Polymorphic Strictness Analysis using Frontiers Draft version: comments welcomed

Julian Seward*
University of Manchester
sewardj@uk.ac.man.cs

September 23, 1992

Abstract

This paper shows how to implement sensible polymorphic strictness analysis using the Frontiers algorithm. A central notion is to only ever analyse each function once, at its simplest polymorphic instance. Subsequent non-base uses of functions are dealt with by generalising their simplest instance analyses. This generalisation is done using an algorithm developed by Baraki, based on embedding-closure pairs. Compared with an alternative approach of expanding the program out into a collection of monomorphic instances, this technique is hundreds of times faster for realistic programs. There are some approximations involved, but these do not seem to have a detrimental effect on the overall result. The overall effect of this technology is to considerably expand the range of programs for which the Frontiers algorithm gives useful results reasonably quickly.

1 Introduction

The Frontiers algorithm was introduced in [CP85] as an allegedly efficient way of doing forwards strictness analysis, although this presentation was restricted to the two-point, first-order, monomorphic case. Further work showed what a versatile beast it was: higher-order analysis [HH91] and arbitrary non-flat datatypes [Sew91] were subsequently incorporated. Recently, Hankin and Hunt showed a variant [HH92] which greatly increased controllability by allowing arbitrary tradeoffs between work and quality of result.

What remains missing is an effective way to deal with parametric polymorphism. Polymorphic strictness analysis in the large has until recently been a grey area. Investigations of the semantics of polymorphic functions by Abramsky [Abr85] showed that strictness analysis was polymorphically invariant but offered no method of exploiting the result. Subsequently, Hughes developed a way to deal effectively with first order

* Author's address: Department of Computer Science, Victoria University of Manchester, Oxford Road, Manchester M13 9PL, UK. Tel: +44 61 275 6291 Fax: +44 61 275 6236

polymorphism, on which a successful implementation was based [KHL91]. A more recent paper by Abramsky and Jensen [AJ91] investigated higher-order polymorphic functions but still fails to offer a useable algorithm. This impasse was finally broken by Baraki's work on embedding-closure pairs [Bar91] which supplied a simple way to relate strictness information of different polymorphic instances.

One obvious way to deal with polymorphic programs is to expand them out into a collection of monomorphic instances, and analyse each instance separately. Unfortunately, as shown in [HH91] section 5, when coupled with Frontiers, this approach is so appallingly expensive even for trivial functions as to be quite impractical. Making a particular instance of a polymorphic function f means making instances of all polymorphic functions which f depends on, either directly or indirectly. Since these could be in diverse modules, monomorphisation also interacts badly with separate compilation.

What would be preferable is to analyse each function only once, at its simplest polymorphic instance, wherein each type variable is assigned the two-point domain. Subsequent uses at non-basic instantiations are dealt with by generalising the simplest instance analysis. This paper shows how Baraki's work [Bar91] can be used to achieve such a goal. The technique has been successfully implemented, and speedup factors of literally hundreds have been achieved relative to making a monomorphic expansion.

Alas, as ever, the lunch is not quite free. In the higher-order case, Baraki's technique generates non-exact results. The effect of this inexactitude seems fairly minor. Section 5 shows some real results and performance figures.

2 Technical preliminaries

2.1 Source language and types

The abstract interpretation is defined over a simple lazy functional language called **Core**. Core is not intended as a vehicle for writing functional programs. Rather, Core is a useful staging post in the journey from the considerable complexity of Haskell [HWe90] to nasty details of a particular back end. In particular, Core is used as an internal form in the forthcoming Glasgow Haskell Compiler. Additionally, Core has been used for didactic purposes, as the source language for a text on lazy FL back ends [PL92].

For the purposes of this discussion, what is interesting is not the syntax of Core but the abstract interpretation of its types. Unsurprisingly, Core deals in the usual Milner-Hindley types [Mil78] which are base types, like `Int`, function spaces, user-defined sum-of-products types and type variables. The strictness analysis is BHA style forwards strictness analysis [BHA85]. We define functor D which gives the abstract interpretation of every type:

$D_{\texttt{Int}} = \mathbf{2}$

$D_\alpha = \mathbf{2}$ where α is any type variable

$D_{(typename\ \ te_1\ \dots\ te_n)} = ((D_{te_1} \times\ \dots\ \times D_{te_n})_\perp)_\perp$
if *typename* is a recursive type

$D_{(typename\ te_1\ ...\ te_n)} = (D_{te_1} \times\ ...\ \times D_{te_n})_\perp$
if *typename* is not a recursive type

$D_{(te_1 \to ... \to te_n \to te_t)} = [D_{te_1} \times \ldots \times D_{te_n} \to D_{te_t}]$
provided te_t is not itself a function space

This last condition indicates all functions are represented in their uncurried form, so an eta-abstraction pass is necessary.

The rules for user-defined types are a generalisation of Philip Wadler's original abstract interpretation for lists [Wad87]. In this scheme, a user-defined type is represented by the product of the domains of the type arguments, lifted twice for recursive types and once otherwise. Enumerated types, like `Bool`, are therefore represented as the lifting of an empty product. This interpretation regards such a domain as equivalent to **2**, which it rightly is, and thus enumerated types acquire domains isomorphic to that of `Int`. There are some restrictions on non-flat types needed to make the interpretation tractable. These are fairly inconsequential in their restriction of the programmer, and are reported independantly in [Sew91] section 6.4.2 and [KHL91] section 7.

A concept needed below is that of *domain variable*, which plays the same rôle in domain expressions as type variables do in type expressions. Hence, we really ought to write

$$D_\alpha = \alpha$$

understanding that the left α is a type variable and the right one is a domain variable. Furthermore, analysis of a polymorphic function is done at its simplest instance, with all type variables replaced by **2**. In what follows, α and β denote domain variables.

