A Relational Approach to Strictness Analysis for Higher-Order Polymorphic Functions*

Samson Abramsky Thomas P. Jensen†

Imperial College‡

Abstract

This paper defines the categorical notions of relators and transformations and shows that these concepts enable us to give a semantics for polymorphic, higher order functional programs. We demonstrate the pertinence of this semantics to the analysis of polymorphic programs by proving that strictness analysis is a polymorphic invariant.

1 Introduction

Recently, there has been some effort to construe the semantics of polymorphic functional programming languages using the categorical notion of a natural transformation. The idea can be sketched as follows: we have a "universe of computational discourse" given by some category (in practice, a suitable category of domains). *Types* are objects of . Type *constructions* (e.g. product, function space) are functors (of appropriate arity) over . *Monomorphic* functional programs are morphisms of ; *polymorphic* programs are natural transformations. E.g.

$$\mathsf{append} : \forall t.\, t^\star \times t^\star \to t^\star$$

$$\mathsf{append} : (\cdot)^\star \times (\cdot)^\star \dot{\to} (\cdot)^\star$$

where $(\cdot)^\star : \ \to$ is the list construction functor.

These ideas are used in [10] to develop a general framework for abstract interpretation of first-order, polymorphic functions. We extend the basic approach of [10] to cover higher-order functions and show that with this extension, the short-cut techniques for computing function abstractions given in [10] can no longer work; thus our main emphasis is on re-establishing the polymorphic invariance results of [1] in this framework.

The obvious problem in extending this framework to higher-order functions is that function-space is contravariant in its first argument:

$$[\cdot \!\to\! \cdot] : \ {}^{\mathsf{op}} \times \ \to$$

(we assume throughout that is cartesian closed). Several solutions to this problem have been suggested involving dinatural transformations [5], structors [9], section-retraction pairs on domains *etc.* In this paper we pursue Reynolds' idea of viewing a type as a relation [14]. We give this a categorical formulation introducing the concepts of *relators* and *transformations* and arrive at characterising polymorphic functions as transformations between relators instead of natural transformations between functors.

In this setting, we can define the notion of *semantic polymorphic invariance*: a property $P(f)$ of polymorphic programs is a semantic polymorphic invariant if, for each $f : F \dot{\to} G$,

- *either* $P(f_A)$ for all A *or* $P(f_A)$ for no A.

In [1] it was shown that strictness analysis is a polymorphic invariant, *i.e.*, the analysis would find that an instance is strict if and only if it would find this for *all* instances. This result, which was then obtained at some labour using an operational semantics, can now be proved easily from naturality. Furthermore, we conjecture that relators and transformations form a general basis for extending abstract interpretations of monomorphic programs to cover polymorphic programs.

The paper is organised as follows. In section 2 we introduce the notion of relator and transformations between relators. Sections 3 and 4 introduce a higher-order, polymorphic functional language and show how polymorphic types can be modelled as relators and polymorphic programs as transformations between relators. In section 5 we define polymorphic invariance and demonstrate how our model of polymorphism enables us to prove a polymorphic invariance result for strictness analysis. The practical use of this fact and its relation to the results in [10] is discussed in section 6. We assume some familiarity with basic notions of category theory, see [12] or [6].

2 Relators and transformations

This section defines the notion of a *relator* and *transformations* between relators. Relators and transformations can be seen as a categorical framework for formulating Reynolds' types-as-relations paradigm. Let **Rel** be the category with sets as objects and relations as morphisms and let **C** be a

*This paper was presented at the 14th ACM Symposium on Principles of Programming Languages,1991

†This work was supported by ESPRIT grant BRA 3124 SEMANTIQUE

‡Authors' address: Dept. of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, U.K. Email: {sa,tpj}@doc.ic.ac.uk

category that can be embedded into **Rel** via a faithful functor $\mathcal{U}$. We have

Definition 1 A *relator* $R : \mathbf{C}^n \to \mathbf{C}$ maps objects in $\mathbf{C}^n$ to objects in $\mathbf{C}$ and morphisms in $\mathbf{C}^n$ to morphisms in $\mathbf{C}$, such that $f_{A\to B}$ is mapped to $R(f)_{R(A)\to R(B)}$. Furthermore R must preserve identity *i.e.*, $R(id_A) = id_{R(A)}$ Def1

Note, that relators differ from functors in that they are not required to preserve composition.

