Beyond Prototype Implementations: Polymorphic Projection Analysis for Glasgow Haskell

Julian Seward
sewardj@cs.man.ac.uk

Department of Computer Science
Victoria University of Manchester
Oxford Road, Manchester M13 9PL, UK.

Abstract. Building effective abstract interpreters requires careful attention to both theory and engineering issues. The theory of abstract interpretation is well developed, but the engineering side has fallen behind somewhat, an unfortunate event given that abstract interpretation is potentially a key technology for declarative language compilation. This paper shows the thinking behind the design and implementation of a first-order, polymorphic projection analyser for Haskell, capable of detecting head strictness, and therefore suitable for supporting all known strictness-related transformations. Development culminated in a successful trial installation in version 0.19 of the Glasgow Haskell Compiler, and the system has been used to analyse a range of Haskell programs, including an earlier version of itself, 13000 lines long.

1 Introduction

The past decade has seen a dramatic improvement in the performance of lazy functional languages, notably Haskell, as implementors have developed ways to map such languages efficiently onto stock architectures. Nevertheless, they still run an order of magnitude slower than their imperative counterparts, and, worse, it looks like the performance improvements are tailing off. Further increases in performance depend on finding ways to replace copying by update-in-place, and lazy evaluation by strict evaluation, when possible. A critical element in this is the semantic analyses which discover when such transformations are valid.

Successful use of strictness and sharing analyses in compilers demands careful consideration of the engineering constraints imposed by separate compilation, and by the limited computing resources available. Section 2 of this paper examines such constraints in detail, as a way of generating useful metrics with which to assess possibilities in the design space. Based on this, we outline in Section 3 the design of a polymorphic, backwards strictness analyser for Haskell. The analyser was installed in Glasgow Haskell 0.19, and shows an unprecedented price-performance ratio. Section 4 presents the results of this experiment and concludes. Although our primary concern is strictness analysis of Haskell, these ideas have wider applicability to all abstract-interpretation based analyses for functional languages, and should interest those concerned by the seemingly large gap between the theory and practice of abstract interpretation.

Real, usable analyses differ from prototype implementations in several ways. The analysis has to run in reasonable time and memory, even when presented with unreasonable inputs; it should "fit" properly into the existing compiler framework, vis-a-vis separate compilation, and it should do a good job for important language features, particularly sum-of-products types, polymorphism and higher-order functions. We shortly examine the consequences of these constraints in some detail.

The main focus of this paper is on global design tradeoffs. Based on that discussion, we outline a particularly effective polymorphic projection analyser for Haskell, and show some results from it. The analyser's design details are both interesting and voluminous, but not appropriate here, so references to more thorough treatments are given throughout – the first port of call is probably the author's PhD thesis [Sew94].

Architectural Overview This paper divides semantic analysis systems into two conceptual parts: an abstract interpreter, which generates recursive equations over some domains, and a fixpointer, which removes recursion from the equations, thereby making them useful to compilers. As usual, we require the domains to be of finite height to guarantee termination of fixpointing, and further restrict them to be distributive lattices to make fixpointing easier.

Several different schemes for fixpointing have been proposed. The Frontiers algorithm cleverly exploits monotonicity to build syntactically unique representations, thus trivialising the detection of fixpoints. Introduced by Clack and Peyton Jones [PC87], the algorithm has been much tuned over the years [Mar92, Hun91], and supports higher order fixpointing. A second line of enquiry is that of lazy fixpointing, where the fixpoint is only computed where needed, on demand. The precise origins of lazy fixpointing are debatable, but two early presentations are due to the Cousots [CC78] and to Jones and Mycroft [JM86]. Extensions to higher-order fixpointing are shown by Ferguson and Hughes [FH93], and by Rosendahl [Ros93]. The work of Hankin and Le Métayer on "lazy types" [HM94] is also related.

In this paper we take a third approach: computation of complete fixpoints by using a term rewrite system to transform semantically equivalent terms to syntactically identical normal forms. Consel describes a simple rewriter for two-point lattices used in Yale Haskell [Con91], and the technique was extended to other lattices in [Sew94], Section 5. Rewrite-based fixpointers cannot, in general, solve higher-order equations, but, as we shall see, this may not matter much.

Nocker [Noc93] shows a working analyser, based on "abstract reduction", where term-rewriting is used to execute the program using some abstract semantics. By comparison, our term-rewriting is used to detect fixpoints, and there is a clear distinction between the abstract interpretation and fixpointing activities.

The precise nature of abstract interpretations is determined, naturally, by the transformation to be supported. One particularly important characterisation is the lattices to which different types are mapped. In the simplest case, all non-functional types could be mapped to a two-point lattice, with the points representing no demand and weak head normal form (WHNF) demand respectively in strictness analysis. For a simple sharing analysis, we might use a 3-point chain, denoting no references, exactly one reference, and two or more references. Simple abstractions like these are routinely used in existing Haskell compilers [Con91, PJ93]. The simplicity of the abstractions makes these analyses cheap and reliable, so we shall not consider them further.

