$, $\cdot_\perp$, $\oplus$, $\otimes$ and μ, together with composition, selection and tuple construction. □

Having no contravariant constructions, type functors are all covariant. Furthermore, because they are defined in terms of functors that act on $\mathcal{C}_r$ by restriction, every type functor $F : \mathcal{C}_s^n \to \mathcal{C}_s$ may be viewed as a functor $F : \mathcal{C}_r^n \to \mathcal{C}_r$, also by restriction. In addition, as $\mathcal{C}_s$ is a sub-category of $\mathcal{C}$, all such functors may be viewed as functors $F : \mathcal{C}_s^n \to \mathcal{C}$ and $F : \mathcal{C}_r^n \to \mathcal{C}$ by inclusion. Similarly, as all the basic constructions are continuous, so is every type functor.

We can extend the notion of a projection to $\mathcal{C}_s^n$. A morphism α in $\mathcal{C}_s^n$ is a projection if $\alpha \circ \alpha = \alpha$ and $\alpha \sqsubseteq ID$ (where $\mathcal{C}_s^n$ is ordered componentwise). By continuity, type functors map projections to projections.

Type functors also preserve certain cancellation properties. Recall that a morphism f is *monic* if $f \circ h = f \circ k$ implies $h = k$, and *epic* if $h \circ f = k \circ f$ implies $h = k$. Type functors map monics to monics and epics to epics. To prove this we present two lemmata that characterise monics and epics in categories of Scott domains.

Lemma 1. In $\mathcal{C}$, a function $f : t \to u$ is monic iff it is one-to-one.

Proof. Suppose that f is monic and that $f\ x = f\ y$ for some $x, y \in t$. Let $\bar{x}, \bar{y} : \mathbf{1} \to t$ be the constant functions that pick out x and y respectively. Then $f \circ \bar{x} = f \circ \bar{y}$, but as f is monic we deduce that $\bar{x} = \bar{y}$, that is, $x = y$.

Conversely, suppose that f is one-to-one and also that $f \circ h = f \circ k$ for some functions

$h, k : C \to t$. Then for each $x \in C$ it is the case that $f\ (h\ x) = f\ (k\ x)$. But, as f is one-to-one, we conclude that $h\ x = k\ x$ for each $x \in C$ and so $h = k$.

Lemma 2. In $\mathcal{C}$, a function $f : t \to u$ is epic iff it is onto $K(u)$ (the finite elements of u).

Proof. Suppose f is onto $K(u)$, and let $h, k : u \to C$ be any two functions such that $h \circ f = k \circ f$. Then for any $x \in u$,

$$\begin{array}{rcll} h\ x & = & h\ (\bigsqcup\{y \mid y \in K(u), y \sqsubseteq x\}) & [u \text{ algebraic}] \\ & = & \bigsqcup\{h\ y \mid y \in K(u), y \sqsubseteq x\} & [h \text{ continuous}] \\ & = & \bigsqcup\{h\ (f\ z) \mid f\ z \in K(u), f\ z \sqsubseteq x\} & [f \text{ surjective on } K(u)] \\ & = & \bigsqcup\{k\ (f\ z) \mid f\ z \in K(u), f\ z \sqsubseteq x\} & [h \circ f = k \circ f] \\ & = & \bigsqcup\{k\ y \mid y \in K(u), y \sqsubseteq x\} & [f \text{ surjective on } K(u)] \\ & = & k\ x & [k \text{ continuous}, u \text{ algebraic}] \end{array}$$

and so $h = k$ as required.

Conversely, suppose that f is not surjective on $K(u)$. Then there exists $b \in K(u)$ which is not in the image of f. Define $h, k : u \to \mathbf{1}_\perp \times u$ as follows.

$$\begin{array}{rcll} h\ x & = & (\perp,\ x) & \textbf{if } x \sqsubseteq b \\ & = & (\mathit{lift}\ \perp,\ x) & \textbf{otherwise} \end{array}$$

$$\begin{array}{rcll} k\ x & = & (\perp,\ x) & \textbf{if } x \sqsubseteq b \textbf{ and } x \neq b \\ & = & (\mathit{lift}\ \perp,\ x) & \textbf{otherwise} \end{array}$$

Both h and k are monotonic and, as b is finite, continuous. Furthermore, $h \circ f = k \circ f$ but as $h \neq k$ we conclude that f is not monic.

Exactly the same arguments hold in both $\mathcal{C}_s$ and $\mathcal{C}_r$. We use these results in the proof of the following theorem.

Theorem 3. Every type functor $F : \mathcal{C}_s^n \to \mathcal{C}_s$ maps monics to monics and epics to epics.

Proof. The proof is by induction on the form of the functor definition. The base cases of sel_k and $\mathbf{1}$ are trivial, and the cases $F = G \circ H$ and $F =< H_1, \ldots, H_m >$ are immediate inductions. For the other cases we will draw on the previous lemmata that monics are exactly the one-to-one functions, and that epics are exactly those functions which are surjective on finite elements.

Case: $F = G_\perp,\ F = G \oplus H,\ F = G \otimes H$

From the definitions, it is easy to see that if both $f : t \to u$ and $g : v \to w$ are one-to-one then each of $f_\perp,\ f \oplus g$ and $f \otimes g$ are also one-to-one. Thus, if G and H map monics to monics, then so do $G_\perp,\ G \oplus H$ and $G \otimes H$.

We can characterise the finite elements of $u_\perp,\ u \oplus w$ and $u \otimes w$ as follows

$$\begin{array}{rcl} K(u_\perp) & = & \{\mathit{lift}\ b \mid b \in K(u)\} \cup \{\perp\} \\ K(u \oplus w) & = & \{\mathit{inl}\ b \mid b \in K(u) \setminus \{\perp\}\} \cup \{\mathit{inr}\ d \mid d \in K(w) \setminus \{\perp\}\} \cup \{\perp\} \\ K(u \otimes w) & = & \{(b, d) \mid b \in K(u) \setminus \{\perp\}, d \in K(w) \setminus \{\perp\}\} \cup \{\perp\} \end{array}$$

If $f : t \to u$ and $g : v \to w$ are surjective on $K(u)$ and $K(w)$ respecively, then clearly each of $f_\perp,\ f \oplus g$ and $f \otimes g$ are surjective on $K(u_\perp),\ K(u \oplus w)$ and $K(u \otimes w)$. Thus, if G and H map epics to epics, then so do $G_\perp,\ G \oplus H$ and $G \otimes H$.

Case: $F = \mu H$
Suppose that f is monic and that $\mu H(f) \circ h = \mu H(f) \circ k$. We want to show that $h = k$. Consider the diagram in Figure 2.

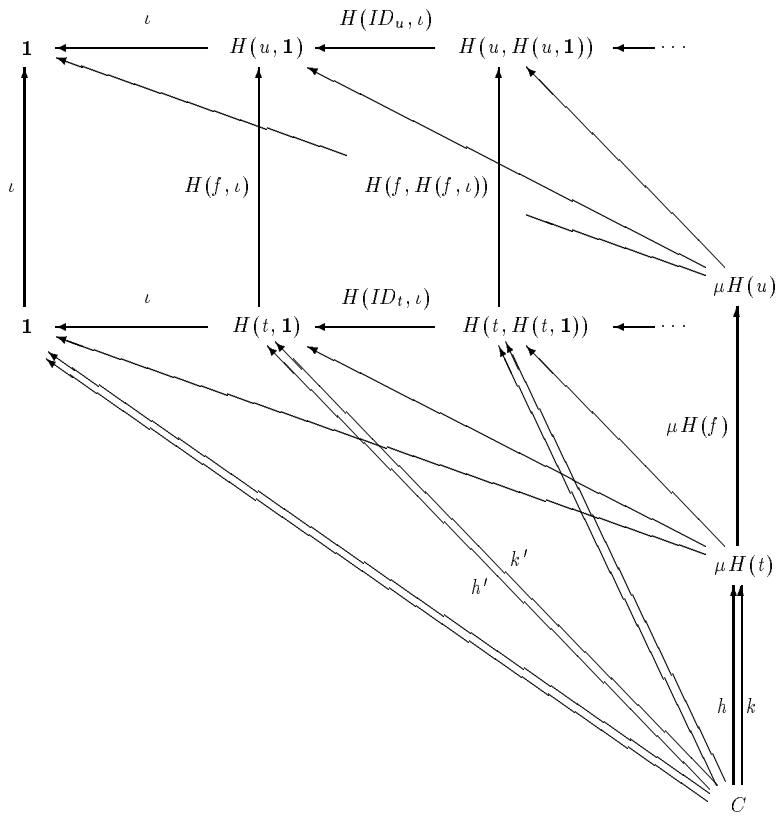


Fig. 2. μH preserves monics

Each pair of arrows from C is produced by composing h and k respectively with the cone arrows from $\mu H(t)$. So, for example, the arrow $h' : C \to H(t, \mathbf{1})$ is the composition of h with the cone arrow going from $\mu H(t)$ to $H(t, \mathbf{1})$.