Next, we define what we mean by a transformation between relators. Assume that we have two categories, $\mathbf{C}_1$ and $\mathbf{C}_2$, with the same collection of objects. Assume furthermore we have embeddings $\mathcal{U}_1 : \mathbf{C}_1 \hookrightarrow \mathbf{Rel}$ and $\mathcal{U}_2 : \mathbf{C}_2 \hookrightarrow \mathbf{Rel}$ such that $\mathcal{U}_1$ and $\mathcal{U}_2$ agree on objects of $\mathbf{C}_1$ and $\mathbf{C}_2$.

Definition 2 A $\mathbf{C}_2$-*transformation* τ between two $\mathbf{C}_1$-relators $R, S : \mathbf{C}_1^n \to \mathbf{C}_1$ is a family of $\mathbf{C}_2$-morphisms, $\tau_A : R(A) \twoheadrightarrow S(A)$, indexed by the objects of $\mathbf{C}_1$ such that for all morphisms r from A to B in $\mathbf{C}_1^n$ the following property holds in **Rel**:

$$\mathcal{U}_2(\tau_B) \circ \mathcal{U}_1(R(r)) \subseteq \mathcal{U}_1(S(r)) \circ \mathcal{U}_2(\tau_A)$$

Def2

We write $\tau : F \quad G$ to say that τ is a transformation from relator F to relator G.

When working in concrete categories, the embeddings usually just mean that the same entity is seen from two different categorical viewpoints. We omit them when they can be derived from context. This convention will allow us to express the above requirement by the following diagram:

$$\begin{tikzcd}
A \arrow[d, two heads, "r"'] & R(A) \arrow[r, two heads, "\tau_A"] \arrow[d, two heads, "R(r)"'] & S(A) \arrow[d, two heads, "S(r)"] \\
B & R(B) \arrow[r, two heads, "\tau_B"] & S(B)
\arrow[from=1-2, to=2-3, phantom, "\subseteq"]
\end{tikzcd}$$

We shall study the case where (τ_A) is a family of functions. In this case the above condition reduces to:

$$x\ R(r)\ y \Rightarrow \tau_A(x)\ S(r)\ \tau_B(y)$$

Loosely speaking this says that a family of functions, (τ_A), is a transformation from R to S if it maps $R(r)$-related elements to $S(r)$-related elements.

We can define the composition of two transformations componentwise, *i.e.*, if $\tau : F \quad G$ and $\sigma : G \quad H$ are transformations then $\sigma \circ \tau : F \quad H$ is defined by $(\sigma \circ \tau)_A = \sigma_A \circ \tau_A$. With this definition it is easy to verify

Proposition 3 Relators and transformations form a category with relators as objects and transformations as morphisms. Prop3

3 Types as relators

It is our intention to give semantics to a polymorphic, functional language using relators and transformations. As we intend to include a fixpoint operator in our language, we shall take as our base category the cartesian closed category of complete partial orders and continuous functions, denoted by $\mathbf{CPO}_c$. From this category we can construct a new category, $\mathbf{CPO}_{si}$, with cpo's as objects and strict, inductive[1] relations as morphisms. We will now show how polymorphic type expressions can be interpreted as relators over $\mathbf{CPO}_{si}$.