A more powerful analysis might attempt to give detailed information about functions dealing with structured types such as lists, trees, or whatever programmers care to define. Lattices abstracting such types grow in accordance with the complexity of those types, so there is no arbitrary cutoff point beyond which all details are ignored. In other words, there is a possibility of seeing arbitrarily far "inside" data structures. The downside is that these lattices can and do get very large, so the analysis becomes expensive. In this sense, semantic analysis is a particularly intractable problem, because the lattice sizes and thus amount of effort required can grow exponentially with complexity of types. For example, if an item of type `(Int, Int, Int)` is modelled by a 9 point lattice, $(2 \times 2 \times 2)_\perp$, adding just one more `Int` to the tuple effectively doubles the lattice size, giving $(2 \times 2 \times 2 \times 2)_\perp$. The costs of higher order semantic analyses are almost legendary, but even first order analyses can become excessively expensive. Getting reasonable performance sometimes requires "work limiting" – throwing away detail which abstract interpretation would otherwise pick up.

2 Constraints on viable designs

2.1 Separate Compilation

Curiously enough, separate compilation is the biggest single source of design constraints. A convenient way to view a multi-module program is as a directed, rooted, possibly cyclic graph, with arcs reflecting import dependencies, the `Main` module at the "top", and the Haskell `Prelude` at the bottom[1]. Compilation of a module `M` causes the compiler to read the source text of `M` and interface files of modules immediately below `M`, and emit code and interface files for `M`. Interface files are used by the compiler to communicate results of semantic analyses across module boundaries. In the absence of information about imported entities, there is always a safe assumption which can be made. This framework imposes several limitations:

- The interface files ought to be limited to a reasonable size, otherwise the compiler may spend a lot of time reading them. For example, Glasgow Haskell 0.19's `Prelude` interface is about 100k long, and reading it significantly slows compiler startup for small modules.

[1] It is convenient to assume all modules import the `Prelude`.

- Information only flows upwards, towards `Main`. For questions of the form "how is entity X going to be used?", we also need information flows downward from `Main`.
- To compile a module `M`, it should suffice to read the interfaces for modules immediately below `M`. We certainly want to avoid reading all the interfaces in the downward closure of `M`: for `Main`, that could mean reading the entire set of interfaces.

Mutually recursive modules Existing implementations deal with mutually recursive modules in two ways. One is to compile all modules in a group together, a clean solution, as with Yale Haskell's "compilation units". The other is to compile each module in the group separately, until interface files stabilise, denoting a fixed point. This means that at least one module will have to be compiled with zero semantic knowledge of the others, with the compiler making worst-case assumptions. The upshot of this is that, for semantic analyses involving fixpointing, the fixpoints being computed may be suboptimal.

2.2 Complete or minimal function graphs?

Fixpointing by minimal function graphs[2] (MFGs) builds the minimal set of argument-result pairs G needed to evaluate the fixpoint at a specific argument x_{init}. The initial graph is $\{(x_{init}, \top)\}$[3]. At each iteration, results for all arguments in the current graph G_n are re-evaluated, giving G_{n+1}. When an application of a function to some argument x is encountered, the existing graph G_n is searched for a corresponding result. If that x is not present, the result is taken as $\top$ and the pair $(x, \top)$ is inserted into G_{n+1}. The process repeats until G stabilises, whereupon $G(x_{init})$ delivers the required value. As an example, consider:

$$f \quad :: \; [\mathbf{2}_{\bot_\bot} \rightarrow \mathbf{2}_{\bot_\bot}]$$

$$f\ x = \begin{cases} \bot & \textit{if } x = \bot \\ \textit{lift}(\textit{lift}(\mathbf{0})) \sqcap (\textit{lift}(\bot) \sqcup (f\ \bot)) & \textit{if } x = \textit{lift}(\bot) \\ \qquad\qquad\quad x \sqcap (\textit{lift}(\bot) \sqcup (f\ x)) & \textit{otherwise} \end{cases}$$

Given $x_{init} = \textit{lift}(\bot)$, we have $G_0 = \{(\textit{lift}(\bot), \top)\}$. Evaluating $(f\ \textit{lift}(\bot))$ gives result $\textit{lift}(\textit{lift}(\mathbf{0}))$ and a new pair $(\bot, \top)$, so $G_1 = \{(\bot, \top), (\textit{lift}(\bot), \textit{lift}(\textit{lift}(\mathbf{0})))\}$. Iterating again gives $G_2 = \{(\bot, \bot), (\textit{lift}(\bot), \textit{lift}(\textit{lift}(\mathbf{0})))\}$ and $G_3 = G_2$, so $G(x_{init}) = \textit{lift}(\textit{lift}(0))$.

In this example, the required answer was obtained by evaluating f at two out of its four argument points. When the argument lattices contain thousands of points, the hope is that only a tiny minority of argument-result pairs need to be computed.

MFGs achieve "lazy" fixpointing because they make explicit the dependence between input and output points. Notice how the algorithm is limited to first

[2] A function's graph is no more than a collection of argument-result pairs for it.
[3] $\top$ if maximal fixpoints are sought, $\bot$ for minimal fixpoints.

order functions: higher order fixpointing would require comparing complete function representations, which is precisely what lazy fixpointing is designed to avoid. Nevertheless, by extending the notion of dependency between inputs and outputs, Ferguson and Hughes [FH93] have successfully generalised MFGs to work in the higher order case, calling them concrete data structures (CDSs), and Rosendahl reports similar results [Ros93].