By assumption, $\mu H(f) \circ h = \mu H(f) \circ k$ so the path $C, \mu H(t), \mu H(u), H(u, \mathbf{1})$ going via h is equal to the same path going via k. But the path $\mu H(t), \mu H(u), H(u, \mathbf{1})$ is equal to the path $\mu H(t), H(t, \mathbf{1}), H(u, \mathbf{1})$ as the "rectangle" commutes. Putting these two facts together allows us to conclude that $H(f, \iota) \circ h' = H(f, \iota) \circ k'$. By induction, $H(f, \iota)$ is monic if f is monic (note that $\iota : \mathbf{1} \to \mathbf{1}$ certainly is), and so $h' = k'$. Corresponding results hold for each of the other pairs of arrows from C. We can conclude, therefore, that these arrows together with C form a cone. As $\mu H(t)$ is the limit of the diagram there is a unique morphism from C to $\mu H(t)$ which makes the diagram commute. However, both h and k make the diagram commute, and so we deduce that $h = k$.

To prove the same result for epics we appeal to the well-known limit/colimit coincidence result (again see [32] for the details, or see [29] for a simpler account), and deduce that the limit of a diagram

$$\mathbf{1} \xleftarrow{\iota} H(\mathbf{1}) \xleftarrow{H(\iota)} H^2(\mathbf{1}) \longleftarrow \cdots$$

is isomorphic to the colimit of the corresponding diagram

$$\mathbf{1} \xrightarrow{\kappa} H(\mathbf{1}) \xrightarrow{H(\kappa)} H^2(\mathbf{1}) \longrightarrow \cdots$$

where $\kappa_t : \mathbf{1} \to t$ is given by $\kappa\ \bot_{\mathbf{1}} = \bot_t$. In this diagram, all the arrows are facing in the other direction and so the same proof goes through.

4. Polymorphism

Because type functors are covariant, polymorphic functions correspond to natural transformations. That is, each polymorphic function f is a collection of instances of the form $f_{\vec{t}} : F(\vec{t}) \to G(\vec{t})$, one for every object $\vec{t}$ of $\mathcal{C}_s^n$, such that for any morphism $\alpha : \vec{t} \to \vec{u}$ in $\mathcal{C}_s^n$, the following diagram commutes:

$$\begin{array}{ccc} F(\vec{t}) & \xrightarrow{f_{\vec{t}}} & G(\vec{t}) \\ {\scriptstyle F(\alpha)}\downarrow & & \downarrow{\scriptstyle G(\alpha)} \\ F(\vec{u}) & \xrightarrow[f_{\vec{u}}]{} & G(u) \end{array}$$

Thus we can regard a polymorphic function as a collection of (strongly related) monomorphic instances. However, there is a minor technicality. As F and G are functors $F, G : \mathcal{C}_s^n \to \mathcal{C}_s$ the definition only caters for *strict* polymorphic functions. However, we noted earlier that we may view F and G as functors $F, G : \mathcal{C}_s^n \to \mathcal{C}$ by inclusion, thereby allowing the instances of f to be arrows in the whole of $\mathcal{C}$ and not just in $\mathcal{C}_s$.

Definition 4.
Let $F, G : \mathcal{C}_s^n \to \mathcal{C}$ be type functors. If f is a natural transformation $f : F \to G$, we say that f is *strongly polymorphic*. □

"Strongly polymorphic" is the form of polymorphism seen by a programmer. That is, we assume that *every* polymorphic function definable within the programming language is strongly polymorphic. So long as all the primitive functions are strongly polymorphic, this assumption is valid. Examples of primitives which would *not* be strongly polymorphic are *strictify* and "polymorphic"-equality. In the case of equality, for example, the

naturality square would require $(x = y) \Leftrightarrow (h\ x = h\ y)$ to hold for all (strict) functions h. Clearly it does not. Other "polymorphic" primitives that fail to meet the naturality condition are selection from strict pairs, and tag-testing functions like *is_func*.

In strictness analysis, however, there are some "polymorphic" projections which, even though they are not strongly polymorphic, are crucial to the analysis, and so cannot be completely excluded. For example, recall the projection $STR_t : t_\perp \to t_\perp$ introduced in Section 2. We would like STR to be a polymorphic projection, but consider $STR_{\mathbf{2}}$, where $\mathbf{2}$ is the two-point domain $\mathbf{1}_\perp$. We would need in particular that

$$STR_{\mathbf{2}} \circ BOT_\perp = BOT_\perp \circ STR_{\mathbf{2}}$$

(where $BOT_\perp$ denotes the constant bottom function, lifted). But

$$(\ STR_{\mathbf{2}} \circ BOT_\perp)\ (lift\ (lift\ \perp)) = STR_{\mathbf{2}}\ (lift\ \perp) = \perp$$

whereas

$$(BOT_\perp \circ STR_{\mathbf{2}})\ (lift\ (lift\ \perp)) = BOT_\perp\ (lift\ (lift\ \perp)) = lift\ \perp$$

We shall have to weaken our definition of polymorphism slightly in order to admit STR. We do so by requiring the diagram above to commute for a smaller class of functions α. Recall that any type functor $F : \mathcal{C}_s^n \to \mathcal{C}_s$ may also be regarded as a functor $F : \mathcal{C}_r^n \to \mathcal{C}_r$ by restriction, and then also $F : \mathcal{C}_r^n \to \mathcal{C}$ by inclusion.

Definition 5.
Let $F, G : \mathcal{C}_r^n \to \mathcal{C}$ be type functors. If f is a natural transformation $f : F \to G$, we say that f is *weakly polymorphic*. □

Weakly polymorphic functions need only commute for those α's that are both strict and $\perp$-reflecting. Clearly, STR is weakly polymorphic. It is also clear that strongly polymorphic functions are weakly polymorphic (as any functor $F : \mathcal{C}_s^n \to \mathcal{C}$ may be regarded as a functor $F : \mathcal{C}_r^n \to \mathcal{C}$ by restriction). The other important projections ID, ABS and BOT are all strongly polymorphic.

4.1. Single-Instance Characterisation

We now embark on an extended argument to show that weakly polymorphic functions can, in certain circumstances, be characterised by one simple instance.

Let $F, G : \mathcal{C}_s^n \to \mathcal{C}_s$ be type functors, and $f, g : F \to G$ be weakly polymorphic. Suppose that we establish that f and g have equal $\vec{u}$ instances (that is, $f_{\vec{u}} = g_{\vec{u}}$) for *some* object $\vec{u}$ in $\mathcal{C}^n$. Then we will show that $f_{\vec{t}} = g_{\vec{t}}$ for *all* objects $\vec{t}$ in $\mathcal{C}^n$, subject to two conditions. Firstly, $\vec{u}$ (which is a tuple of types) must contain no trivial component (in the sense defined below)—such instances contain too little information to characterise a function. Secondly, for weakly polymorphic functions we must prove separately that $f \sqsubseteq g$, as even the $\mathbf{2}^n$ instances do not contain quite enough information to distinguish between some functions (notation: t^n denotes the object in $\mathcal{C}^n$ whose components are all t). This extra condition is unnecessary for strongly polymorphic functions.

Our eventual aim is to use the theorem to show that the results of analysing the strictness of a single instance of a polymorphic function can be applied to all instances.

We make the argument in four major stages.

(a) We show that if $f_{\vec{u}} = g_{\vec{u}}$ for some $\mathcal{C}^n$ object $\vec{u}$ containing no trivial component, then $f_{\mathbf{2}^n} = g_{\mathbf{2}^n}$.