Definition 4 Let R and S: ${\mathbf{CPO}_{si}}^n \to \mathbf{CPO}_{si}$ be two relators. The relator $R \times S : {\mathbf{CPO}_{si}}^n \to \mathbf{CPO}_{si}$ is defined by:

1. $(R \times S)(A)$ is the product of the objects R(A) and S(A).
2. Given a relation $r : A \twoheadrightarrow B$ the relation $(R \times S)(r)$ is defined by

$$\perp_{R(A)\times S)(A)} (R \times S)(r) \perp_{R(B)\times S)(B)}$$

$$(a, a')\ (R \times S)(r)\ (b, b') \textit{ iff } a\ R(r)\ b \textit{ and } a'\ S(r)\ b'$$

Def4

We have to verify that this defines a relator on $\mathbf{CPO}_{si}$. Firstly, it is straightforward to check that $(R \times S)(id_A) = id_{(R\times S)(A)}$. Secondly, we have to ensure that the relation so defined is strict and inductive. Strictness is by definition and inductiveness follows from structural induction on the types and the fact that if the n'th elements of two chains (a_n, a'_n) and (b_n, b'_n) are related then $\sqcup(a_n, a'_n) = (\sqcup(a_n), \sqcup(a'_n))\ (R \times S)(r)\ (\sqcup(b_n), \sqcup(b'_n)) = \sqcup(b_n, b'_n)$.

We can model the function type constructor by using the concept of logical relation [13]

Definition 5 Given two relators R and S we can define a new relator

1. $(R \Rightarrow S)(A)$ is the function space object $R(A) \Rightarrow S(A)$ in $\mathbf{CPO}_c$.
2. For $r : A \twoheadrightarrow B$ a strict,inductive relation, $R \Rightarrow S(r)$ is the relation from $(R \Rightarrow S)(A)$ to $(R \Rightarrow S)(B)$ defined by

$$\begin{array}{l} f\ (R \Rightarrow S)(r)\ g \textit{ iff} \\ \forall x, x' : x\ R(r)\ x' \text{ implies } f(x)\ S(r)\ g(x') \end{array}$$

Def5

Again, we can easily check that $(R \Rightarrow S)(id_A) = id_{(R\Rightarrow S)(A)}$. Strictness is checked as follows:

$$\begin{array}{l} \perp_{R(A)\Rightarrow S(A)} (R \Rightarrow S)(r) \perp_{R(A)\Rightarrow S(A)} \\ \Leftrightarrow \quad \forall x, x' : x\ R(r)\ x' \Rightarrow \perp_{S(A)} S(r) \perp_{S(A)} \end{array}$$

[1] A relation r is termed *strict* if $\perp r \perp$. It is termed *inductive* if for all chains (a_n) and (b_n) we have $\forall n : a_n\ r\ b_n$ implies $\sqcup a_n\ r \sqcup a_n$

For proving inductiveness let $r : A \twoheadrightarrow B$ be an arbitrary strict, inductive relation, let $(f_n), (g_n)$ be chains in $(R \Rightarrow S)(A)$ and $(R \Rightarrow S)(B)$ resp., and assume that

$$\forall n : f_n \ (R \Rightarrow S)(r) \ g_n$$

i.e.,

$$\forall n : x \ R(r) \ x' \Rightarrow f_n(x) S(r) g_n(x')$$

But since, for fixed x and x', $(f_n(x))$ and $(g_n(x'))$ are chains in $S(A)$ and $S(B)$ resp., we have that

$$\begin{array}{rcl} x \ R(r) \ x' & \Rightarrow & \forall n : f_n(x) \ S(r) \ \ g_n(x') \\ & \Rightarrow & (\sqcup(f_n))(x) \ S(r) \ \ (\sqcup(g_n))(x') \end{array}$$

4 Terms as transformations

We now give a semantics for the expressions in our polymorphic higher order language. We shall interpret a polymorphic expression of (polymorphic) type $A \to B$ as a $\mathbf{CPO}_c$-transformation between the $\mathbf{CPO}_{si}$-relators corresponding to A and B.

Polymorphic functional programs are built up from the constants id, Ap, fst, snd, fix and variables by the constructors $-;-$, $\langle -, - \rangle$ and $\Lambda(\cdot)$. The type assignments for these expressions can safely be left to the reader. Semantically, we define for domains A, B, C:

$$\begin{array}{rcl} \mathsf{fst}_{A,B}(x,y) & = & x \\ \mathsf{snd}_{A,B}(x,y) & = & y \\ \langle f, g \rangle_{A,B,C}(x) & = & (f(x), g(x)) \\ f; g(x) & = & g(f(x)) \\ \mathsf{Ap}_{A,B}(f,x) & = & f(x) \\ \Lambda(f)(x)(y) & = & f(x,y) \\ \mathsf{fix}(f) & = & \bigsqcup_{n \in \omega} f^n(\perp) \end{array}$$

etc.