Functional types abstract to domains with so many points that lazy fixpointing can make an enormous difference for higher order analysis. Ferguson and Hughes give figures for the famous `foldr-concat` example:

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)

concat :: [[a]] -> [a]
concat = foldr (++) []
```

where `(++)` is the Haskell list-append operator. Using a BHA-style forward abstract interpretation [BHA85] requires building an instance of `foldr` in the domain $[\mathbf{4} \rightarrow \mathbf{4} \rightarrow \mathbf{4}] \rightarrow \mathbf{4} \rightarrow \mathbf{6} \rightarrow \mathbf{4}$. This gives an argument lattice of almost 600,000 points. Computation of the entire function graph takes on the order of an hour using the frontiers algorithm [FH93], whereas with CDSs computing just those parts needed to determine the entire fixpoint of `concat` takes two or three seconds under similar circumstances. For more complex function spaces the difference is even greater.

Given such an overwhelming performance advantage, it is tempting to forget entirely about any other fixpointing scheme. Yet integration of CDS based fixpointing proves troublesome when viewed from a wider framework. There are two as-yet unresolved issues. Firstly, as discussed below, we have a deep desire to avoid monomorphic analyses of any kind. CDSs as presented so far are purely monomorphic. Because of the need to closely expose dependencies between subparts of the input and sub-parts of the output, it is not clear how they may be used in a polymorphic way. For first-order MFG-style fixpointing, polymorphism is not a problem: Kubiak [KHL91] has implemented a polymorphic projection analyser with MFG fixpointing in the Glasgow Haskell compiler.

A more immediate problem with lazy fixpointing is how to deal with modules. A crude scheme is to export whatever parts of the abstract function graph have been aggregated whilst compiling the defining module. If, compiling some other module, the compiler needs to know the fixpoint at some point for which no information is available, it is sometimes possible to make a safe estimate using monotonicity and the information which *is* available.

For example, having exported $G = \{(x_1, f\ x_1), (x_2, f\ x_2)\}$ we may wish to know $(f\ x_3)$ in the case where $x_3 \sqsubseteq x_1$ and $x_3 \sqsubseteq x_2$. Since f is monotonic it follows that $f\ x_3 \sqsubseteq f\ x_1$ and $f\ x_3 \sqsubseteq f\ x_2$, and so $f\ x_3 \sqsubseteq (f\ x_1 \sqcap f\ x_2)$. Provided we are working in a framework where overestimation is safe, this gives an approximate value for $f\ x_3$.

Clearly, the bigger the exported G, the better the estimates that can be made from it. But the whole purpose of MFGs is to keep G as small as possible, containing perhaps a handful of points out of thousands of candidates. This militates against making good approximations. Worse, not all the points in G may be used in making the approximation: in the example above, we are limited to using points $\{(y, f\ y) \in G \mid x_3 \sqsubseteq y\}$.

A second, equally unedifying solution is to export not only G, but the recursive domain equation from whence it came – or even the source text which begat the equation. Then, when G cannot answer a query, it is extended, in the manner described above, as required. If the abstract interpretations get large, dumping them into interface files may not be practical. This scheme also has a more subtle weakness. A crucial insight is that finding the fixpoint of a domain equation makes it stand on its own: it no longer references any other equation. Consider the following three modules defining recursive functions `f`, `g` and `h` with corresponding equations `f#`, `g#` and `h#`:

```
module F(f) where {             f = ... f ...       }
module G(g) where { import F;   g = ... g ... f ... }
module H(h) where { import G;   h = ... h ... g ... }
```

Dumping the unsolved domain equations into interface (`.hi`) files gives:

```
F.hi:   f# = ... f# ...
G.hi:   g# = ... g# ... f# ...
H.hi:   h# = ... h# ... g# ...
```

When module `H` is compiled it may be necessary to extend the function graph for `g#`. The trouble is this means reading interface `F.hi` even though the text of module `H` does not directly refer to `F`: in general, it might be necessary to read all interfaces in the downward closure of `H`. Any scheme which exports into interface files references to functions in other modules suffers the same problem. Merely inlining references to other equations does not help, as this makes the interfaces contain everything below them in the hierarchy, and hence very large:

```
F.hi:   f# = ... f# ...

G.hi:   g# = letrec f# = ... f# ...         in ... g# ... f# ...