(b) We show that if $f_{\mathbf{2}^n} = g_{\mathbf{2}^n}$ and $f \sqsubseteq g$ then $f_{Nat^n} = g_{Nat^n}$, where Nat is the flat domain of natural numbers.

(c) We show that if $f_{Nat^n} = g_{Nat^n}$ then $f_{\vec{u}} = g_{\vec{u}}$ for every $\mathcal{C}^n$ object $\vec{u}$ which contains no trivial component.

(d) Finally, we show that if $f_{\vec{u}} = g_{\vec{u}}$ for every $\mathcal{C}^n$ object $\vec{u}$ containing no trivial component, then $f_{\vec{t}} = g_{\vec{t}}$ for all $\mathcal{C}^n$ objects $\vec{t}$ as well (including those containing trivial components).

Together these results imply that if $f_{\vec{u}} = g_{\vec{u}}$ for *any* $\mathcal{C}^n$ object $\vec{u}$ containing no trivial component, and if $f \sqsubseteq g$, then $f_{\vec{t}} = g_{\vec{t}}$ for *all* $\mathcal{C}^n$ objects $\vec{t}$.

Definition 6.
An object $\vec{u} = (u_1, .., u_n)$ of $\mathcal{C}^n$ *contains no trivial component* iff $u_k \ncong \mathbf{1}$ for all k. □

Definition 7.
For each type $u \ncong \mathbf{1}$, define

$$\begin{array}{lcll} \Delta_u : u \to \mathbf{2} & & & \\ \Delta_u\ x & = & \bot & \textbf{if } x = \bot \\ & = & \mathit{lift}\ \bot & \textbf{if } x \neq \bot \end{array}$$

□

Δ_u can be thought of as a definedness test. It is the unique strict and $\bot$-reflecting map from u to $\mathbf{2}$ and, as it is onto, it is epic. If $\vec{u} = (u_1, \ldots, u_n)$ is an object of $\mathcal{C}^n$ containing no trivial component, we write $\Delta_{\vec{u}}$ for the $\mathcal{C}^n_r$ morphism $(\Delta_{u_1}, \ldots, \Delta_{u_n}) : u \to \mathbf{2}^n$. Note that $\Delta_{\vec{u}}$ is epic.

Lemma 4. Let $f, g : F \to G$ be weakly polymorphic, and suppose that $f_{\vec{u}} = g_{\vec{u}}$ for some $\mathcal{C}^n$ object $\vec{u}$ containing no trivial component. Then $f_{\mathbf{2}^n} = g_{\mathbf{2}^n}$.

Proof.

$$\begin{array}{lcll} f_{\vec{u}} = g_{\vec{u}} & \Longrightarrow & G(\Delta_{\vec{u}}) \circ f_{\vec{u}} = G(\Delta_{\vec{u}}) \circ g_{\vec{u}} & \\ & \Longrightarrow & f_{\mathbf{2}^n} \circ F(\Delta_{\vec{u}}) = g_{\mathbf{2}^n} \circ F(\Delta_{\vec{u}}) & \text{[by weak polymorphism]} \\ & \Longrightarrow & f_{\mathbf{2}^n} = g_{\mathbf{2}^n} & [\Delta_{\vec{u}} \text{ is epic, so } F(\Delta_{\vec{u}}) \text{ is epic}] \end{array}$$

The second stage of our argument shows that if $f_{\mathbf{2}^n} = g_{\mathbf{2}^n}$ then $f_{\vec{v}} = g_{\vec{v}}$ for any tuple of flat domains $\vec{v}$, and in particular for Nat^n. We begin with a lemma.

Lemma 5. Let F and G be type functors, $\vec{v}$ be a tuple of flat domains $(v_1, .., v_n)$, and $h, k : F(\vec{v}) \to G(\vec{v})$. If $h \sqsubseteq k$ and $G(\Delta_{\vec{v}}) \circ h = G(\Delta_{\vec{v}}) \circ k$ then $h = k$.

Proof. By structural induction on G.

Case: $G = sel_i$

In this case, $h\ x, k\ x \in v_i$ for all $x \in F(\vec{v})$. Since v_i is a flat domain and $h\ x \sqsubseteq k\ x$, the fact that $\Delta_{v_i}\ (h\ x) = \Delta_{v_i}\ (k\ x)$ implies that $h\ x = k\ x$. As this holds for all $x \in F(\vec{v})$, we conclude that $h = k$.

Case: $G = \mathbf{1}$

Both h and k are arrows from a common source into the terminal object so they must be equal.

Case: $G = H \circ K,\ G =< H_1, \ldots, H_m >$

Immediate inductions.

Case: $G = H_\perp$

By assumption, $(H\ \Delta_{\vec{v}})_\perp \circ h = (H\ \Delta_{\vec{v}})_\perp \circ k$. Apply both sides to some value $x \in F(\vec{v})$. If $h\ x = \perp$, then because $(H\ \Delta_{\vec{v}})_\perp$ is strict and $\perp$-reflecting (as $\Delta_{\vec{v}}$ is strict and $\perp$-reflecting and H is a functor $H : \mathcal{C}_r^n \to \mathcal{C}_r$ by restriction) we conclude that $k\ x = \perp$ also.

Alternatively, suppose that $h\ x = \mathit{lift}\ y$ for some $y \in H(\vec{v})$. By the argument above, $k\ x = \mathit{lift}\ z$ for some $z \in H(\vec{v})$. Then

$$\mathit{lift}\ (\Delta_{\vec{v}}\ y) = (H\ \Delta_{\vec{v}})_\perp\ (h\ x) = (H\ \Delta_{\vec{v}})_\perp\ (k\ x) = \mathit{lift}\ (\Delta_{\vec{v}}\ z)$$

As *lift* is one-to-one, $H\ (\Delta_{\vec{v}}\ y) = H\ (\Delta_{\vec{v}}\ z)$. But $y \sqsubseteq z$ as $h \sqsubseteq k$, so by induction we conclude that $y = z$. Together, these two cases establish that $h = k$.

Case: $G = H \oplus K$

The proof follows the previous case except that in this case there are three situations to consider: either $h\ x = \perp$, or $h\ x = \mathit{inl}\ y$ for some $y \in H(t) \setminus \{\perp\}$, or $h\ x = \mathit{inl}\ z$ for some $z \in K(t) \setminus \{\perp\}$.

Case: $G = H \otimes K$

As before, if $h\ x = \perp$ then $k\ x = \perp$ also. Conversely, suppose that $h\ x = (a, b)$ where $a \neq \perp \neq b$. Then, similarly, $k\ x = (c, d)$ where $c \neq \perp \neq d$. Furthermore $(a, b) \sqsubseteq (c, d)$ as $h \sqsubseteq k$, so $a \sqsubseteq c$ and $b \sqsubseteq d$. By induction, therefore, $a = c$ and $b = d$ and hence $h = k$.

Case: $G = \mu H$

We reuse the diagram in Figure 2 with $C = F(\vec{v})$. Taking f to be $\Delta_{\vec{v}}$ we apply induction (for if $h \sqsubseteq k$ then $h' \sqsubseteq k'$) and conclude that $h' = k'$ (following the same argument used in Theorem 3). The same applies for every pair of arrows from C, and so $h = k$ by uniqueness.

Lemma 6. If $f, g : F \to G$ are weakly polymorphic with $f \sqsubseteq g$, and if $f_{\mathbf{2}^n} = g_{\mathbf{2}^n}$, then $f_{\vec{v}} = g_{\vec{v}}$ for any tuple of flat domains $\vec{v}$.

Proof.

$$\begin{array}{lll} f_{\mathbf{2}^n} = g_{\mathbf{2}^n} & \Longrightarrow & f_{\mathbf{2}^n} \circ F(\Delta_{\vec{v}}) = g_{\mathbf{2}^n} \circ F(\Delta_{\vec{v}}) \\ & \Longrightarrow & G(\Delta_{\vec{v}}) \circ f_{\vec{v}} = G(\Delta_{\vec{v}}) \circ g_{\vec{v}} \qquad \text{[weak polymorphism]} \\ & \Longrightarrow & f_{\vec{v}} = g_{\vec{v}} \qquad \text{[}f \sqsubseteq g\text{, previous lemma]} \end{array}$$

This lemma applies in particular when $\vec{v}$ is Nat^n. For the third stage of our argument we need to introduce a function from *Nat* to every type. Note that the finite elements of a Scott domain are countable. This is crucial to our proof, and we do not know whether the theorem holds in more general situations.