LEMMA 6 Let F, G, H be relators. Then

(i) $\mathsf{fst}_{F(\_) \times G(\_)}$ is a transformation from $F \times G$ to F.

(ii) Similarly, $\mathsf{snd} : F \times G \quad G$ and $\mathsf{Ap} : (F \Rightarrow G) \times F \quad G$ are transformations.

(iii) $\mathsf{fix} : (F \Rightarrow F) \quad F$ is a transformation.

(iv) Let $f : F \quad G$, $g : F \quad H$ be transformations; then $\langle f, g \rangle : F \quad G \times H$ is a transformation.

(v) Let $f : F \times G \quad H$ be a transformation; then $\Lambda(f) : F \quad (G \Rightarrow H)$ is a transformation.

(vi) Let $f : F \quad G$ and $g : G \quad H$. Then $f; g : F \quad H$ is a transformation.

PROOF We prove (i), (iii) and (v).

(i) To show that fst is a transformation we must show that for any strict, inductive relation $r : A \twoheadrightarrow B$ we have:

$$\begin{tikzcd} (F \times G)(A) \arrow[r, two heads, "\mathsf{fst}_{F(A) \times G(A)}"] \arrow[d, two heads, "(F \times G)(r)"'] \arrow[dr, phantom, "\subseteq"] & F(A) \arrow[d, two heads, "F(r)"] \\ (F \times G)(B) \arrow[r, two heads, "\mathsf{fst}_{F(B) \times G(B)}"] & F(B) \end{tikzcd}$$

But if $(x, y) \ \mathsf{fst} \circ (F \times G)(r) \ z$ then there exists $y' \in F(A)$ such that $(x, y) \ (F \times G)(r) \ (z, y')$ *i.e.*, $x \ F(r) \ z$ and so $(x, y) \ F(r) \circ \mathsf{fst} \ z$.

(iii) Showing that fix is a transformation from $F \Rightarrow F$ to F amounts to showing that for $r : A \twoheadrightarrow B$ strict, inductive the following diagram "semi-commutes":

$$\begin{tikzcd} A \arrow[d, two heads, "r"] & (F \Rightarrow F)(A) \arrow[r, two heads, "\mathsf{fix}_A"] \arrow[d, two heads, "(F \Rightarrow F)(r)"'] \arrow[dr, phantom, "\subseteq"] & F(A) \arrow[d, two heads, "F(r)"] \\ B & (F \Rightarrow F)(B) \arrow[r, two heads, "\mathsf{fix}_B"] & F(B) \end{tikzcd}$$

which asserts that if two continuous functions f, g are related by $(F \Rightarrow F)(r)$ then their fixpoints are related by $F(r)$. But since $F(r)$ is inductive, $\mathsf{fix}_A(f)$ is related to $\mathsf{fix}_B(g)$ if $f^n(\perp) \ F(r) \ g^n(\perp)$ for all n which follows from $\perp F(r) \perp$ and $f \ (F \Rightarrow F)(r) \ g$.

(v) We have to show that $\Lambda(f) : F \quad (G \Rightarrow H)$ is a transformation provided $f : F \times G \quad H$ is. In diagrams

$$\begin{tikzcd} A \arrow[d, two heads, "r"] & F(A) \arrow[r, two heads, "\Lambda(f)_A"] \arrow[d, two heads, "F(r)"'] \arrow[dr, phantom, "\subseteq"] & (G \Rightarrow H)(A) \arrow[d, two heads, "(G \Rightarrow H)(r)"] \\ B & F(B) \arrow[r, two heads, "\Lambda(f)_B"] & (G \Rightarrow H)(B) \end{tikzcd}$$

This states that under the assumption that $xF(r)x'$ we can prove

$$\Lambda(f)_A(x) \ (G \Rightarrow H)(r) \ \Lambda(f)_B(x')$$

which is equivalent to

$$\forall y, y' : yG(r)y' \text{ implies } f_A(x,y)H(r)f_B(x',y')$$

The last property follows from $xF(r)x'$ and the fact that f is a transformation from $F \times G$ to H.