2.2 Representing functions as frontiers

Implied from the abstract interpretation of types is the family $\mathcal{L}$ of distributive lattices needed:

$$\begin{array}{rll}
\mathbf{2} & \in \mathcal{L} & \\
D_\perp & \in \mathcal{L} & \text{if } D \in \mathcal{L} \\
(D_1 \times \ldots \times D_n) & \in \mathcal{L} & \text{if } D_1 \in \mathcal{L} \wedge \ldots \wedge D_n \in \mathcal{L} \\
[D_1 \times \ldots \times D_n \to D_t] & \in \mathcal{L} & \text{if } D_1 \in \mathcal{L} \wedge \ldots \wedge D_n \in \mathcal{L} \wedge D_t \in \mathcal{L}
\end{array}$$

For this last case, D_t may not itself be a function space, although it may contain embedded function spaces. In other words, all functions are represented in their uncurried form.

Representation of non-function values is straightforward. For functions, the representation defined in [HH91] is used. Bearing in mind that D_t is never a function space, there are three possibilities for a function f:

- $D_t = A_\perp$. In this case we represent the function by a *low factor* $\in [D_1 \times \ldots \times D_n \to \mathbf{2}]$ and a *high factor* $\in [D_1 \times \ldots \times D_n \to A]$, where the low factor produces 0 if f maps an argument to $\perp_{A_\perp}$, and the high factor tells us what happens when f does not map a value to $\perp_{A_\perp}$.

In what follows, a function of this form is represented by the term `(RepLift lf hf)` where the lf and hf are the low and high factors. Note that to keep representations unique it is necessary to observe the convention that for all $x_1 \ldots x_n$, if $lf(x_1 \ldots x_n) = 0$ then $hf(x_1 \ldots x_n) = \perp_A$.

- $D_t = (A_1 \times \ldots \times A_m)$. A function of this form is isomorphic to a collection of m functions with the same argument domains, but with result domains A_1 to A_m respectively. Accordingly, this is represented by a list of the m functions, with the term (RepCross [$\mathtt{f}_1$, $\mathtt{f}_2$, ... $\mathtt{f}_m$]).

- $D_t = \mathbf{2}$. This is where the frontiers come in. The set $F_1 = Min(f^{-1}(1))$ is termed the *minimum-1* frontier and is a complete representation of the function graph of f. Alternatively, the *maximum-0* frontier $F_0 = Max(f^{-1}(0))$ is equivalent. For technical reasons it is essential to retain both. A value of this type is denoted (Min1Max0 $\mathtt{F}_1$ $\mathtt{F}_0$).

  How are F_1 and F_0 actually used? Well, suppose $F_1 = f^{-1}(1)$ and $F_0 = f^{-1}(0)$. Then we could tell what the value of $f(x)$ is by considering F_1:

  $$\begin{aligned} f(x) &= 1 \quad \text{if } x \in F_1 \\ &= 0 \quad \text{otherwise} \end{aligned}$$

  But F_0 contains exactly the same information, since

  $$\begin{aligned} f(x) &= 0 \quad \text{if } x \in F_0 \\ &= 1 \quad \text{otherwise} \end{aligned}$$

  Since f can only map points to 0 or 1, the sets $f^{-1}(1)$ and $f^{-1}(0)$ completely cover the argument lattice for f, whilst remaining disjoint. Furthermore, since f is monotonic, $f^{-1}(1)$ is an *upper set*. In handwaving terms, this means it completely fills some upper part of the argument lattice, without any holes. Similarly, $f^{-1}(0)$ is a *lower set*.

  A more economical way to store an upper set U is obtained by observing the one-to-one correspondance between U and $Min(U)$: rather than storing all of U, we only need to store the points on the lower boundary of it, since any point above the lower boundary (or frontier) is also guaranteed to be in U. $Min(U)$ computes that lower boundary by filtering out all those points $x \in U$ for which another point $y \in U$ satisfying $y \sqsubset x$ can be found. Typically, $Min(U)$ is much smaller than U, giving a considerable saving in representation. Dually, lower sets like $f^{-1}(0)$ are uniquely represented by their upper frontier, as computed using Max.

  Therefore, we define $F_1 = Min(f^{-1}(1))$ and $F_0 = Max(f^{-1}(0))$, and use them thus:

  $$\begin{aligned} f(x) &= 1 \quad \text{if } \exists y \in F_1 \,.\, y \sqsubseteq x \\ &= 0 \quad \text{otherwise} \end{aligned}$$

  F_0 contains exactly the same information, since

  $$\begin{aligned} f(x) &= 0 \quad \text{if } \exists y \in F_0 \,.\, y \sqsupseteq x \\ &= 1 \quad \text{otherwise} \end{aligned}$$

For reasons of efficiency and symmetry, it is convenient to generate and use both representations, despite the redundancy introduced.

The astute reader may have noticed that products and liftings are always used together. In an implementation, considerable performance gain is obtained by merging the products and liftings so only those lattices required by the abstract interpretation can be represented. This implementation technicality is henceforth ignored.

3 The algorithm

3.1 Overview

The algorithm is based directly on Gebreselassie Baraki's paper [Bar91]. Readers interested in the theory should refer to it. What follows is merely a description of how to implement it.

Since all polymorphic functions are analysed at their simplest instance, the only remaining problem is how to manufacture arbitrary instances of the function from the simplest one. The algorithm's input consists of three pieces of information:

- The abstract interpretation of the simplest instance, $f_{2\ldots 2}$.
- The domain expression D of f, with domain variables left in place. This must be of the form $[D_1 \times \ldots \times D_n \rightarrow D_t]$. In what follows, $F = (D_1 \times \ldots \times D_n)$ is termed the *argument domain expression* and $G = D_t$ the *result domain expression*. Let $v_1 \ldots v_m$ be the m domain variables in D.
- A mapping $\tau = \{v_1 \mapsto A_1 \ldots v_m \mapsto A_m\}$ from the domain variables in D to arbitrary domain expressions $A_1 \ldots A_m$. This is the instantiation required. Note that we never instantiate a domain variable to an expression which contains another domain variable. This restriction is automatically observed because all analysis is done at the simplest instance.