Definition 8.

For every type $u \ncong \mathbf{1}$, let $\epsilon_u : \mathit{Nat} \to u$ be a function that enumerates the finite elements of u, such that $\epsilon_u\ x = \perp$ if and only if $x = \perp$. □

It is immediately clear that for each type u, the function ϵ_u is monotonic, strict and $\perp$-reflecting. As *Nat* has no infinite directed sets, ϵ_u is also continuous and, by lemma

2, it is epic. If $\vec{u} = (u_1, \ldots, u_n)$ is an object of $\mathcal{C}^n$ containing no trivial component, we will write $\epsilon_{\vec{u}}$ for the (epic) $\mathcal{C}^n_r$ morphism $(\epsilon_{u_1}, \ldots, \epsilon_{u_n}) : Nat^n \to \vec{u}$

Lemma 7. Suppose that $f, g : F \to G$ are weakly polymorphic and that $f_{Nat^n} = g_{Nat^n}$. Then $f_{\vec{u}} = g_{\vec{u}}$ for every object $\vec{u}$ of $\mathcal{C}^n$ containing no trivial component.

Proof.

$$\begin{array}{lcll} f_{Nat^n} = g_{Nat^n} & \Longrightarrow & G(\epsilon_{\vec{u}}) \circ f_{Nat^n} = G(\epsilon_{\vec{u}}) \circ g_{Nat^n} & \\ & \Longrightarrow & f_{\vec{u}} \circ F(\epsilon_{\vec{u}}) = g_{\vec{u}} \circ F(\epsilon_{\vec{u}}) & \text{[weak polymorphism]} \\ & \Longrightarrow & f_{\vec{u}} = g_{\vec{u}} & [\epsilon_{\vec{u}} \text{ is epic, so } F(\epsilon_{\vec{u}}) \text{ is epic}] \end{array}$$

Finally, we must show that if $f_{\vec{u}} = g_{\vec{u}}$ for every $\vec{u}$ containing no trivial component, then it is also true for the rest.

Lemma 8. Let $f, g : F \to G$ be weakly polymorphic, and let $t = (t_1, \ldots, t_n)$ be any object of $\mathcal{C}^n$. Let $u = (u_1, \ldots, u_n)$ be the $\mathcal{C}^n$ object defined by

$$\begin{array}{rcll} u_i & = & t_i & \textbf{if } t_i \ncong \mathbf{1} \\ & = & \mathbf{2} & \textbf{if } t_i \cong \mathbf{1} \end{array}$$

Then $\vec{u}$ contains no trivial component, and if $f_{\vec{u}} = g_{\vec{u}}$ then $f_{\vec{t}} = g_{\vec{t}}$.

Proof. We define a $\mathcal{C}^n_r$ morphism $h = (h_1, \ldots, h_n) : \vec{t} \to u$ by

$$\begin{array}{rcll} h_i & = & ID & \textbf{if } t_i \ncong \mathbf{1} \\ & = & \lambda x.\bot_{\mathbf{2}} & \textbf{if } t_i \cong \mathbf{1} \end{array}$$

It is clearly $\bot$-reflecting and monic. Now,

$$\begin{array}{lcll} f_{\vec{u}} = g_{\vec{u}} & \Longrightarrow & f_{\vec{u}} \circ F(h) = g_{\vec{u}} \circ F(h) & \\ & \Longrightarrow & G(h) \circ f_{\vec{t}} = G(h) \circ g_{\vec{t}} & \text{[weak polymorphism]} \\ & \Longrightarrow & f_{\vec{t}} = g_{\vec{t}} & [h \text{ is monic, so } G(h) \text{ is monic}] \end{array}$$

Combining these results gives

Theorem 9. Let $f, g : F \to G$ be weakly polymorphic, with $f \sqsubseteq g$. If $f_{\vec{u}} = g_{\vec{u}}$ for any $\mathcal{C}^n$ object $\vec{u}$ containing no trivial component, then $f_{\vec{t}} = g_{\vec{t}}$ for all $\mathcal{C}^n$ objects $\vec{t}$.

Note that the restrictions that u should contain no trivial component, and that $f \sqsubseteq g$, are necessary. To see the first, consider the two polymorphic functions $STR_t : t_\bot \to t_\bot$ and $B_t : t_\bot \to t_\bot$ defined by $B_t\ x = \bot$ (B is just a less polymorphic version of BOT). Then $B \sqsubseteq STR$ and $B_{\mathbf{1}} = STR_{\mathbf{1}}$, but it is not generally true that $B_t = STR_t$. For the second, consider the selection functions over strict-pairs $s\_fst, s\_snd : \forall t\ .\ t \otimes t \to t$. These are both weakly polymorphic, and yet their $\mathbf{2}$ instances are equal, i.e. $s\_fst_{\mathbf{2}} = s\_snd_{\mathbf{2}}$ even though $s\_fst \neq s\_snd$ in general.

There is an interesting corollary to this theorem.

Corollary 10. ID and BOT are the only weakly polymorphic projections $\forall t\ .\ t \to t$; and ID, STR, ABS and BOT are the only weakly polymorphic projections $\forall t\ .\ t_\bot \to t_\bot$

Sketch Proof. Any projection $\alpha : \forall t\ .\ t \to t$ will satisfy $BOT \sqsubseteq \alpha \sqsubseteq ID$, by definition of projections. However, the only projections on the two-point domain are $BOT_{\mathbf{2}}$ and $ID_{\mathbf{2}}$, and so by the theorem above, α equals one of these.

The second part is proved similarly by noting that any projection $\alpha : \forall t\ .\ t_\bot \to t_\bot$ will satisfy either $BOT \sqsubseteq \alpha \sqsubseteq STR$ or $ABS \sqsubseteq \alpha \sqsubseteq ID$.

4.2. Strictness Analysis

Now let us apply this result to projection-based strictness analysis introduced in Section 2. If $f : F \to G$ is weakly polymorphic, and $\alpha : F \to F$, and $\beta : G \to G$ are weakly polymorphic projections, and if for all vectors of types $\vec{t}$, $f_{\vec{t}} : \beta_{\vec{t}} \Rightarrow \alpha_{\vec{t}}$, then we write $f : \beta \Rightarrow \alpha$.

Theorem 11. Let $f : F \to G$ be weakly polymorphic and $\alpha : F \to F$, and $\beta : G \to G$ be weakly polymorphic projections. If $f_{\vec{u}} : \beta_{\vec{u}} \Rightarrow \alpha_{\vec{u}}$ for any $\mathcal{C}^n$ object $\vec{u}$ containing no trivial component, then $f : \beta \Rightarrow \alpha$.

Proof. Consider the functions $\beta \circ f$ and $\beta \circ f \circ \alpha$. They are both weakly polymorphic and, as $\alpha \sqsubseteq ID$, $\beta \circ f \circ \alpha \sqsubseteq \beta \circ f$. Now, $f_{\vec{u}} : \beta_{\vec{u}} \Rightarrow \alpha_{\vec{u}}$ so $\beta_{\vec{u}} \circ f_{\vec{u}} \circ \alpha_{\vec{u}} = \beta_{\vec{u}} \circ f_{\vec{u}}$. By Theorem 9, $\beta_{\vec{t}} \circ f_{\vec{t}} \circ \alpha_{\vec{t}} = \beta_{\vec{t}} \circ f_{\vec{t}}$ for all $\mathcal{C}^n$ objects t, that is, $f : \beta \Rightarrow \alpha$.

Notice, in particular, that $\mathbf{2}^n$ is a $\mathcal{C}^n$ object containing no trivial component. Since we can use the methods of [33] to find a good $\alpha_{\mathbf{2}^n}$ such that $f_{\mathbf{2}^n} : \beta_{\mathbf{2}^n} \Rightarrow \alpha_{\mathbf{2}^n}$ it follows that we can find a good polymorphic α, given polymorphic f and β, such that $f : \beta \Rightarrow \alpha$.

Of course, when we call a polymorphic function we call a particular instance. In general, the context in which we call it may be any projection over the instance type, not necessarily an instance of a polymorphic projection. We need a way to apply *polymorphic* information about the function in such cases. The following theorem enables us to do this. See Section 5.3 for an example of its use.