LEMMA6

We can summarise the results obtained so far as:

THEOREM 7 Let e be a polymorphic expression that maps arguments of (polymorphic) type F to a result of (polymorphic) type G. Then e can be interpreted as a $\mathbf{CPO}_c$-transformation between the $\mathbf{CPO}_{si}$-relators corresponding to types F and G. THEOREM7

The general import of the results so far is that we can derive from a cartesian closed category (*i.e.*, a model of our monomorphically typed, higher order language) a new cartesian closed category of relators and transformations in which we can interpret ML-style parametric polymorphism. For further details, see [4].

5 Semantic Polymorphic Invariance and Strictness

The aim of this section is to provide some initial evidence that the relator-semantics for polymorphism can be a useful tool when trying to extend methods for analysing monomorphic programs to cover polymorphic programs. The framework for analysis which we have in mind is *abstract interpretation* [3]. In the following, we demonstrate how a particular analysis, *strictness analysis* of higher-order, monomorphic functions, can give information about polymorphic functions.

5.1 Strictness analysis

A function is *strict* if its result is undefined whenever its argument is undefined. The use of strictness information to implement functional languages efficiently is widely studied [7] ,[11]. In [8] it was shown how to perform strictness analysis of higher-order, monomorphically typed programs by abstract interpretation. The method works by interpreting the program text over non-standard domains built up from the two-point domain $\mathbf{2}$ using the usual domain constructors $\times$ and $\Rightarrow$. The abstract interpretation of a function $f : Int \rightarrow Int$, say, would then be the function $f^{\#} : \mathbf{2} \rightarrow \mathbf{2}$ obtained from f by replacing the operations associated with type Int, like +, by their abstract interpretation on the domain $\mathbf{2}$. The main result in [8] is that $f^{\#}$ gives safe information about the strictness properties of f, *i.e.*, $f^{\#}(\perp) = \perp \Rightarrow f(\perp) = \perp$. As domains built over $\mathbf{2}$ are all finite lattices, $f^{\#}$ can be computed by a fixpoint iteration that is guaranteed to terminate.

This gives a computable method for doing strictness analysis of monomorphic, higher-order functions. Strictness properties of a polymorphic function can be obtained by applying the method to the monomorphic instances of the function. There are, however, two problems connected with this approach:

- The number of monomorphic instances of a polymorphic function becomes infinite as soon as we allow structured or higher-order types.
- The size of the abstract domain for structured and higher-order types grows so fast that fixpoint computations become infeasible

The solution to these problems is to prove that the strictness analysis described here is *polymorphically invariant.* We prove that all *abstract* functions corresponding to monomorphic instances have the same strictness properties. This implies that we only have to compute the abstraction of the simplest instance of the function to obtain all information about strictness that the analysis can provide.

5.2 Polymorphic invariance

Polymorphic invariance of a property P of a polymorphic program means that P holds for all monomorphic instances or none. The notion of polymorphic invariance can be transferred to the semantic level as follows:

DEFINITION 8 Let P be a property of morphisms. P is a *semantic polymorphic invariant* (with respect to a class of objects $\mathcal{T}$ and a class of transformations $\mathcal{F}$) if, for every $A, B \in \mathcal{T}$ and $f \in \mathcal{F}$: $P(f_A) \iff P(f_B)$. DEF8

The particular property we shall be interested in is that of *strictness* of a function: $\mathsf{strict}(f) \stackrel{\text{def}}{=} f\perp = \perp$. The proof of polymorphic invariance of strictness is based on two relations, r_{refl} and r_{pres} defined as follows:

DEFINITION 9 Let $\mathbf{2}$ denote the two-point lattice and let A be a domain with a greatest element $\top$ different from $\perp$. Define the relations $r_{pres}, r_{refl} : \mathbf{2} \twoheadrightarrow A$ by

$$
\begin{array}{llll}
\perp_{\mathbf{2}}\ r_{pres}\ \perp_A & & \perp_{\mathbf{2}}\ r_{refl}\ a & \forall a \in A \\
\top_{\mathbf{2}}\ r_{pres}\ a & \forall a \in A & \top_{\mathbf{2}}\ r_{refl}\ \top_A &
\end{array}
$$

DEF9

In the following, let F denote an arbitrary relator built using $\times$ and $\Rightarrow$.