H.hi:   h# = letrec f# = ... f# ...         in
             letrec g# = ... g# ... f# ...  in ... g# ... h# ...
```

Because of these difficulties, lazy fixpointing does not seem to fit well in an environment where separate compilation is the norm.

2.3 Polymorphic or Monomorphic?

Parametric polymorphism is an important element of the type systems of modern functional languages. Given a parametrically polymorphic function, we could:

– Redo the analysis for each different instantiation of the function. This is equivalent to expanding out the program into a collection of monomorphic instances before analysis.
– Analyse the function once, and use polymorphic invariance results to deal with differently instantiated uses later on.

An oft-quoted disadvantage of the former approach is that polymorphic functions may be analysed many times, which is wasteful. In the worst case, the number of instances can be exponential in the length of the program. Nevertheless, such behaviour does not appear to occur. Measurements by Mark Jones of six substantial programs in the `nofib` suite [Par92] suggest a twofold increase in the number of binding groups [Jon93]. Further, only simple list-handling functions like `map` and `foldr` have a large number of instances, and such functions are often inlined by optimising compilers anyway. So monomorphic analysis may not actually involve much duplication of work. Observe also that since the abstract interpretations of types often maps many types to the same lattice, monomorphisation on the basis of lattices causes even less expansion.

More serious is, once again, the interaction of monomorphisation with modules. For a module `M` exporting a polymorphic function `f`, we can either analyse `f` at all the instances it will get used at, or we can wait until all modules above `M` are compiled and reanalyse `f` on an as-needed basis. Both schemes are unattractive, the first because it requires a downward (`Main` to `Prelude`) pass through the module graph to establish instantiations, the second because of the need to carry around `f`'s source text when compiling modules above `M`.

Polymorphic analysis circumvents both problems because a single analysis serves all queries, even those from different modules. Work by Baraki, Hughes and Launchbury [Bar93, HL92b] (to mention but a few) has established techniques for both forward and backward polymorphic strictness analyses. Backward analyses lend themselves particularly well to such an approach: Section 5 of [Sew94] describes such an analysis in considerable detail. Baraki's work indicates how some forward analyses may be made polymorphic: [Sew94], Section 3 describes an implementation.

For first order functions, polymorphic techniques give the same results as monomorphic analysis: they are exact. But in the higher order case, accuracy may be lost. For backward analyses, which are inherently first-order, this may be acceptable. For other analyses, the magnitude of this problem has yet to be quantified.

2.4 Higher or first order?

Higher order fixpointing is difficult because the lattice sizes for function spaces grow so rapidly. For example, in strictness analysis, an item of type `[Int] -> [Int]` might abstract to domain $[\mathbf{4} \rightarrow \mathbf{4}]$, which has 35 points. Adding a second parameter of the same type gives domain $[\mathbf{4} \rightarrow \mathbf{4} \rightarrow \mathbf{4}]$, with 24,696 points, an increase of over 700 times. The number of points in $[\mathbf{4} \rightarrow \mathbf{4} \rightarrow \mathbf{4} \rightarrow \mathbf{4}]$ is certainly very large. Many functional parameters have types more complicated than this, and calculating complete fixpoints in their presence is impractical.

Fixpointing by term rewriting sometimes fails in the presence of functional parameters. In the general case, higher order equations cannot be solved, because their fixpoints depend on the fixpoints of the functional parameters. The usual method of rewriting adjacent approximations to normal form fails to detect an overall fixpoint because subsequent approximations contain some term involving a functional parameter applied more and more times. This occurs, for example, in fixpointing forwards abstractions of `foldr`:

```
foldr#(n)   = \f a xs -> ... f (f    ... (f E) ...   ) ...