Theorem 12. Let $f : F \to G$ be a strongly polymorphic function, and let $\alpha : F \to F$ and $\beta : G \to G$ be weakly polymorphic projections such that $f : \beta \Rightarrow \alpha$. Then, for any instance $\vec{t}$ and projection $\gamma : \vec{t} \to \vec{t}$,

$$f_{\vec{t}} : \beta_{\vec{t}} \circ G(\gamma) \Rightarrow \alpha_{\vec{t}} \circ F(\gamma)$$

Proof. The proof is most conveniently performed using the alternative formulation of the safety condition, namely, $\beta \circ f \sqsubseteq f \circ \alpha$[§].

$$\begin{array}{rcll} f : \beta \Rightarrow \alpha & \Longrightarrow & \beta_{\vec{t}} \circ f_{\vec{t}} \sqsubseteq f_{\vec{t}} \circ \alpha_{\vec{t}} & \\ & \Longrightarrow & \beta_{\vec{t}} \circ f_{\vec{t}} \circ F(\gamma) \sqsubseteq f_{\vec{t}} \circ \alpha_{\vec{t}} \circ F(\gamma) & \\ & \Longrightarrow & \beta_{\vec{t}} \circ G(\gamma) \circ f_{\vec{t}} \sqsubseteq f_{\vec{t}} \circ \alpha_{\vec{t}} \circ F(\gamma) & [f \text{ is strongly polymorphic}] \\ & \Longrightarrow & f_{\vec{t}} : \beta_{\vec{t}} \circ G(\gamma) \Rightarrow \alpha_{\vec{t}} \circ F(\gamma) & \end{array}$$

Note that f's strong polymorphism is necessary since γ might not be $\perp$-reflecting. Note also, that in using the notation $f_{\vec{t}} : \beta_{\vec{t}} \circ G(\gamma) \Rightarrow \alpha_{\vec{t}} \circ F(\gamma)$ to describe the conclusion, the theorem tacitly assumed the following lemma (whose rather technical proof may be skipped).

Lemma 13. If $\alpha : F \to F$ is a weakly polymorphic projection, and $\gamma : \vec{t} \to \vec{t}$ is a projection (for some $\mathcal{C}^n$ object $\vec{t}$), then $\alpha_t \circ F(\gamma)$ is a projection.

Proof. It is easy to see that $\alpha \circ F(\gamma) \sqsubseteq ID$ for both $\alpha \sqsubseteq ID$ and $F(\gamma) \sqsubseteq ID$. However, demonstrating idempotence is surprisingly tricky as α is only weakly polymorphic and so $\alpha \circ F(\gamma) \neq F(\gamma) \circ \alpha$ in general (as γ is not necessarily $\perp$-reflecting).

§ A proof that this is equivalent to $\beta \circ f = \beta \circ f \circ \alpha$ may be found in [33].

To simplify the proof we assume that $\vec{t}$ is a single domain t. Any projection is $\perp$-reflecting on its range, but it is not always the case that the range of a projection is a domain (as it may not be algebraic). We will construct a projection whose range is a sub-domain of t and over which γ is $\perp$-reflecting. Let

$$\begin{array}{rcll} \gamma'\ x & = & \perp & \textbf{if } \gamma\ x = \perp \\ & = & x & \textbf{otherwise} \end{array}$$

Note that $\gamma' \circ \gamma = \gamma$. The range of γ' is a domain (algebraicity is preserved as γ' maps finite elements to finite elements, the other properties are straightforward), so we may express γ' as an embedding/projection pair: if t' is a domain isomorphic to the range of γ', then there exist functions $e : t' \to t$ and $p : t \to t'$ such that $e \circ p = \gamma'$ and $p \circ e = ID_{t'}$. As e is an embedding it is monic and, therefore, $\perp$-reflecting. What is more, as both γ and γ' (and hence p also) map exactly the same elements to $\perp$, both $\gamma \circ e$ and $p \circ e$ map exactly the same elements to $\perp$. However, as $p \circ e = ID_{t'}$ it is $\perp$-reflecting and, hence, so is $\gamma \circ e$. Using these facts,

$$\begin{array}{lll} \alpha_t \circ F(\gamma) \circ \alpha_t \circ F(\gamma) & & \\ \quad = \quad \alpha_t \circ F(\gamma) \circ \alpha_t \circ F(\gamma' \circ \gamma) & & [\gamma' \circ \gamma = \gamma] \\ \quad = \quad \alpha_t \circ F(\gamma) \circ \alpha_t \circ F(e) \circ F(p \circ \gamma) & & [\gamma' = e \circ p,\ F \text{ a functor}] \\ \quad = \quad \alpha_t \circ F(\gamma \circ e) \circ \alpha_{t'} \circ F(p \circ \gamma) & & [e \perp\text{-reflecting},\ \alpha \text{ weakly polymorphic}] \\ \quad = \quad \alpha_t \circ \alpha_t \circ F(\gamma \circ e) \circ F(p \circ \gamma) & & [\gamma \circ e \perp\text{-reflecting},\ \alpha \text{ weakly polymorphic}] \\ \quad = \quad \alpha_t \circ F(\gamma \circ \gamma' \circ \gamma) & & [\alpha \text{ idempotent},\ e \circ p = \gamma'] \\ \quad = \quad \alpha_t \circ F(\gamma) & & [\gamma' \circ \gamma = \gamma,\ \gamma \text{ idempotent}] \end{array}$$

And so $\alpha_t \circ F(\gamma)$ is idempotent as required.

5. Putting it into Practice

In this less formal section we offer suggestions for implementing the theory of the preceding sections in practice, and we provide an example of the sort of analysis that results.

5.1. Finite Domains of Projections

As we noted at the end of Section 2, the standard methods of projection analysis use finite lattices of projections. We show how to produce such lattices whose size and structure is based on the *textual definitions* of the types themselves. We call these specially chosen projections *contexts*¶. Note that many choices of context domain are possible: we are making a particular one.

Definition 9.
The projection $\alpha : t \to t$ is a *context* for a type t if α **cxt** t can be inferred using the rules

: ¶ While the definition of contexts is syntactic, we deliberately de-emphasise the distinction between the syntax and semantics and move between them at will. This simplifies the account, and is sufficient for our purposes here. A fully rigorous treatment would make precise where the distinction lies.

below.

$$ID_{\mathbf{1}}\ \mathbf{cxt}\ \mathbf{1}$$

$$\frac{\alpha\ \mathbf{cxt}\ t}{\alpha_\perp\ \mathbf{cxt}\ t_\perp} \qquad \frac{\alpha\ \mathbf{cxt}\ t}{STR \circ \alpha_\perp\ \mathbf{cxt}\ t_\perp}$$

$$\frac{\alpha\ \mathbf{cxt}\ s \qquad \beta\ \mathbf{cxt}\ t}{\alpha \oplus \beta\ \mathbf{cxt}\ s \oplus t}$$

$$\frac{\alpha\ \mathbf{cxt}\ s \qquad \beta\ \mathbf{cxt}\ t}{\alpha \otimes \beta\ \mathbf{cxt}\ s \otimes t}$$

$$\frac{F(\alpha)\ \mathbf{cxt}\ T(t) \qquad [\alpha\ \mathbf{cxt}\ t]}{\mu\alpha.F(\alpha)\ \mathbf{cxt}\ \mu t.T(t)}$$

□

Observe that in the rule for μ, we restrict our attention to projections on recursive types that treat each level in the same way.

Using the rules above, we may infer that (projections equal to) ID_t and BOT_t are contexts for all types t. In fact, the set of contexts over any type forms a lattice. We can therefore abstract the domain of projections over a type into the domain of contexts by mapping each projection to the least context greater than it. This mapping is continuous, and so we can use it to induce a monotonic function on contexts from any continuous function on projections—we just apply the function to contexts viewed as projections, and then map the result back into the lattice of contexts as just described. A practical strictness analyser will of course compute with these induced operations on contexts. Since the induced operations always overestimate the true result it is safe to do so. This is a simple application of the techniques of abstract interpretation [9].