LEMMA 10 $F(r_{pres})$ and $F(r_{refl})$ is strict, inductive and relates $\top$ to $\top$.
PROOF Straightforward LEMMA10

LEMMA 11 $F(r_{refl})$ is $\perp$-reflecting and $F(r_{pres})$ is $\perp$-preserving, *i.e.*,

$$
\begin{gathered}
d\ r_{refl}\ \perp_{F(A)} \Rightarrow d = \perp_{F(\mathbf{2})} \\
\perp_{F(\mathbf{2})}\ r_{pres}\ d \Rightarrow d = \perp_{F(A)}
\end{gathered}
$$

PROOF That r_{refl} is $\perp$-reflecting is essentially Proposition 4.2 in [2]. The proof that r_{pres} is $\perp$-preserving proceeds by induction on the structure of F. We shall only give the case when F is a function type. So suppose F has the form $G \Rightarrow H$ and that $\lambda x.\perp\ (G \Rightarrow H)(r_{pres})\ f$ for some $f \in (G \Rightarrow H)(A)$. From lemma 10 we have that $\top\ G(r_{pres})\ \top$, hence, as $(G \Rightarrow H)(r_{pres})$ is a logical relation we get that $\perp\ H(r_{pres}) f(\top)$ and by induction hypothesis that $f(\top) = \perp$. The monotonicity of f now implies that $f = \lambda x.\perp$ as required. LEMMA11

With these lemmas at hand we can now prove

PROPOSITION 12 Strictness is a semantic polymorphic invariant with respect to the class of all non-trivial domains with top-elements in $\mathbf{CPO}_c$ (i.e. those with more than one element), and all transformations.

PROOF Let f be a transformation from relator F to relator G *i.e.*, f_A is a continuous function from $F(A)$ to $G(A)$ for any domain A. We now prove f_A is strict iff $f_{\mathbf{2}}$ is strict. The relations r_{refl} and r_{pres} are strict and inductive, so from the fact that f is a transformation we get the following diagram, where r stands for either r_{refl} or r_{pres}:

$$\begin{tikzcd}
\mathbf{2} \arrow[d, "r"'] & F(\mathbf{2}) \arrow[r, twoheadrightarrow, "f_{\mathbf{2}}"] \arrow[d, "F(r)"'] \arrow[dr, phantom, "\subseteq"] & G(\mathbf{2}) \arrow[d, "G(r)"] \\
A & F(A) \arrow[r, twoheadrightarrow, "f_A"] & G(A)
\end{tikzcd}$$

So suppose $f_{\mathbf{2}}$ is strict and that $f_A(\perp) = d$. We show that $d = \perp$. Take r to be r_{pres}. As $F(r_{pres})$ is strict we have that $\perp\ f_A \circ F(r_{pres})\ d$ and hence $\perp\ G(r_{pres}) \circ f_{\mathbf{2}}\ d$ *i.e.*, $f_{\mathbf{2}}(\perp)\ G(r_{pres})\ d$. As f_2 is strict we get $\perp\ G(r_{pres})\ d$ and lemma 11 gives that $d = \perp$ *i.e.*, f_A is strict.