foldr#(n+1) = \f a xs -> ... f (f (f ... (f E) ... ) ) ...
```

where `E` is some arbitrary term, and `foldr#(n)` and `foldr#(n+1)` denote two adjacent approximations to the fixpoint. Here, `f` is a functional parameter being wrapped more and more times around `E`, thereby making syntactic detection of a fixpoint impossible. Of course, if the term rewriter is extremely clever it can detect that, for some suitably large n, $\texttt{f}^n = \mathit{fix}\ \texttt{f}$, so it can replace `f` (*fix* `f`) by *fix* `f` and thereby generate normal forms which depend explicitly on `f`'s fixpoint. The required pattern matching is tricky – not so simple for doubly recursive terms – and we need to decide on a suitable n. The value of n depends on the function space in question – clearly, we need a quick algorithm for giving a good overestimate of the function space's height. Nielson and Nielson looked at this problem [NN92], but their work can give excessively large estimates and is of limited applicability. A more fundamental problem is polymorphism: in that case, n varies with the instantiation, so the notion of computing a single value for it is nonsense.

Nevertheless, for higher order equations which do not depend on the fixpoints of their functional parameters, term based fixpointing works well and can be thousands of times cheaper than using frontiers. Such equations can be generated, for example, from the forwards abstractions of `map` and `filter` ([Sew94], Section 4.3.5).

In defence of first order analysis This paper argues against higher order analyses. When modules are also taken into account, the number of design constraints becomes intolerable. Can such a step be justified? Well, taken in a wider context, maybe. Higher order functions defeat advanced code generation techniques, like arity check avoidance, unboxing and arguments in registers, since they represent calls to completely unknown functions. Augustsson [Aug93] suggests that calling a known function costs half as much as calling an unknown one.

The (Haskell) compiler community go to considerable trouble to get rid of calls to higher order functions. For example:

– Simple non-recursive combinators like `(.)` and `thenS` are inlined (in Glasgow Haskell).

– Prelude functions which pass functional parameters along unchanged, like `map` and `foldr`, are unfolded to form specialised versions (in Yale Haskell).
– Overloaded functions are specialised at their use instances, at least within single modules (Glasgow and Chalmers). This avoids a great deal of the higher order machinery which is often associated with implementations of overloading.

Nelan [Nel91] describes at some length techniques for what he termed *firstification*: the automatic removal of higher order functions. Although conversion of higher-order programs to first-order is, in general, impossible, programmers, by and large, use higher-orderness in certain idiomatic ways which *are* amenable to firstification using Nelan's machinery. This machinery, combined with aggressive inlining, can be remarkably effective. Even without using firstification, much of the optimised code dealt with inside Glasgow Haskell is purely first order.

From a strictness analysis point of view, first order analysis has two major advantages:

– Fixpointing is easier. The frontiers algorithm runs reasonably quickly for first order examples ([Sew94], Section 3), and term rewrite based fixpointing is guaranteed to work. Computing complete function graphs does not look quite so bad in the first order case, and doing so makes modules easier to deal with.
– Polymorphic generalisation gives exact results. Monomorphic analysis and its attendant module-related problems can be avoided entirely.

3 The Derived Design

3.1 Overview

Considering the design constraints, it looks like a first-order, polymorphic analysis computing complete fixed points might form a plausible basis for a working system, so this is what has been constructed. Abstract interpretation produces terms in what can be considered to be an abstract lambda-calculus, and fixpointing is done with a term-rewrite system which transforms semantically equivalent terms to syntactically identical normal forms.

The analyser forms part of the Glasgow Haskell compiler, and operates on a desugared Haskell form loosely referred to as Core. Core allows lambda terms, applications, literals, local bindings, one-level pattern matching (`case`s) and constructor applications. Type information supplied by previous compiler phases means the type (and hence the abstract domain) associated with every subexpression is known. The Core tree is annotated with the strictness information computed, making it available to subsequent transformation passes, and to the interface-printing machinery, so other modules can know about the strictness of functions in this one.

3.2 The abstract interpretation

Abstract interpretations for strictness analysis have historically been categorised as forward or backward [Hug90], although the appearance of relational analyses is now blurring that distinction. Backward analyses generate information about function arguments given knowledge of some property of the application as a whole. For example, backward strictness analysis of an n-argument function produces n maps, one for each argument, mapping demand on an application to demand on each argument. As this implies, the fundamental abstract entities are points in lattices, where different points denote different demands or "neededness" for some data structure. It is usual to (at least potentially) allow each different concrete type to have its own lattice of abstract demands, in order that we can do detailed analyses with data structures.

Although the first strictness analyses were of the forwards type, later development suggested backwards analysis might be quicker and give a simpler treatment of polymorphism, and practical work seems to bear this out ([Sew94], Section 5). The few papers showing how strictness information can be used to generate better code [PJ93, Hal93] all either suggest or imply that backwards information is what is actually useful. Taken together, the case for building a backwards abstract interpretation seems overwhelming.

Space limitations preclude much discussion of the abstract interpretation. Suffice it to say that it is fairly conventional: the overall structure is similar to that presented by John Hughes in [Hug90], although the mechanism for dealing with data structures, constructors and case terms is different. As with any purely backward analysis, higher-order analysis is impossible, so we naively assume that all unknown functions do not demand their arguments at all.

Many backward abstract interpretations go to a great deal of trouble to model data structures well, thereby inducing considerable complication in the machinery which deals with data structures – constructor functions and `case`s. Fortunately, the presentation can be simplified by the observation that, provided the abstraction of `case`s and constructor functions obeys certain constraints, it doesn't matter at all what they are. So the backwards abstraction can be specified without saying anything much about data structures. Subsequent parameterisations of it produce complete, workable abstract interpretations, and different parameterisations can produce interpretations giving very different levels of detail for data structures. This modular approach and some parameterisations of it are developed in [Sew94], Section 5; we outline one such parameterisation shortly.

An important feature of the abstraction is that suitable parameterisations produce polymorphically invariant interpretations. Polymorphic invariance facilitates separate compilation, as discussed in Section 2.3, and it also means the size of terms generated by (hence, the cost of) abstract interpretation is not related to the instantiation. This is in marked contrast to, for example, Wadler's original non-flat interpretation [Wad87], in which abstraction of `case` expressions could generate terms whose sizes grow exponentially with the size of the instantiating types.

Backwards strictness analyses work, in a sense, by detecting free variables in expressions. Because most variables do not appear free in most expressions, a naive analysis will tend to generate many huge terms which trivially reduce to $\perp$, the point denoting zero demand. Still, generating these things, then reducing them, takes effort, and it proves valuable to plant certain key rewrite rules inside the abstract interpreter itself to sidestep the process – a trivial modification to the code. Incorporating four of the most popular simple rewrite rules dramatically cuts the size of generated terms and so expands the range of source functions for which abstract interpretation can be done in reasonable time.

The Head-Strict Parameterisation Our implementation uses a powerful parameterisation designed to give detailed results for the vast majority of data declarations a Haskell programmer could reasonably use. This is in contrast to the often-limited range of types which other implementations can deal with, sometimes just lists [HH91], sometimes a fuller range of data declarations where constructor arguments may be one of the type variables, or a recursive call to the type, but nothing else ([Sew91], Section 6). The parameterisation is designed to detect head strictness. We wanted to move away from the prototype stage and create an analysis which (1) gives useful information for, effectively, any (non-functionally-typed) code fragment, and (2) generates information to support all known strictness-related transformations – hence the need for head-strictness. The result is that the parameterisation is rather complicated, so we will try and convey the essential ideas with an example.

The parameterisation derives from Burn's idea of evaluation transformers. [Sew91] generalised this to work for a wider range of data types, and [Sew94] further extended it to detect head-strictness. Let us consider lists. Lists abstract to three collections of points: bottom (`B`), denoting no evaluation, the set `W[a]`, denoting evaluation to weak head normal form, and then, if a cons cell is produced, evaluation of the first element with `a`, and the set `S[a][b]`, denoting evaluation of the complete structure of the list, evaluation of the first element with `a` and all the rest with `b`. If we model integers with the two evaluators `0` (no-eval) and `1` (eval), the lattice for `[Int]` becomes:

```
                S[1][1]
               /       \
          S[1][0]       S[0][1]
           /    \        /
      W[1]       S[0][0]
         \      /
          W[0]
            |
            B
```

For example, `W[1]` is a head-strict evaluator, evaluating to weak head normal form, and the head `Int` too, if possible. `S[0][1]` mean evaluate the entire list structure, and all `Int`s except the first. `S` (Structure) evaluators maintain the head/non-head element distinction so that, for example, when a list is known to be demanded both to `W[1]` and `S[0][1]`, we know we can demand the list to `S[1][1]` (which is `W[1]`$\sqcup$`S[0][1]`). In other words, this means the head-element information on a `W` (WHNF) evaluator is not (necessarily) obliterated when it is combined with an `S` evaluator.

The scheme generalises in several directions. Recursive types with more than one type variable are modelled with the same `B/W/S` collection of points, with the `W` and `S` points containing multiple parameterising points, rather than just one and two respectively as with lists. Lattices for non-recursive types omit the `S` points, but are otherwise identical. Constructors often have "constant" types – those not mentioning type variables – for arguments, or have arguments in which the type variables and recursive instances of the type appear buried inside arbitrary levels of "intervening" types. Many programs also employ mutually recursive types. All these cases can be handled, although they make the precise details of the parameterisation rather complicated. This parameterisation, and thus the derived interpretation, is polymorphically invariant.

3.3 Fixpointing

Once we regard the output of the abstract interpreter as a term in some "abstract lambda calculus", and we know we want to compare such terms for semantic equality, it seems natural to investigate the possibility of building a semantics-preserving transformation engine. We give only the barest summary here – for details, see [Sew94], Section 4. Some important points are:

– Terms are built from a small selection of constructions. Lambdas, variables and applications introduce and refer to variables, and apply abstract functions thus formed. "Constructional" terms denoting $\perp$, the lifting of some other term and tupling are used to build values in lifted and product lattices, and there are a corresponding set of "destructional" terms which test lifted-lattice points for $\perp$, undo liftings, and select components from tuples. Finally, joins and meets of terms are allowed.
– The terms are strongly-typed, using what amounts to an abstract version of the Milner-Hindley type system. Well-formed terms must be well-typed, and must also be monotonic with respect to their free variables. The type system incorporates a notion of parametric polymorphism. The abstract interpreter exploits this by mapping polymorphism in the source program onto polymorphism in the abstract terms, which gives polymorphic analysis essentially for free.
– The rewriter proper consists of a scheduling mechanism, which selects redexes and applies rewrite rules to them, and the rewrite rules themselves. The scheduler determines the reduction order, which has a major bearing on performance. There are about fifty rules, many with complex side conditions. Since the rules are coded in Haskell, they can use the full power of Haskell pattern-matching. The rewriter is complicated somewhat by the need to propogate and use partial knowledge (ie, is/is not $\perp$) of the values of subterms in order to reach normal form.

Making the rewriter work well Once a normal form has been decided on and suitable rewrite rules generated, there is still the issue of how the rules are to be applied. This boils down to a probably-irreconcilable conflict between innermost-first and outermost-first rewriting. Both strategies work well much of the time, but both also suffer from pathological interactions between terms emitted by the abstract interpreter and some of the rewrite rules. In these cases, rewriting can take a small term and expand it exponentially to a gigantic intermediate term before employing some other rule to shrink it to another small term. The effect on performance is disastrous. Getting round this problem requires a hybrid reduction strategy which tries to avoid known pathological cases. This works well, although deciding on the split requires a great deal of detailed investigation into the precise dynamics of the reduction system as applied to realistic workloads.

Watching the rewriter at work revealed another interesting, but, in retrospect, unsurprising fact: the abstract interpreter's "vocabulary" of term-combinations emitted is particularly limited. The same small collection of idioms occurs over and over again. Some of these idioms are extremely common, but they are also quite big. Big terms are a performance liability; consistently small terms tend to give much better performance. That makes it attractive to capture such idioms in special, new terms, and adjust the rewrite rules accordingly. This should be done sparingly, since the term rewriter becomes complicated by having to deal with multiple representations of the same term. Still, capturing just two common idioms like this has given performance gains very roughly on the order of 25 times for some inputs.

Even with this trickery, the abstract interpreter can sometimes emit terms so large they swamp the rewriter. In this situation it is helpful to devise a way to prune terms so the most important information is retained, whilst dramatically shrinking the term. Of course, defining what information is important and how safety is preserved depends entirely on the abstract interpretation. For this kind of strictness analysis, it is safe to replace any term by $\perp$, since $\perp$ characterises "no demand". The scheme employed prunes terms after a specified depth of nested `W` or `S` constructions has been seen. This depth-based pruning, whilst crude, is frequently effective, but does not work well for very wide, shallow terms. Further investigation into cleverer pruning strategies, and their relationship to widening, could be worthwhile.

3.4 Interfacing the abstract interpreter and fixpointer