Note that the domain of contexts is not a subdomain of the domain of projections, because the least upper bound operation differs. In the domain of contexts for $t_\perp \otimes t_\perp$, we have

$$(STR \otimes ID) \sqcup (ID \otimes STR) = ID \otimes ID$$

but in the domain of projections this is not so: the left hand side maps the pair (*lift* $\perp$, *lift* $\perp$) to $\perp$, while the right hand side leaves it unchanged.

5.2. Examples of Contexts

Let us calculate the contexts for a number of interesting types. First of all consider the type $\mathbf{2}$ $(= \mathbf{1}_\perp)$. Since there is only one context for $\mathbf{1}$, we can infer that there are only two contexts for $\mathbf{2}$:

$$STR_{\mathbf{1}} \circ (ID_{\mathbf{1}})_\perp$$
$$(ID_{\mathbf{1}})_\perp$$

The former equals $BOT_{\mathbf{2}}$, and the latter equals $ID_{\mathbf{2}}$. Now consider the type Nat, which can be defined by $Nat = \mu n \ . \ \mathbf{2} \oplus n$. Given that α **cxt** n, we can infer both $(BOT_{\mathbf{2}} \oplus \alpha)$ **cxt** $(\mathbf{2} \oplus n)$ and $(ID_{\mathbf{2}} \oplus \alpha)$ **cxt** $(\mathbf{2} \oplus n)$, so there are two contexts for Nat:

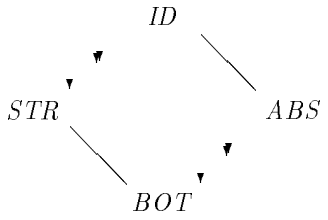
$$\mu\alpha \ . \ BOT_{\mathbf{2}} \oplus \alpha$$
$$\mu\alpha \ . \ ID_{\mathbf{2}} \oplus \alpha$$

These are equal to BOT_{Nat} and ID_{Nat} respectively. So as we would expect, the contexts chosen for Nat ignore the value of the natural, and indeed are just the two contexts usually chosen for every non-trivial base type.

More interesting are the contexts for $Nat_\perp$ i.e. closures of type Nat. Since there are two contexts for Nat, there are four for $Nat_\perp$:

$$(BOT_{Nat})_\perp$$
$$(ID_{Nat})_\perp$$
$$STR \circ (BOT_{Nat})_\perp$$
$$STR \circ (ID_{Nat})_\perp$$

$(BOT_{Nat})_\perp$ does not "evaluate" the closure, but discards its contents—this context was called ABS in [33] and corresponds intuitively to ignoring the closure altogether. $(ID_{Nat})_\perp$ is the identity on closures, and $STR \circ (BOT_{Nat})_\perp$ is the bottom projection. $STR \circ (ID_{Nat})_\perp$ $(= STR_{Nat_\perp})$ "evaluates" the closure. Thus these four contexts form the basic four-point domain of [33]:



Finally let us consider lazy lists. We must describe the list type using our chosen type formers: it is

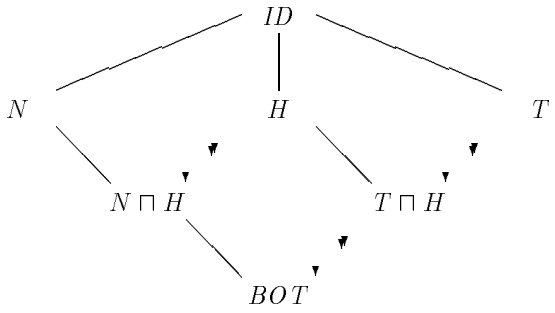
$$List(t) = \mu l \ . \ \mathbf{2} \oplus (t_\perp \otimes l_\perp)$$

This is isomorphic to the more familiar $List(t) = \mu l \ . \ \mathbf{1} + (t \times l)$ which uses separated sum and ordinary product, but note that our formulation uses lifting exactly where an implementation uses closures. As a result the contexts selected by our rules carry information directly useful to an implementation.

Since there are two contexts for $\mathbf{2}$, and two contexts for $t_\perp$ and $l_\perp$ corresponding to each context for t and l, our rules generate eight syntactically different list contexts to correspond to each element context. The contexts can be expressed as greatest lower bounds of just three of their number:

$$N = \mu\alpha \ . \ BOT_{\mathbf{2}} \oplus (ID_\perp \otimes \alpha_\perp)$$
$$H = \mu\alpha \ . \ ID_{\mathbf{2}} \oplus (STR \otimes \alpha_\perp)$$
$$T = \mu\alpha \ . \ ID_{\mathbf{2}} \oplus (ID_\perp \otimes (STR \circ \alpha_\perp))$$

N is the projection that maps a *nil* at the end of a finite list to $\perp$. An N-strict function (for example *head*) gives the same result for *nil* or $\perp$. The information given by this projection has never been used in improving language implementations, so its practical importance is hard to gauge. H and T, however, capture the intuitive ideas of "head-strictness" and "tail-strictness", and were used for that purpose in [33]. Now we can express the contexts for $List(t)$ as the composition of any of the polymorphic list contexts below, with $List(\gamma)$, where γ is any context for t.



Note that since $T \sqcap N = BOT$ there are only seven different contexts (recall from [33] that the $\sqcap$ of two projections is the greatest *projection* beneath the two, not merely the greatest function).

When performing strictness analysis on lists, we will actually be analysing *closures* of lists, that is, lifted lists. Each of the contexts above may then be composed with any of the four basic contexts ID, STR, ABS, and BOT. The contexts ID, STR, $STR \circ T_\perp$ and $STR \circ (T \sqcap H)_\perp$ correspond to Burn's evaluation transformers $\mathcal{E}_0$, $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{E}_3$ [8]. The others cannot be expressed in Burn's system, not being so-called *smash* projections [7]. Burn's evaluator H_B is not represented here as it treats the first element of the list differently from the others.

5.3. Factorising Projections

Notice that contexts for $List(t)$ can be "factored" into the form $\alpha \circ List(\gamma)$ where α is independent of the element type and γ is a context for t. Recall also that in Theorem 12 we showed that such a factorisation can help us to apply polymorphic knowledge about a function in the analysis of a particular instance.

More generally, suppose we are analysing a program and we come across the t instance, say, of a polymorphic function $f : F \to G$. We will have a context $\delta : G(t) \to G(t)$ describing the demand of f_t and wish to find another projection $\epsilon : F(t) \to F(t)$ describing the demand f_t places on its argument. If we can express δ as a composition of the form $\delta = \alpha_t \circ G(\gamma)$ where $\alpha : G \to G$ is polymorphic, then all we need to show is that $f : \alpha \Rightarrow \beta$ for some polymorphic projection $\beta : F \to F$ to enable us to deduce that $f_t : \delta \Rightarrow (\beta_t \circ F\ \gamma)$.

A practical strictness analyser based on this method would analyse a polymorphic function function as follows: on encountering its definition, enumerate all the polymor-

phic contexts for its result type, and analyse the function for each of them using the monomorphic techniques of [33] at any convenient instance, say **2**. Because of the theorems in the first part of the paper the results must hold polymorphically, that is, at every instance. Having produced a table of these results, the analysis of any instance of the function requires only the factorisation described in the previous paragraph.

The advantage of this approach is twofold. First, in general there are fewer polymorphic projections than there are monomorphic. As a consequence, calculating a safe β will take place within smaller lattices if done polymorphically, and so will be more efficient. Secondly, the polymorphic fact that $f : \beta \Rightarrow \alpha$ may be saved and reused, even for a whole series of different instances of f.