Hence, the domain of the instantiation must be $\tau(D)$ and that of the simplest instance $\tau_2(D)$ where τ_2 maps all domain variables to $\mathbf{2}$. Let $f_{A_1\ldots A_m}$ denote the instance of f which is to be created. In order to relate $f_{A_1\ldots A_m}$ to $f_{2\ldots 2}$ it is necessary to relate $\tau_{\mathbf{2}}(F)$ to $\tau(F)$, and also $\tau_{\mathbf{2}}(G)$ to $\tau(G)$.

In general, it is useful to associate with some arbitrary domain expression H an environment ρ which binds the domain variables in H to particular points. This is denoted H_ρ. If, for example, we were to consider the domain expression D and the mapping τ above, ρ would bind each v_i to some $a_i \in A_i$. This gives rise to an embedding-closure pair D_ρ from $\tau_{\mathbf{2}}(D)$ to $\tau(D)$. That is, D_ρ is a pair of functions (D^e_ρ, D^c_ρ) satisfying the usual embedding-closure pair axioms $(D^e_\rho . D^c_\rho) \sqsupseteq id$ and $(D^c_\rho . D^e_\rho) = id$. Do not confuse ρ with τ, the latter of which binds domain variables to *domain expressions* and not to actual points.

With this terminology in place, Baraki's main result and the core of the algorithm are stated hence:

$$f_{A_1\ldots A_m} \sqsubseteq \sqcap \; \{ \; G^e_\rho \, . \, f_{2\ldots 2} \, . \, F^c_\rho \quad | \; \rho \in \mathcal{P}\}$$

where the indexing set $\mathcal{P}$ is defined as

$$\mathcal{P} = \{ \ \{v_1 \mapsto a_1 \ldots v_m \mapsto a_m\} \quad | \ a_1 \in \text{zapTop}(A_1), \ldots, a_m \in \text{zapTop}(A_m) \ \}$$

and

$$\text{zapTop}(X) = X - \{\top_X\}$$

$\mathcal{P}$ is a collection of environments ρ in which each ρ represents a different way of binding the domain variables in D to the points in the instantiating lattices $A_1 \ldots A_m$. A small subtlety is the need to remove the top points of the instantiating lattices as these do not give rise to embedding-closure pairs.

Computing the overall meet ($\sqcap$) is straightforward given that any higher-order analyser will have to provide a way to compute meets and joins in function spaces. The really interesting part of the problem is computation of the "sandwiches" $(G^e_\rho \ . \ f_{2\ldots 2} \ . \ F^c_\rho)$. It is essential to manipulate the frontier representations directly, rather than compute the compositions by extensional means, as the latter approach involves frontier searches in the instantiation lattice and thus comprehensively defeats the purpose of using this technique at all.

The compositions may be built in either order, as $((G^e_\rho \ . \ f_{2\ldots 2}) \ . \ F^c_\rho)$ or $(G^e_\rho \ . \ (f_{2\ldots 2} \ . \ F^c_\rho))$ so the problem breaks into two: given some suitable function f, computing $(G^e_\rho \ . \ f)$ and $(f \ . \ F^c_\rho)$. These are detailed below. The latter problem requires a way to to apply G^e_ρ to points, and since that is a simpler problem, we begin with it.

3.2 Computing $D^e_\rho(x)$

This follows directly from Baraki's definitions. Note however that the lifting construction $D_\perp$ is an extension. The behaviour of $D_\perp = (D^e_\perp, D^c_\perp)$ is defined so that if D is an embedding-closure pair then so is $D_\perp$, a trivially verified result.

For some domain variable α the environment supplies a point $\rho(\alpha)$. Let $\mathcal{D}(\rho(\alpha))$ denote the domain to which the point belongs. Also, let $P \preceq Q$ indicate that P precedes Q in the instantiation ordering, so, intuitively, P is "smaller" than Q. This gives:

$$
\begin{array}{lll}
\alpha^e_\rho \ (x) & = & \text{if } x = 0 \text{ then } \rho(\alpha) \text{ else } \top_{\mathcal{D}(\rho(\alpha))} \\
\alpha^e_\rho & :: & \mathbf{2} \rightarrow \mathcal{D}(\rho(\alpha)) \\
\mathbf{2}^e_\rho \ (x) & = & x \\
\mathbf{2}^e_\rho & :: & \mathbf{2} \rightarrow \mathbf{2} \\
D^e_{\perp\,\rho} \ (x) & = & \text{if } x = \perp_{P_\perp} \text{ then } \perp_{Q_\perp} \text{ else } lift(D^e_\rho(drop(x))) \text{ where } drop(lift(a)) = a \\
D^e_{\perp\,\rho} & :: & P_\perp \rightarrow Q_\perp \\
 & & \text{provided } P \preceq Q \\
(D_1 \times \ldots \times D_n)^e_\rho \ (x_1 \ldots x_n) & = & (D^e_{1\,\rho}(x_1) \ldots D^e_{n\,\rho}(x_n)) \\
(D_1 \times \ldots \times D_n)^e_\rho & :: & (P_1 \times \ldots \times P_n) \rightarrow (Q_1 \times \ldots \times Q_n) \\
 & & \text{provided } P_1 \preceq Q_1 \ \ldots P_n \preceq Q_n \\
[F \rightarrow G]^e_\rho \ (f) & = & G^e_\rho \ . \ f \ . \ F^c_\rho \\
[F \rightarrow G]^e_\rho & :: & [X_s \rightarrow X_t] \rightarrow [F_s \rightarrow G_t] \\
 & & \text{provided } f :: [X_s \rightarrow X_t] \\
 & & \text{and } F \equiv [F_s \rightarrow F_t] \text{ and } G \equiv [G_s \rightarrow G_t]
\end{array}
$$

Applying the corresponding closure domain expression D^c_ρ is almost identical, except for the function space and variable cases. Interestingly, though, they are not needed here. See Baraki for further details.