All implementation was done in the lazy functional language Haskell. Performance profiling revealed an important fact: how well the system as a whole works depends heavily on how much of the abstract interpreter's output the fixpointer (hence, the rewriter) needs to see. If the fixpointer is very clever it may ignore most of the interpreter's output. But since writing the interpreter in Haskell makes it demand-driven, cleverness on the fixpointer's part saves a great deal of expense in the interpretation itself. It seems safe to say that the performance results reported below could never have been achieved, nor even approached, had the interpreter been written in a strict language, SML, for example. Futher details are available in [Sew94], Section 4.

The comments three paragraphs back regarding idioms and the term rewriter belie a more fundamental design flaw which limits performance: serious and extensive sharing losses. The idiom-spotting trick gives big performance speedups primarily because it avoids duplication of arbitrarily large subterms. Many of the cases of the abstract interpreter duplicate terms indiscriminately. All this points to a need for explicit sharing in terms: non-recursive abstract `let`s. Modifying the term rewriter to work with `let`s sounds less than straightforward, so it might be helpful to borrow key ideas from the call-by-need lambda calculus described in [AFM$^+$95].

Good performance hinges on generating small terms and keeping them small throughout their various manglings. Ideally, the size of the abstract interpreter's output should be proportional only to the size of the source program. This is probably unrealistic. Nevertheless, designers of abstract interpretations would be well advised to strive towards this goal. As an example, Phil Wadler's original non-flat strictness analysis scheme [Wad87] gave an interpretation of `case` expressions which expands exponentially with the size of instantiating types. This renders it impractical for all but the tiniest programs ([Sew94], Section 5.8). Our backwards framework is polymorphically invariant, so it avoids that particular horror, but it still has plenty of potential for optimisation using abstract `let`s. The point of this example is that, with a little care, much of the "exponentialness" can be engineered away, giving substantial performance benefits.

4 Results

4.1 In the small

The technology described above was constructed inside a purpose-built development framework. The framework is a quick-and-dirty implementation of the front part of a compiler for an overloading-free subset of Haskell, designed to facilitate experimentation. A higher-order removal transformation in the style of Nelan [Nel91] allows first-order analyses of programs making substantial use of higher-order functions. All analyses were done monomorphically, to make comparison with a forward interpretation fairer. For the backwards analyses, of course, there is no intrinsic reason to use monomorphic analysis. Four inputs were used:

– `avlTree`: insertion of items into an AVL tree (44 lines).
– `life`: Launchbury's implementation of Conway's "life" simulation (311 lines).
– `fft`: Fast fourier transform, from Hartel's benchmark suite [HL92a] (421 lines).
– `ida`: Solves the 15-puzzle, also from Hartel's suite (728 lines).

`life`, `fft` and `ida` are substantial inputs. Quoted sizes are after insertion of suitable prelude functions. They make substantial use of higher-order functions,

	Forwards		**EvalTrans**		**HeadStrict**	
Program	Time	Resid	Time	Resid	Time	Resid
`avlTree`	9.58	1514	0.23	*197*	0.27	*200*
`life`	359.20	11203	2.01	628	2.97	*615*
`fft`	28.76	*1473*	6.38	*1473*	8.38	*1524*
`ida`	920+	54000+	9.32	*1725*	7.70	*1734*

Table 1. Performance of abstract interpreters (time in seconds, residencies in Kbytes). Italicised figures indicate peak residencies defined by preprocessing phases, rather than abstract interpretation or fixpointing. The **Forwards** analysis of `ida` did not complete even in 54 megabytes of heap.

which the firstifier has to work quite hard to remove, and which causes the input to the abstract interpreter proper to be considerably bigger than the untransformed sources.

The analyser was written in standard Haskell 1.2, compiled with Glasgow Haskell 0.19 "`-O`", and run on a quiet 64-megabyte Sun Sparc 10/31 running SunOS 4.1.3. A generational garbage collector was used, and heap sizes were set to try and keep garbage collection costs below 10%, although for larger residencies this is difficult. Times under sixty seconds are averaged over six runs. They include the preprocessing times of parsing, desugaring, typechecking, monomorphisation and firstification, but in no case do these exceed 20% of the total. Table 1 shows the results, using three different abstract interpretations:

– **HeadStrict**: a head strict, backwards interpretation precisely as described in this paper.
– **EvalTrans**: another backwards interpretation, formed, as with HeadStrict, using the generic framework described above, but using a simpler characterisation of data structures, essentially a generalisation of Geoff Burn's Evaluation Transformers.
– **Forwards**: a forwards BHA-style interpreter, with handling of data structures as described in [Sew91]. The data-structure handling can be considered in a sense of "equivalent" detail to that of EvalTrans.

Each interpreter was measured with a rewrite-based fixpointer designed specially for it. The rewriters are very similar, and by using the same kind of optimisations in each we hope to make the comparison reasonably fair.

These measurements make clear just what an advantage the backwards interpretations have – the EvalTrans analysis of `life` runs almost two hundred times faster than the forwards analysis, and in a fraction of the space. The HeadStrict interpretation is not significantly slower than the EvalTrans version. This comes as a bit of a surprise, since one might expect it to produce much more detail, and take correspondingly longer. Inspection of outputs reveals that it doesn't always find a great deal more strictness than the EvalTrans analysis. And quite why the HeadStrict analysis should sometimes be quicker (for `ida`) is mysterious, possibly due to the precise details of reduction paths in the term rewriters.

4.2 In the large

After extensive tuning in the development framework, the (polymorphic) HeadStrict interpreter and its fixpointer were installed in Glasgow Haskell 0.19 to assess the viability of such strictness analysis “in the large”. Integration added about 5700 non-blank, non-comment lines of code to an existing total of about 50000. Because the compiler is well designed, integration was remarkably clean, with the smallest of modifications to existing code. The existing framework for transmitting pragmatic information between modules was extended to allow full intermodule analysis; given the arguments presented in Section 2, the experiment would have been largely worthless had this been omitted.