5.4. An Example of Analysis

To make this more concrete, we discuss the analysis of a simple polymorphic function—the well-known function *append*. As described in Section 2, we will analyse the lifted version[||]. This has the type

$$append_\perp : \forall t \ . \ (List(t) \times List(t))_\perp \to List(t)_\perp$$

or, equivalently,

$$append_\perp : \forall t \ . \ List(t)_\perp \otimes List(t)_\perp \to List(t)_\perp$$

Note that its arguments are passed lazily, that is, as closures. Using the techniques of monomorphic strictness analysis at the **2** instance, we may, for example, discover that

$$(append_{\mathbf{2}})_\perp : STR_{List\ (\mathbf{2})} \circ (T_{\mathbf{2}})_\perp \Rightarrow (STR_{List(\mathbf{2})} \circ (T_{\mathbf{2}})_\perp) \otimes (STR_{List(\mathbf{2})} \circ (T_{\mathbf{2}})_\perp)$$

and Theorem 11 then allows us to conclude the polymorphic property

$$append_\perp : STR \circ T_\perp \Rightarrow (STR \circ T_\perp) \otimes (STR \circ T_\perp)$$

A similar result that the analysis may derive is that,

$$append_\perp : STR \circ H_\perp \Rightarrow (STR \circ H_\perp) \otimes H_\perp$$

The first property states that if the result of $append_\perp$ is in a strict and tail-strict context, then both of the arguments are also. The second property states that if the result of $append_\perp$ is in a strict, head-strict context, then so is its first argument, but its second is in a lazy head-strict context. That is, the second argument may not be required, but if it is, then it will be head-strict. These and similar polymorphic properties are saved for later use. Now, to analyse a call of a particular instance of *append*, the t instance say, in a context $\delta : List(t) \to List(t)$, we first factorise δ into the form

$$\delta = \alpha_t \circ List(\gamma)$$

[||] The details of this lifting transformation are presented in [22] but need not concern us here.

where α is a polymorphic projection over $List$, and γ is a projection over t. As α is polymorphic, there must be a saved fact

$$append : \alpha \Rightarrow \beta$$

for some polymorphic β. By Theorem 12 we can conclude

$$append_t : \alpha_t \circ List(\gamma) \Rightarrow \beta_t \circ List(\gamma)$$

and so

$$append_t : \delta \Rightarrow \beta_t \circ List(\gamma)$$

This completes the analysis of the call.

To take a concrete example, suppose the result of *append* is passed to a function such as $(sum \circ map\ fst)$ (which adds together the first components of the list). This type of this function when lifted is

$$(sum \circ map\ fst)_\perp : \forall t\ .\ List(Int \times t)_\perp \to Int_\perp$$

Further suppose that the result of the sum is demanded strictly. The context for the result of $append_\perp$ is therefore $STR \circ T_\perp \circ List(STR_{Int} \otimes ABS)_\perp$. Using the fact above and Theorem 12, we can infer at once that, for all t,

$$append_{Int \times t} : STR \circ T_\perp \circ List(STR_{Int} \otimes ABS)_\perp \Rightarrow \\ (STR \circ T_\perp \circ List(STR_{Int} \otimes ABS)_\perp) \otimes (STR \circ T_\perp \circ List(STR_{Int} \otimes ABS)_\perp)$$

That is, the spines of both arguments are strictly evaluated together with the integer first components of the list elements, while the second components of each are discarded.

As another example, suppose that the result of an append is passed to the function $(sum \circ take\ 10)_\perp$ whose result is again demanded strictly (the function *take 10* takes the first 10 elements of its list argument). This gives a context of $STR \circ H_\perp \circ List(STR_{Int})_\perp$ for the result of *append*. Again, using the fact above and Theorem 12, we can infer at once that

$$append_{Int \times t} : STR \circ H_\perp \circ List(STR_{Int})_\perp \Rightarrow \\ (STR \circ H_\perp \circ List(STR_{Int})_\perp) \otimes (H_\perp \circ List(STR_{Int})_\perp)$$

Thus, *append*'s first argument is required strictly and head-strictly, while its second is only required head-strictly.

5.5. Approximate Factorisations

Unfortunately, there is no hope that every projection $\delta : G(t) \to G(t)$ may be factorised into the form $\alpha \circ G\ \gamma$. The best we can hope for, therefore, is the existence of an optimal approximate factorisation, i.e. a choice of α and γ such that $\delta \sqsubseteq \alpha \circ G\ \gamma$ where α and γ are least. It is not clear whether such an optimal factorisation exists in general: our strenuous attempts to prove that it does have all failed, as have our attempts to find a counter example. In the case of contexts, however, it is rare that any approximation needs to be introduced at all. For example, as we noted earlier, all contexts over list domains may be appropriately factorised. In practice, a strictness analyser must incorporate

a factorisation algorithm operating on the syntax of contexts, and producing results expressible in that syntax. In general, this may not represent the optimal factorisation even if it exists.

For some types, however, approximation must be introduced, and so the method of Theorem 12 gives poorer results than analysing each instance separately. For example, consider

$$\begin{aligned} &dup_t : t \to t \times t \\ &dup_t\ x = (x, x) \end{aligned}$$

Analysis shows that $dup_\perp : STR \otimes STR \Rightarrow STR$ and so using Theorem 12 and factorisation we conclude that

$$dup_\perp : (STR \circ \alpha_\perp) \otimes (STR \circ \beta_\perp) \Rightarrow STR \otimes (\alpha \sqcup \beta)_\perp$$

The use of $\sqcup$ arises because we need a single safe approximation to both α and β. But note that if either $\alpha\ x = \perp$ or $\beta\ x = \perp$, then $dup_\perp(lift\ x)$ is mapped to $(\perp, \perp)$ by the context on the left, because the product is strict. Wadler and Hughes' earlier monomorphic techniques take advantage of this to derive

$$dup_\perp : (STR \circ \alpha_\perp) \otimes (STR \circ \beta_\perp) \Rightarrow STR \circ (\alpha \& \beta)_\perp$$

where

$$\begin{array}{lcll} (\alpha \& \beta)\ x & = & \perp & \textbf{if } \alpha\ x = \perp \textbf{ or } \beta\ x = \perp \\ & = & \alpha\ x \sqcup \beta\ x & \textbf{otherwise} \end{array}$$

It is not clear how to derive the same information from a polymorphic analysis. Future work will investigate factorisation taking into account the function to be analysed, since although it is not true that

$$(STR \circ \alpha_\perp) \otimes (STR \circ \beta_\perp) \sqsubseteq (STR \otimes STR) \circ ((\alpha \& \beta)_\perp \otimes (\alpha \& \beta)_\perp)$$

it is the case that

$$(STR \circ \alpha_\perp) \otimes (STR \circ \beta_\perp) \circ dup_\perp \sqsubseteq (STR \otimes STR) \circ ((\alpha \& \beta)_\perp \otimes (\alpha \& \beta)_\perp) \circ dup_\perp$$

6. Related Work and Conclusions

Strictness analysis of functional languages has attracted a great deal of interest over the past decade, beginning with Mycroft's seminal paper [27]. Mycroft introduced the idea of using abstract interpretation for strictness analysis, and although his method was limited to first order functions operating on non-lazy data-structures it is the foundation for much later work. Mycroft worked with an untyped programming language, and this theme has been continued by Hudak and Young, who discovered a technique for analysing higher-order functions in this setting [15]. Others, however, have concentrated on exploiting a type system. Burn, Hankin and Abramsky (BHA) developed an abstract interpretation of higher-order typed programs [6], which Wadler extended to handle lazy lists [34]. A common feature of this work is its emphasis on working in finite domains, in which fixed points can be calculated directly by iteration. Clack and Peyton-Jones invented a technique for finding such fixed points efficiently [10].

The foregoing are *forwards* analysis methods; backwards analysis is interesting because of its apparent ability to handle lazy data-structures well. Hughes developed an ad-hoc analyser for first-order untyped languages that relied on algebraic simplification on infinite domains to analyse recursive functions over lazy data-structures [17]. More recently Hall and Wise have integrated a similar analyser with a program transformer [14]—they also use algebraic simplification in infinite domains. Theories of backwards strictness analysis were developed based on continuations [18], Scott-open sets [11], and projections [33]. The projection-based theory has inspired similar approaches to binding-time analysis [24, 25] (a forwards analysis) and complexity analysis [35], [5]. Recently Hunt has invented an analysis based on PERs which generalises both BHA and projection-based analyses [23].

Backwards analysis, like forwards analysis, can take advantage of type information. An early backwards analyser that did so was developed by Wray: it handled a powerful polymorphic type system with higher-order functions, and moreover ran in an insignificant fraction of the total compile-time [37]. However, it was not proved correct, and was unable to analyse functions on lazy data-structures usefully. Wadler and Hughes used a monomorphic type system for their work on projection based analysis, and Hughes has continued to do so in more recent work on analysing higher-order functions [19] and performing sharing and life-time analysis backwards [21].