Clearly, domain expressions may only be applied to values of the correct "type". This is assured by the typechecking phase which precedes strictness analysis.

3.3 Computing $(f \,.\, F^c_\rho)$

The function $(f \,.\, F^c_\rho)$ acts like f but its arguments pass through F^c_ρ first. It should not be entirely surprising therefore that this is achieved by mapping F^e_ρ through the points in the maximum-0 frontiers in f, and recomputing the equivalent minimum-1 frontiers. The operation is pushed down through the function in the obvious way:

$$((RepLift\ lf\ hf) \,.\, F^c_\rho) \quad = \quad RepLift\ (lf \,.\, F^c_\rho)\ (hf \,.\, F^c_\rho)$$

$$((RepCross\ [f_1 \ldots f_n]) \,.\, F^c_\rho) \quad = \quad RepCross\ [(f_1 \,.\, F^c_\rho) \ldots (f_n \,.\, F^c_\rho)]$$

The base case uses an auxiliary function **min1FromMax0** to recalculate the corresponding minimum-1 frontier:

$$\begin{aligned}((Min1Max0\ F_1\ F_0) \,.\, F^c_\rho) \quad = \quad & Min1Max0\ \ (\text{min1FromMax0}(newF_0))\ \ newF_0 \\ & \text{where}\ \ newF_0 = \{F^e_\rho(x) \mid x \in F_0\}\end{aligned}$$

Min1FromMax0 must be defined with some care to avoid running into massive inefficiencies. Using the function **succs** defined in [HH91], we have

$$\text{min1FromMax0}(F_0) = \sqcap_{min1}\{\text{succs}(x) \mid x \in F_0\}$$

where $\sqcap_{min1}$ computes the meet of a set of minimum-1 frontiers, as defined in the same paper.

Proof of base case

Let F_0 be the maximum-0 frontier of $f \in [\ldots \to \mathbf{2}]$. Let $g = (f \,.\, F^c_\rho)$, with $G_0 = \{F^e_\rho(x) \mid x \in F_0\}$. We need to show that G_0 is the maximum-0 frontier of g.

$$\begin{aligned}\Longrightarrow\quad & \text{Suppose } (f \,.\, F^c_\rho)(y) = 0 \\ & \Rightarrow f(F^c_\rho(y)) = 0 \\ & \Rightarrow \exists x \in F_0 \,.\, F^c_\rho(y) \sqsubseteq x \\ & \Rightarrow \exists x \in F_0 \,.\, F^e_\rho(F^c_\rho(y)) \sqsubseteq F^e_\rho(x) \\ & \Rightarrow \exists x \in F_0 \,.\, y \sqsubseteq F^e_\rho(x) \\ & \Rightarrow \exists x_g \in G_0 \,.\, y \sqsubseteq x_g \\ & \Rightarrow g(y) = 0\end{aligned}$$

$$\begin{aligned}\Longleftarrow\quad & \text{Suppose } g(y) = 0. \\ & \Rightarrow \exists x \in F_0 \,.\, y \sqsubseteq F^e_\rho(x) \\ & \Rightarrow \exists x \in F_0 \,.\, (f \,.\, F^c_\rho)(y) \sqsubseteq (f \,.\, F^c_\rho)(F^e_\rho(x)) \\ & \Rightarrow \exists x \in F_0 \,.\, (f \,.\, F^c_\rho)(y) \sqsubseteq f(x) \\ & \Rightarrow (f \,.\, F^c_\rho)(y) = 0\ \ \text{since}\ \ f(x) = 0\end{aligned}$$

Hence $(f \,.\, F^c_\rho)(y) = 0$ if and only if $g(y) = 0$, so the base case is proved.

3.4 Computing $(G^e_\rho \,.\, f)$

The function $(G^e_\rho \,.\, f)$ acts like f but sends its result through G^e_ρ. Manufacturing the composition boils down to glueing together various copies of f along with $\bot_{\mathcal{D}(f)}$ and $\top_{\mathcal{D}(f)}$. Most cases are straightforward, with the transformation simply propogating inwards:

$$
\begin{aligned}
\mathbf{2}^e_\rho \,.\, f &= \lambda x \,.\, \mathbf{2}^e_\rho(f(x)) \\
&= \lambda x \,.\, f(x) \\
&= f
\end{aligned}
$$

$$
\begin{aligned}
D^e_{\bot\ \rho} \,.\, (f \text{ \textbf{as} } \mathit{RepLift}\ lf\ hf) &= \lambda x \,.\, D^e_{\bot\ \rho}(f(x)) \\
&= \lambda x \,.\, D^e_{\bot\ \rho}(\text{if } \ lf(x) = 0 \ \text{ then } \ \bot \ \text{ else } \ lift(hf(x))) \\
&= \lambda x \,.\, (\text{if } \ lf(x) = 0 \ \text{ then } \ \bot \ \text{ else } \ lift(D^e_\rho(drop(lift(hf(x)))))) \\
&= \mathit{RepLift}\ lf\ (\lambda x \,.\, D^e_\rho(hf(x))) \\
&= \mathit{RepLift}\ lf\ (D^e_\rho \,.\, hf)
\end{aligned}
$$

$$
\begin{aligned}
(D_1 \times \ldots \times D_n)^e_\rho \,.\, (f \text{ \textbf{as} } \mathit{RepCross}\ [f_1 \ldots f_n]) &= \lambda x \,.\, (D_1 \times \ldots \times D_n)^e_\rho(f(x)) \\
&= \lambda x \,.\, (D_1 \times \ldots \times D_n)^e_\rho(f_1(x) \ldots f_n(x)) \\
&= \lambda x \,.\, (D^e_{1\ \rho}(f_1(x)) \ldots D^e_{n\ \rho}(f_n(x))) \\
&= \mathit{RepCross}\ [(D^e_{1\ \rho} \,.\, f_1) \ldots (D^e_{n\ \rho} \,.\, f_n)]
\end{aligned}
$$

The variable case $(\alpha^e_\rho \,.\, f)$ is considerably more tricky because it depends on the value of $\rho(\alpha)$. Environment updates are denoted $\rho\{v \mapsto y\}$ indicating that ρ now binds v to y, regardless of its previous binding.