Compiling Glasgow’s implementation of the Haskell `Prelude`, plus some other general-purpose libraries, totalling about 4000 lines, takes some four hours on a Sun SparcStation IPX, roughly a doubling of the no-analysis time. A heap size of 24 megabytes proved more than adequate, and, pleasingly, there was no need to resort to the tree-pruning described in Section 3.3. An earlier version of the analyser, in its development framework, constituting 13000 lines in 28 modules, was subsequently compiled. 25 modules went through with no problems, whilst three required pruning.

By any standards, such experimentation with real compilers and real programs lends major credibility to the engineering described in this paper. Observation of the interpreter and fixpointer grappling with non-toy inputs lead to many refinements, and to the suggestions for further development towards the end of Section 3.4.

4.3 Conclusions

The abstract interpreter and fixpointing mechanism outlined above have been developed and tuned by running them over approximately twenty thousand lines of Haskell source code. Assessing the viability of new schemes only on small examples or using paper analyses can be misleading. Good engineering requires not only supporting theory – of which the field shows no shortage – but also extensive quantitative analysis on realistic-sized inputs. The relative lack of effective implementations of strictness and other semantic analyses for Haskell highlights the need for further attention to engineering issues in abstract interpretation.

Using these principles, a polymorphic projection analyser for Haskell has been created. The analyser is effective, robust, reliable, turns in good performances, and has been used to support research in automatic strictification of Haskell programs[4]. It also lays out a design with considerable potential for refinement, particularly with respect to building more economical abstract interpretations, and to fixing sharing problems in the fixpointing mechanism. Further development may yield impressive strictness and sharing analyses for Haskell.

[4] Denis Howe, Department of Computing, Imperial College, London.

References

[AFM$^+$95] Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *22nd Symposium on Principles of Programming Languages*, San Francisco, California, January 1995. ACM Press.

[Aug93] Lennart Augustsson. Implementing haskell overloading. In *Proceedings of the Functional Programming Languages and Computer Architecture Conference, Copenhagen, Denmark*, June 1993.

[Bar93] Gebreselassie Baraki. *Abstract Interpretation of Polymorphic Higher-Order Functions*. PhD thesis, Department of Computer Science, University of Glasgow, Lilybank Gardens, Glasgow G12 8QQ, UK, February 1993. Also available as a Glasgow Tech Report FP-93-07.

[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.

[CC78] Cousot and Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*. North-Holland, 1978.

[Con91] Charles Consel. Fast strictness analysis via symbolic fixpoint iteration. Unpublished. Yale University, Department of Computer Science, September 1991.

[FH93] Alex Ferguson and John Hughes. Fast abstract interpretation using sequential algorithms. In *Proceedings of the Chalmers Programming Methodology Group Winter Meeting*, January 1993. Department of Computer Sciences, Chalmers University of Technology.

[Hal93] Cordelia V. Hall. Using strictness analysis in practice for data structures. In John T. O'Donnell and Kevin Hammond, editors, *Glasgow Workshop on Functional Programming*, Workshops in Computing, Ayr, Scotland, July 1993. Springer-Verlag.

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

[HL92a] Pieter H. Hartel and Koen G. Langendoen. Benchmarking implementations of lazy functional languages. Technical report, Department of Computer Systems, Faculty of Mathematics and Computer Science, University of Amsterdam, December 1992.

[HL92b] John Hughes and John Launchbury. Projections for polymorphic first-order strictness analysis. In *Mathematical Structures in Computer Science*, volume 2, pages 301–326, 1992.

[HM94] C. L. Hankin and D. Le Métayer. Deriving algorithms from type inference systems: Application to strictness analysis. In *Proceedings of POPL'94*, 1994.

[Hug90] John Hughes. Compile-time analysis of functional programs. In David A. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley Publishing Company, 1990. From the 1987 Year of Programming, University of Texas, Austin, Texas.

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

[JM86] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs: Abridged version. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 296–306, 1986.

[Jon93] Mark P. Jones. Partial evaluation for dictionary-free overloading. Technical Report RR-959, Department of Computer Science, Yale University, April 1993.

[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.

[Mar92] Christopher Charles Martin. *Algorithms for Finding Fixpoints in Abstract Interpretation.* PhD thesis, Imperial College, University of London, June 1992.

[Nel91] George C. Nelan. *Firstification.* PhD thesis, Arizona State University, 1991.

[NN92] F. Nielson and H.R. Nielson. Finiteness conditions for fixed point iteration. Technical report, Computer Science Department, Aarhus University, Denmark, February 1992.

[Noc93] Eric Nocker. Strictness analysis using abstract reduction. In *Proceedings of FPCA93, Copenhagen, Denmark*, June 1993.

[Par92] Will Partain. The `nofib` benchmark suite of haskell programs. In *Fifth Annual Glasgow Workshop on Functional Programming, Ayr*, 1992.

[PC87] Simon Peyton Jones and Chris Clack. Finding fixpoints in abstract interpretation. In S. Abramsky and C.L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Computers and Their Applications, chapter 11, pages 246–265. Ellis Horwood, 1987.

[PJ93] Will Partain and Simon L. Peyton Jones. Measuring the effectiveness of a simple strictness analyser. In John T. O'Donnell and Kevin Hammond, editors, *Glasgow Workshop on Functional Programming*, Workshops in Computing, Ayr, Scotland, July 1993. Springer-Verlag.

[Ros93] Mads Rosendahl. Higher-order chaotic iteration sequences. In *PLILP'93*, pages 332–345, Tallinn, Estonia, 1993. Springer-Verlag LNCS714.

[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.

[Sew94] Julian Seward. *Abstract Interpretation: A Quantitative Assessment.* PhD thesis, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK, October 1994.

[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.

This article was processed using the LaTeX macro package with LLNCS style