For the converse, assume f_A is strict and let r be r_{refl} in the diagram. The lower part of the diagram is thus a strict relation hence the upper part of the diagram is a strict relation, so $\perp\ G(r_{refl}) \circ f_{\mathbf{2}}\ \perp$. Lemma 11 tells us that $G(r_{refl})$ is $\perp$-reflecting, so $f_{\mathbf{2}}(\perp) = \perp$ hence $f_{\mathbf{2}}$ is strict. PROP12

6 Related Approaches

In [10], an ingenious method is given to compute any instance, f_A, of an abstract function from its simplest instance $f_{\mathbf{2}}$. The method works for any abstract interpretation, but is limited to first-order functions. The basic result from [10] which underlies the method given there is:

PROPOSITION 13 If A is a type built from product, lifting and $\mathbf{1}$, and $f : F \dot{\to} G$ is a polymorphic program, where F, G are first-order types (i.e. not using function space), then for some function Φ_A, depending on A but *not* on f,

$$f_A = \Phi_A(f_{\mathbf{2}}).$$

PROP13

Even if a property can be shown to polymorphic invariant, information about the abstract functions at non-basic types might be needed. This occurs when a function g calls another function f with an argument of non-basic type. In this case the abstract function of g will be defined in terms of a non-basic instance of the abstract function of f. An obvious question is thus whether the result in proposition 13 carries over in the presence of function spaces. The next result shows that it does not.

PROPOSITION 14 The previous proposition fails when F and G are allowed to be higher-order.

PROOF Consider the polymorphic integers:

$$\bar{n} \stackrel{\text{def}}{=} \lambda f.\lambda x.f^{(n)}(x) : \forall t.(t \to t) \to (t \to t) \quad (n \in \quad).$$

For $n \neq m$ we can find a finite lattice type A such that $\bar{n}_A \neq \bar{m}_A$ (let $k = \max(m, n)$, and take $A = \mathbf{2}^k$). But there are only finitely many functions in $[[\mathbf{2} \Rightarrow \mathbf{2}] \Rightarrow [\mathbf{2} \Rightarrow \mathbf{2}]]$. PROP14

Note that this result holds for any finite domain in place of $\mathbf{2}$, and even when A is built from product and lifting (so it is independent of what our abstract interpretation of function space happens to be). The conclusion of all this is that analysis of higher-order, polymorphic functions is best done by proving a polymorphic invariance result for the analysis at hand and then computing those non-basic instances of the abstract functions necessary to compute all basic instances of the abstract functions.

7 Conclusion

The major achievements of this paper can be summarised as:

- We have given a semantics for a first order polymorphic, higher order functional language. The semantics is a combination of Reynolds' types-as-relations idea and the idea of naturality from category theory. This approach to polymorphism is treated in depth in [4].
- using the naturality property we can give a simple proof of the polymorphic invariance of strictness analysis.

We take this as initial evidence of the fact that relators and transformations can be of use in analysis of polymorphic programs. They seem especially useful for establishing polymorphic invariance results, which is the best one can hope for when dealing with higher-order functions. Future work should investigate if polymorphic invariance can be proved for abstract interpretation in general using the relator semantics of polymorphism.

8 Acknowledgements

The authors would like to thank the members of the Abstract Interpretation Group at Imperial College (Geoff Burn, Chris Hankin, Sebastian Hunt and David Sands) for numerous discussions related to this topic. Thanks are also due to Paul Taylor for letting us use his TEX-macros for drawing diagrams.

References

[1] S. Abramsky. Strictness analysis and polymorphic invariance. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, pages 1–23, Berlin, 1986. Springer Verlag. Lecture Notes in Computer Science Vol. 217.

[2] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1, 1990.

[3] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[4] S. Abramsky, J. Mitchell, A. Scedrov, and P. Wadler. Relators. Draft paper, 1990.

[5] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. In G. Huet, editor, *Logical Foundations of Functional Programming*. Addison Wesley, 1990.

[6] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[7] G.L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Imperial College, University of London, 1987.

[8] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.

[9] P. Freyd. Structural polymorphism. Unpublished, 1989.

[10] R.J.M. Hughes. Abstract interpretation of first–order polymorphic functions. In *Glasgow Workshop on Functional Programming*, University of Glasgow, Department of Computing Science, August 1988. Research Report 89/R4.

[11] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[12] S. MacLane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[13] G. D. Plotkin. λ-definability and logical relations. Technical Report SAI-RM-4, School of A.I., University of Edinburgh, 1973.

[14] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*. North Holland, 1983.