Case 1.1: $\mathcal{D}(\rho(\alpha)) = \mathbf{2}$ **and** $\rho(\alpha) = 0$

$$
\begin{aligned}
\alpha^e_\rho \,.\, f &= \lambda x \,.\, \text{if } \ f(x) = 0 \ \text{ then } \ 0 \ \text{ else } \ 1 \\
&= f
\end{aligned}
$$

Case 1.2: $\mathcal{D}(\rho(\alpha)) = \mathbf{2}$ and $\rho(\alpha) = 1$

$$\begin{aligned}
\alpha^e_\rho \,.\, f &= \lambda x \,.\, \text{if } f(x) = 0 \text{ then } 1 \text{ else } 1 \\
&= \top_{\mathcal{D}(f)}
\end{aligned}$$

Case 2.1: $\mathcal{D}(\rho(\alpha)) = (D_1 \times \ldots \times D_n)$ and $\rho(\alpha) = (y_1 \ldots y_n)$

$$\begin{aligned}
\alpha^e_\rho \,.\, f &= \lambda x \,.\, \text{if } f(x) = 0 \text{ then } (y_1 \ldots y_n) \text{ else } (\top_{D_1} \ldots \top_{D_n}) \\
&= (\lambda x \,.\, \text{if } f(x) = 0 \text{ then } y_1 \text{ else } \top_{D_1} \ldots \lambda x \,.\, \text{if } f(x) = 0 \text{ then } y_n \text{ else } \top_{D_n}) \\
&= (\lambda x \,.\, (\alpha^e_{\rho\{\alpha \mapsto y_1\}} \,.\, f)(x) \ldots \lambda x \,.\, (\alpha^e_{\rho\{\alpha \mapsto y_n\}} \,.\, f)(x)) \\
&= \mathit{RepCross}\ [(\alpha^e_{\rho\{\alpha \mapsto y_1\}} \,.\, f) \ldots (\alpha^e_{\rho\{\alpha \mapsto y_n\}} \,.\, f)]
\end{aligned}$$

Case 3.1: $\mathcal{D}(\rho(\alpha)) = D_\perp$ and $\rho(\alpha) = \perp$

$$\begin{aligned}
\alpha^e_\rho \,.\, f &= \lambda x \,.\, \text{if } f(x) = 0 \text{ then } \perp \text{ else } \top_{D_\perp} \\
&= \lambda x \,.\, \text{if } f(x) = 0 \text{ then } \perp \text{ else } \mathit{lift}(\top_D) \\
&= \mathit{RepLift}\ f\ (\lambda x \,.\, \text{if } f(x) = 0 \text{ then } \perp_D \text{ else } \top_D) \\
&= \mathit{RepLift}\ f\ (\lambda x \,.\, \alpha^e_{\rho\{\alpha \mapsto \perp_D\}}(f(x))) \\
&= \mathit{RepLift}\ f\ (\alpha^e_{\rho\{\alpha \mapsto \perp_D\}} \,.\, f)
\end{aligned}$$

Case 3.2: $\mathcal{D}(\rho(\alpha)) = D_\perp$ and $\rho(\alpha) = \mathit{lift}(y)$

$$\begin{aligned}
\alpha^e_\rho \,.\, f &= \lambda x \,.\, \text{if } f(x) = 0 \text{ then } \mathit{lift}(y) \text{ else } \top_{D_\perp} \\
&= \lambda x \,.\, \text{if } f(x) = 0 \text{ then } \mathit{lift}(y) \text{ else } \mathit{lift}(\top_D) \\
&= \lambda x \,.\, \mathit{lift}\ (\text{if } f(x) = 0 \text{ then } y \text{ else } \top_D) \\
&= \lambda x \,.\, \mathit{lift}\ (\alpha^e_{\rho\ \{\alpha \mapsto y\}}(f(x)) \\
&= \mathit{RepLift}\ (\top_{[E \to \mathbf{2}]})\ (\alpha^e_{\rho\ \{\alpha \mapsto y\}} \,.\, f) \\
&\quad \text{where } E \text{ is the source domain of } f
\end{aligned}$$

Whilst updating the environment ρ serves as a convenient specification of the $(\alpha^e_\rho \,.\, f)$ cases, a better implementation is obtained by making a special purpose group of functions. These explicitly pass around the "current value" of $\rho(\alpha)$, rather than updating ρ.

Notice also how there is no case for $\rho(\alpha)$ being bound to a function space value, and also no case for computing $(G^e_\rho \,.\, f)$ when G^e_ρ is itself a function space. Both cases were omitted from the implementation because they look difficult to implement. Furthermore they cover the relatively rare cases of instantiation of type variables to function valued objects, and more than one level of function spaces in the argument domains of a function respectively.

3.5 Relationship to *Conc*

If these cases do crop up, the implementation makes the instantiation using safe **Conc**, as defined in [HH91]. **Conc** is somewhat simpler to implement than the technique described above, and these strange cases are easily catered for. Hunt proposed the **Abs** and **Conc** Galois connections for quite a different purpose. As it happens, though, they form an embedding-closure pair and thus are a special case of Baraki's technique, in which $\mathcal{P}$ only has one element:

$$\mathcal{P} = \{\ \{v_1 \mapsto \bot_{A_1} \dots v_m \mapsto \bot_{A_m}\}\ \}$$

Because only one embedding-closure pair is used, and from the bottom of the instantiation lattices, the results are quite dismal, but the technique does at least allow the analyser to keep going under all circumstances. In the long run it will be necessary to extend the implementation to deal with these cases.