Monomorphically typed languages have proved amenable to compile-time analysis, whether forwards or backwards. Their advantage is that infinitely many different pieces of possible information may be divided into many finite sets, one per type. Thus an analyser gains the advantage of being able to represent infinitely many different possibilities, without the cost of needing to find one solution from an infinite set in any particular case. The disadvantage of basing an analyser on a monomorphic type system is that most practical functional languages are polymorphic.

Abramsky pointed the way to a solution of this problem by treating a polymorphic function as an infinite collection of monomorphic instances, and showing that some of the results of strictness analysis are common to all [1]. Thus only one instance need be analysed using monomorphic techniques and the results can be applied to all. This idea is at the heart of our own work. However, Abramsky used a syntactic characterisation of polymorphism—the type inference rules—and therefore had to prove his result by structural induction over terms. Hughes used the semantic characterisation of first order polymorphic functions as natural transformations to strengthen Abramsky's result in the first-order case, showing that an approximation to the abstract function of any instance can be calculated from that of the simplest [20]. Abramsky and Jensen have shown that higher-order functions are lax-natural transformations in a more complex category, and thereby proved a stronger version of Abramsky's original result [2]. In addition, Abramsky and Jensen have shown that our Theorem 9 does not hold for higher-order functions: polymorphic higher-order functions are *not* characterised by *any* finite instance.

In the future, we hope to generalise the result of [20] to the abstract interpretation of higher-order functions. This is difficult because the function-space type former is not a covariant functor. Possible approaches are to use dinatural transformations [12], or to make the function type covariant by working in a more complex category as both

Abramsky and Jensen, and Wadler [36] did. Unfortunately this can be done in several different ways, and it is not yet clear which way is best suited to this particular problem. An alternative is suggested by Baraki [4], where sets of closure/embedding pairs are used to relate distinct instances. In the longer term it would be interesting to make a connection with operational semantics, so as to attack a wider class of analysis problems.

The particular result in this paper is a new strictness analysis method for polymorphic first-order functions. A major advantage over previous methods is the ease with which polymorphic information can be applied to the call of a particular instance: neither Hughes' nor Abramsky's previous work can do so as smoothly as Theorem 12.

Our more general thesis is that the categorical view is a potent tool for solving practical problems involving polymorphic functions.

7. Acknowledgements

Thanks are due to Phil Wadler for his inspiration, and to Alan Mycroft for his many excellent suggestions for improving the presentation. This work was carried out under a grant from the UK Science and Engineering Research Council, and under ESPRIT Basic Research Action *Semantique*.

REFERENCES

1 S. Abramsky, *Strictness Analysis and Polymorphic Invariance*, Workshop on Programs as Data Objects, Copenhagen, S-V LNCS 217, 1985.

2 S. Abramsky and T.P. Jensen, *A Relational Approach to Strictness Analysis for Higher-Order Polymorphic Functions*, in POPL 91, Orlando, Florida, 1991.

3 S. Abramsky and C. L. Hankin (eds.) *Abstract Interpretation of Declarative Languages*, Ellis-Horwood, 1987.

4 G. Baraki. *A Note on Abstract Interpretation of Polymorphic Functions*, in FPCA 91, Cambridge MA, S-V LNCS 523, 1991.

5 B. Bjerner, S. Holmstrøm, *A Compositional Approach to Time Analysis of First Order Lazy Functional Programs*, in IFIP Functional Programming Languages and Computer Architecture, London, S-V LNCS, 1989.

6 G. L. Burn, C. L. Hankin, S. Abramsky, *The Theory of Strictness Analysis for Higher-order Functions*, Workshop on Programs as Data Objects, Copenhagen, S-V LNCS 217, 1985.

7 G. Burn. *A Relation between Abstract Interpretation and Projection Analysis*, 17th POPL, San-Francisco, 1990.

8 G. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*, Pitman, 1991.

9 P. Cousot and R. Cousot. *Abstract interpretation: a unified lattice-theoretic model for static analysis of programs by construction or approximation of fixpoints.* Proc 4th POPL, 1977.

10 C. Clack and S. L. Peyton-Jones, *Strictness Analysis: a Practical Approach*, in IFIP Conference on Functional Programming Languages and Computer Architecture, Nancy, France, S-V LNCS 201, 1985.

11 P. Dybjer, *Inverse Image Analysis Generalises Strictness Analysis*, J. Inf. and Comp. Vol 90 (2), Feb 91.

12 P.J. Freyd, J.Y. Girard, A. Scedrov, and P.J. Scott. *Semantic Parametricity in Polymorphic Lambda Calculus.* In 3^{rd} *Annual Symposium on Logic in Computer Science*, Edinburgh, Scotland, 1988.

13 B. Goldberg, *Detecting Sharing of Partial Applications in Functional Programs*, in IFIP Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, S-V LNCS 274, 1987.

14 C. Hall and D. S. Wise, *Compiling Strictness into Streams*, in ACM Symposium on Principles of Programming Languages, 1987.

15 P. Hudak and J. Young, *Higher-order Strictness Analysis for Untyped Lambda Calculus*, in ACM Symposium on Principles of Programming Languages, 1986.

16 P. Hudak, *Abstract reference counting*, in [3]

17 J. Hughes, *Strictness Detection in Non-Flat Domains*, Workshop on Programs as Data Objects, Copenhagen, S-V 217, 1985.

18 J. Hughes, *Analysing Strictness by Abstract Interpretation of Continuations*, in [3]

19 J. Hughes, *Backwards Analysis of Functional Programs*, IFIP Workshop on Partial Evaluation and Mixed Computation, Bjørner, Ershov and Jones (eds.), North-Holland, 1987.

20 J. Hughes, *Abstract Interpretation of First-order Polymorphic Functions*, Proc. Aspenas Workshop on Graph Reduction, University of Gothenburg, 1988.

21 J. Hughes, *Compile-time Analysis of Functional Programs*, in *Research Topics in Functional Programming*, ed. D.A. Turner, University of Texas at Austin, Year of Programming Series, Adison-Wesley, 1990.

22 J. Hughes and J. Launchbury, *Towards Relating Forwards and Backwards Analyses*, Proc Glasgow workshop on Functional Programming, Ullapool, S-V WIC, 1991.

23 S. Hunt, *PERs generalise Projections for Strictness Analysis*, Proc Glasgow workshop on Functional Programming, Ullapool, S-V WIC, 1991.

24 J. Launchbury, *Projections for Specialisation*, IFIP Workshop on Partial Evaluation and Mixed Computation, Bjørner, Ershov and Jones (eds.), North-Holland, 1987.

25 J. Launchbury, *Projection Factorisations in Partial Evaluation*, Ph.D. thesis, Glasgow University, 1989. *Distinguished Dissertation in Computer Science*, Vol 1, CUP, 1991.

26 C. Martin and C. Hankin, *Finding Fixed Points in Finite Lattices*, IFIP Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, S-V LNCS 274, 1987.

27 A. Mycroft, *The Theory and Practice of Transforming Call-by-Need into Call-by-Value*, Proc. International Symposium on Programming, S-V LNCS 83, 1980.

28 J.C. Reynolds. *Towards a Theory of Type Structure*. Proc. *Colloque sur la Programmation*, ed B. Robinet, S-V LNCS 19, 1974.

29 D. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn and Bacon, INC., 1986.

30 D. Scott and C. Gunter, *Semantic Domains*, Handbook in Theoretical Computer Science, North Holland, 1992.

31 M. Sheeran, *Categories for the Working Hardware Designer*, in Proc. Int. Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects, eds. M. Leeser and G. Brown, S-V LNCS 408, 1990.

32 M. Smith and G. Plotkin, *The Category-Theoretic Solution of Recursive Domain Equations*, SIAM J. COMPUT, Vol 11, No 4, pp 761-783, Nov 1982.

33 P. Wadler and J. Hughes, *Projections for Strictness Analysis*, IFIP Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, S-V LNCS 274, 1987.

34 P. Wadler, *Strictness Analysis on Non-flat Domains (by Abstract Interpretation over Finite Domains)*, in [3]

35 P. Wadler, *Strictness Analysis Aids Time Analysis*, in ACM Symposium on Principles of Programming Languages, 1988.

36 P. Wadler, *Theorems for Free!*, in ACM conference on Functional Programming Languages and Computer Architecture, London, ACM Press/Adison-Wesley, 1989.

37 S. C. Wray, *Implementation and Programming Techniques for Functional Languages*, Ph.D. thesis, University of Cambridge, 1986.