This relationship justifies the approach used in an earlier version of the system [Sew91], in which all polymorphic generalisation was done with **Conc**. At the time, although safe results were obtained, no formal justification was available.

4 Optimisations to the basic scheme

4.1 The first-order optimisation

Corollary 4.1 of Baraki's paper presents an optimisation which reduces the size of index set $\mathcal{P}$ in the first order case. Two minor problems need to be overcome.

Firstly, the conditions under which this applies, as specified by Baraki, are too lax. Not only must the function being generalised be first order, but the instances too must be first order. Secondly, this paper introduces the lifting construction $D_\bot$, which unfortunately invalidates the optimisation as originally presented. Nevertheless, it is possible to generalise the optimisation so it works for any first order domain expression at first order instances. Index set $\mathcal{P}$ is replaced by

$$\mathcal{P} = \{\ \{v_1 \mapsto a_1 \dots v_m \mapsto a_m\} \ \ | \, a_1 \in \mathcal{M}\ (A_1), \dots, a_m \in \mathcal{M}\ (A_m)\ \}$$

where $\mathcal{M}\ (A)$ is the set of meet irreducible points in A. When A is first order, $\mathcal{M}\ (A)$ is easily computed: see [Hun91] chapter 8. Importantly, $\mathcal{M}\ (A)$ is typically a much smaller set than **zapTop**(A), so the amount of work is drastically cut.

Proof

Let F be a first order domain expression, and A be a finite first order lattice. Let $\rho\{v \mapsto e\}$ denote that ρ binds v to e, regardless of any other bindings (warning: the same notation was used in Section 3.4 to denote something quite different), and let $a, b \in A$. Then from Baraki's paper we have

$$\begin{aligned} F^e_{\rho\{v \mapsto a \sqcap b\}} &= F^e_{\rho\{v \mapsto a\}} \sqcap F^e_{\rho\{v \mapsto b\}} & (1)\\ F^c_{\rho\{v \mapsto a \sqcap b\}} &= F^c_{\rho\{v \mapsto a\}} \sqcup F^c_{\rho\{v \mapsto b\}} & (2)\\ a \sqsubseteq b \ \Rightarrow\ F^c_{\rho\{v \mapsto a\}} &\sqsupseteq F^c_{\rho\{v \mapsto b\}} & (3)\end{aligned}$$

Now, if G is also a first order domain expression, we have

$$
\begin{array}{lll}
 & G^e_{\rho\{v\mapsto a\sqcap b\}} \cdot f_{\mathbf{2}\ldots\mathbf{2}} \cdot F^c_{\rho\{v\mapsto a\sqcap b\}} & \\
= & (G^e_{\rho\{v\mapsto a\}} \sqcap G^e_{\rho\{v\mapsto b\}}) \cdot f_{\mathbf{2}\ldots\mathbf{2}} \cdot F^c_{\rho\{v\mapsto a\sqcap b\}} & \text{From (1)} \\
= & (G^e_{\rho\{v\mapsto a\}} \cdot f_{\mathbf{2}\ldots\mathbf{2}} \cdot F^c_{\rho\{v\mapsto a\sqcap b\}}) \sqcap (G^e_{\rho\{v\mapsto b\}} \cdot f_{\mathbf{2}\ldots\mathbf{2}} \cdot F^c_{\rho\{v\mapsto a\sqcap b\}}) & \\
\sqsupseteq & (G^e_{\rho\{v\mapsto a\}} \cdot f_{\mathbf{2}\ldots\mathbf{2}} \cdot F^c_{\rho\{v\mapsto a\}}) \sqcap (G^e_{\rho\{v\mapsto b\}} \cdot f_{\mathbf{2}\ldots\mathbf{2}} \cdot F^c_{\rho\{v\mapsto b\}}) & \text{From (2) and (3)}
\end{array}
$$

So the binding for $a \sqcap b$ may be omitted provided those for a and b are included. Taking this to its logical conclusion, we only need to consider bindings of v to the meet-irreducible points in A.

4.2 Speeding up the computation of meets

Computation of $(f \,.\, F^c_\rho)$ consists of mapping F^e_ρ through the maximum-0 frontiers of f followed by recomputation of the minimum-1 frontiers using **min1FromMax0**. The latter operation is both expensive and redundant. A better approach is simply to throw away the minimum-1 frontiers, replacing them with dummy empty sets. The outermost meet is done on the basis of the maximum-0 frontiers alone. Only then are the final, corresponding minimum-1 frontiers reconstructed. Time is saved both because there is now only ever one batch of calls to **min1FromMax0**, rather than many, and because the meet operations only have to deal with maximum-0 frontiers, rather than both kinds of frontier.

4.3 Subsetting $\mathcal{P}$

The rather obvious lattice-theoretic observation

$$\forall S \,.\, \sqcap S \;\sqsupseteq p \quad\Longrightarrow\quad \forall S' \,.\, S' \subseteq S \Rightarrow \sqcap S' \;\sqsupseteq p$$

(where S is a set of lattice points) means a safe approximation to the instantiation sought is obtained using any subset of index set $\mathcal{P}$. As section 5 shows, although using Baraki's technique is vastly cheaper than analysing monomorphic instances directly, it can still be excessively expensive. Here is an opportunity to trade off quality of results against work.

Experimentation with the implementation reveals some surprising and pleasant behaviour. If time constraints do imply subsetting, the central question is what subset to select. An obvious choice is to work up from the bottom of the instantiation lattices $A_1 \ldots A_m$, until the required number of environments ρ have been created. But a better approach is to work down from the tops of these lattices. What appears to happen in the latter case is that "innermost" details emerge quite quickly. "Innermost" details are those which indicate, in a strict function which maps one data structure to another, that full demand on the output structure can be propagated to full demand on the input structure. This is particularly interesting for parallel evaluation, because it means sub-parts of the input data structure may be evaluated in parallel. Working up from the bottom of $A_1 \ldots A_m$ reveals "outermost" details first: for example, propagation of weak head normal form demand across a function. This is less useful in a parallel setting.

Section 5.2 gives a concrete example of these woolly notions. An interesting avenue for further investigation is whether there are any heuristics which could be used to pick a subset of $\mathcal{P}$ in an intelligent way.

Function $\Rightarrow$	`foldr/concat`		`zipWith3/zip3`	
Quantity $\Downarrow$	mono	poly	mono	poly
user time	2494.6s	3.28s	4474.6s	16.27s
max heap	1165.7k	104.5k	2443.9k	268.2k

Table 1: Monomorphic and polymorphic performance figures.

5 Results

5.1 Performance and quality

Two aims here are to find out how much speedup is obtained relative to a monomorphic analysis, and to see what effect the higher-order approximation has. The latter point is a little tricky because it means finding a first order function which can be defined using a non-basic instance of a higher-order polymorphic function. A famous example is the `foldr` and `concat` pair:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f a []     = a
foldr f a (x:xs) = f x (foldr f a xs)

concat = foldr (++) []
```

The Haskell `Prelude` provides a nice example with the `zipWith3` and `zip3` pair:

```
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
zipWith3 f (x:xs) (y:ys) (z:zs) = f x y z : zipWith3 f xs ys zs
zipWith3 f _      _      _      = []

zip3 = zipWith3 (\a b c -> (a, b, c))
```

The `foldr` results shown in Table 1 are quite striking, with the polymorphic analysis about 750 times faster, and a lot lighter on the heap too. As explained in [HH91], this is because the monomorphic analysis works with both type variables of `foldr` instantiated to $\mathbf{4}$ where $\mathbf{4} = (\mathbf{2}_\perp)_\perp$. This generates an argument lattice of almost 600000 points, compared to the 48 points of the simplest-instance argument lattice. A little consideration of the construction of $\mathcal{P}$ reveals it has 9 elements, so building an instance is quite quick.

An important question remains: how good is that instance? Well, it gives the same result for `concat` as the monomorphic analysis does. When printed out, the instance is textually about one-third the size of the monomorphic value, so it appears the approximation gives a much less detailed result, but the details omitted are not important in this case. Similarly, the `zipWith3/zip3` case produces identical results both ways. So it appears the approximations involved are not particularly drastic. In any case, first-order functions are always generalised exactly.

5.2 Effect of subsetting $\mathcal{P}$

As discussed in Section 4.3, working with a carefully selected subset of $\mathcal{P}$ can greatly reduce generalisation time without losing information which is interesting to a parallel implementation.

Let us consider `concat` at its simplest instance. The function is assigned domain $[\mathbf{6} \rightarrow \mathbf{4}]$ where $\mathbf{6} = (\mathbf{4}_\perp)_\perp$ and $\mathbf{4}$ is as above. Let the points in $\mathbf{4}$ be called $\{E_0 \sqsubset \ldots \sqsubset E_3\}$ where E_3 represents a value for which full evaluation is guaranteed to terminate, and those in $\mathbf{6}$ be called $\{E_0 \sqsubset \ldots \sqsubset E_5\}$ where E_5 similarly is the top element. The terminological similarity to Burn's Evaluation Transformers [Bur87] is deliberate, since these points represent evaluation transformers. The *backwards* abstract interpretation of `concat` is

Out	In
E_3	E_5
E_2	E_4
E_1	E_1
E_0	E_0

where the **In** column shows how much the argument to `concat` may be evaluated given that its output is demanded in the context listed under **Out**. The only fact of much interest in a parallel setting is that an output E_3 demand produces an E_5 demand on the input list. This allows the evaluator to spawn off tasks to evaluate each element of each of the argument sublists, which is potentially a lot of parallelism. The E_2 to E_4 demand propagation only allows the evaluator to evaluate the structure of the outer and inner lists, an activity which at best only generates as many tasks as there are elements in the outer list.

Now what happens as $\mathcal{P}$ shrinks, retaining elements from the tops of A_1 and A_2 until the very last?

$\lvert\mathcal{P}\rvert = 9$		$\lvert\mathcal{P}\rvert = 8$		$\lvert\mathcal{P}\rvert = 7$		$\lvert\mathcal{P}\rvert = 6$		$\lvert\mathcal{P}\rvert = 5$	
Out	In	Out	In	Out	In	Out	In	Out	In
E_3	E_5	E_3	E_5	E_3	E_5	E_3	E_5	E_3	E_5
E_2	E_4	E_2	E_4	E_2	E_4	E_2	E_4	E_2	E_4
E_1	E_1	E_1	E_1	E_1	E_1	E_1	E_0	E_1	E_0
E_0	E_0	E_0	E_0	E_0	E_0	E_0	E_0	E_0	E_0

$\lvert\mathcal{P}\rvert = 4$		$\lvert\mathcal{P}\rvert = 3$		$\lvert\mathcal{P}\rvert = 2$		$\lvert\mathcal{P}\rvert = 1$		**Conc**	
Out	In	Out	In	Out	In	Out	In	Out	In
E_3	E_5	E_3	E_5	E_3	E_5	E_3	E_5	E_3	E_1
E_2	E_2	E_2	E_0	E_2	E_0	E_2	E_0	E_2	E_0
E_1	E_0	E_1	E_0	E_1	E_0	E_1	E_0	E_1	E_0
E_0	E_0	E_0	E_0	E_0	E_0	E_0	E_0	E_0	E_0

All information is retained down to size 6, where the weak head normal form (E_1 to E_1) propagation is lost. At size 4, the E_4 from E_2 demand disappears. Remarkably, though, even at size 1, the E_3 to E_5 propagation remains. The entry headed **Conc** shows what happens when generalisation is done using **Conc**, as described in Section 3.5: E_3 demand only propagates E_1 demand, a far worse result.

This seems to suggest that even a single, well chosen embedding-closure pair gives better results than **Conc**. It will be interesting to see if this technology can be used to good effect in replacing uses of **Abs** and **Conc** for the purpose for which Hunt originally developed them: approximate fixed points [HH92].

One place in the Haskell `Prelude` where subsetting is essential to get any results at all in reasonable time is the definition of `partition`:

```
select p x (ts, fs)
   | p x        = (x:ts, fs)
   | otherwise  = (ts, x:fs)

partition p = foldr (select p) ([], [])
```

This requires an instance of `foldr` where both type variables are bound to $(\mathbf{4}\times\mathbf{4})_\perp$, giving overall type

$$\text{foldr}^{\#} \;::\; [[(\mathbf{4}\times\mathbf{4})_\perp \times (\mathbf{4}\times\mathbf{4})_\perp \rightarrow (\mathbf{4}\times\mathbf{4})_\perp] \times (\mathbf{4}\times\mathbf{4})_\perp \times (\mathbf{4}\times\mathbf{4})_{\perp\perp\perp} \rightarrow (\mathbf{4}\times\mathbf{4})_\perp]$$

Here $\mathcal{P}$ has 256 elements, but to get any results within, say, a couple of minutes requires using only the top ten or so. This is because computing meets in such a large function space is very expensive. It appears that computation of the outermost meet is the main expense of Baraki's algorithm. Nevertheless, it is a far better approach than the monomorphic analysis, whose performance would be quite unimaginably bad in this case.

Either way, it is pure insanity to try and analyse `partition` as it stands. A much better approach is to specialise `foldr` by substituting in `select`, which makes the really nasty higher-order parameter go away. A good optimising compiler might do this anyway.

5.3 Effect of first-order optimisation

How much effect does the optimisation of Section 4.1 have? The function

```
test4 = [ ([1],[2],[3],[4]) ] ++ []
```

binds the type variable of `(++)` to $(\mathbf{4}\times\mathbf{4}\times\mathbf{4}\times\mathbf{4})_\perp$ producing a $\mathcal{P}$ with 256 and 13 elements in the unoptimised and optimised cases respectively. A slight variant is

```
test6 = [ ([[1]],[[2]],[[3]],[[4]]) ] ++ []
```

for which the corresponding $\mathcal{P}$ sizes are 1296 and 21. Given such large differences, the time differences, shown in Table 2, seem suprisingly slight, although heap use differences are somewhat more distinctive. No satisfactory explaination is immediately apparent, and further investigation is needed. Nevertheless, some benefit is obtained from an optimisation which is trivial to implement.

Function ⇒	`test4`		`test6`	
Optimised ⇒	no	yes	no	yes
user time	11.20s	6.91s	145.23s	105.30s
max heap	374.4k	176.2k	1764.7k	587.1k

Table 2: First order generalisation, with and without optimisation.

5.4 Details of implementation

The algorithm was implemented in Standard Haskell 1.2, and compiled with Chalmers Haskell 0.997.5. Implementation of $D^e_\rho(x)$, $(f\ .\ F^c_\rho)$ and $(G^e_\rho\ .\ f)$ together with the mechanism to generate and use $\mathcal{P}$ is about 500 lines of code, with the specialised meet operation described in Section 4.2 contributing another 150. These figures do not include the code for computing minimum-1 from maximum-0 frontiers.

All timing figures were obtained using a Sun4-330 with 32 megabytes of real memory. Semispace size was 4 megabytes.

Acknowledgements

I would like to thank Gebreselassie Baraki for many invaluable comments, hints and tips. David Lester and Sava Mintchev also made useful comments.

Bibliography

[Abr85] S. Abramsky. Strictness analysis and polymorphic invariance. In H. Ganzinger and N.D. Jones, editors, *Proceedings of the Workshop on Programs as Data Objects*, number 217 in LNCS, pages 1–23. Springer-Verlag, 17–19 October 1985.

[AJ91] S. Abramsky and T.P. Jensen. A relational approach to strictness analysis for higher-order functions. In *Proceedings of the Symposium on Principles of Programming Languages*, January 1991. To appear.

[Bar91] G. Baraki. A note on abstract interpretation of polymorphic functions. In R.J.M. Hughes, editor, *Proceedings of the fifth ACM conference on Functional Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 367–378, Cambridge, Massachusetts, 26–30 August 1991. Springer-Verlag.

[BHA85] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher-order functions. In *Proceedings of the Workshop on Programs as Data Objects*, pages 42–62, DIKU, Copenhagen, Denmark, 17–19 October 1985. Springer-Verlag LNCS 217.

[Bur87] G.L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Imperial College, University of London, March 1987.

[CP85] C. Clack and S.L. Peyton Jones. Strictness analysis - a practical approach. In J.-P. Jouannaud, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 35–49. Springer-Verlag LNCS 201, September 1985.

[HH91] Sebastian Hunt and Chris Hankin. Fixed points and frontiers: a new perspective. *Journal of Functional Programming*, 1:91 – 120, January 1991.

[HH92] Sebastian Hunt and Chris Hankin. Approximate fixed points in abstract interpretation. In *Fourth European Symposium on Programming, Rennes, France*, 1992. LNCS 582.

[Hun91] Sebastian Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, Imperial College, University of London, 1991.

[HWe90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[KHL91] R. Kubiak, J. Hughes, and J. Launchbury. A prototype implementation of projection-based first-order polymorphic strictness analysis. In R. Heldal, editor, *Draft Proceedings of Fourth Annual Glasgow Workshop on Functional Programming*, pages 322–343, Skye, August 13–15 1991.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[PL92] S.L. Peyton Jones and D.R. Lester. *Implementing Functional Languages: A Practical Approach.* Prentice Hall International, Hemel Hempstead, UK, 1992.

[Sew91] Julian Seward. Towards a strictness analyser for haskell: Putting theory into practice. Master's thesis, University of Manchester, Department of Computer Science, 1991. Available as University of Manchester Technical Report UMCS-92-2-2.

[Wad87] P.L. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In S. Abramsky and C.L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis Horwood Ltd., Chichester, West Sussex, England, 1987.