Strictness Analysis à Grande Vitesse: Rewriting the rules of the Evaluation Transformers game

Second draft: October 19, 1995. Comments are welcomed.*

Julian Seward
Department of Computer Science, University of Manchester, M13 9PL, UK
sewardj@cs.man.ac.uk

Abstract

This paper presents the design of a strictness analyser primarily intended to assist in the parallelisation of Haskell programs. The analyser generates detailed evaluation transformer information suitable for use in a parallel graph reduction machine. By tuning the design to handle common cases rapidly, we achieve good performance over a range of realistic test inputs, of sizes up to a thousand lines. Performance figures are presented. From measuring the time it takes Glasgow Haskell 0.10 to compile those same programs, it is apparent that the analyser is sufficiently quick to be approaching the point where incorporating it into a Haskell compiler is a viable proposition.

Our strictness analyser contains three major innovations. Firstly, we employ a mixed forward and backwards analysis, which allows us to do projection analysis even in the presence of higher order functions. Secondly, we solve the resulting recursive domain equations using a sophisticated term rewriting system, which, as far as we know, has abilities far beyond previous rewrite based solvers in abstract interpretation systems. Finally, we avoid many of the difficulties of higher order analysis by automatically transforming out most higher order functions before analysis begins.

1 Introduction

Higher-order, non-flat strictness analysis techniques have acquired a reputation for being too expensive for use in production compilers. This is a pity, since many such functions behave in relatively simple ways, which suggests their abstract behaviour could be divined without much difficulty.

One of the main themes which characterised the past decade's advances in hardware and software performance is that of measurement-lead design. This school of thought bases design on measuring what programs do most often, and making these common activities as fast as possible, even at the expense of slowing down less common operations. These ideas, elaborated in Hennessy & Patterson [HP90], played a central role in the RISC revolution, and have been taken up recently by the functional programming community [PHHP93] [Par92].

We present, in detail, the design of a fast strictness analyser for Haskell, dealing with higher-order functions and generating evaluation transformer information suitable for use in parallel graph reduction systems [Bur87] [Bur91]. Preliminary experiments have been encouraging. For example, running in compiled Haskell on a Sun Sparc-10, a 618 line program was analysed in 44 seconds. Of this, roughly half was devoted to parsing, desugaring and typechecking, expenses which any compiler would incur. As a naive first implementation, there is ample scope for performance improvement.

This paper argues in favour of a measurement-lead approach for the implementation of semantic analysis techniques for functional languages. A major emphasis is to feed the analyser the kinds of programs people really write, rather than basing design decisions on the usual Mickey Mouse examples occurring so frequently in papers on the subject. To this end, test inputs have been taken from the benchmark suites of Hartel [HL92] and Partain [Par92].

Such an approach reveals some interesting facts. For example, we discovered the main limitation on analyser performance was not the necessity to iterate to and detect fixpoints, as had been assumed by so many theoretics. Rather, it was the sheer size of the terms generated during the abstract interpretation phase. By introducing a little more intelligence into that phase, term size is cut dramatically, giving a corresponding performance increase.

Similarly, polymorphism is dealt with by the crude mechanism of a monomorphisation pass before analysis. Previous workers, including myself, assumed this was a "bad thing" (see [HH91] and [Sew93]), and put much effort into devising polymorphic analysis methods [Bar91]. Yet the feared code explosion, it seems, simply does not happen. Measurements by Mark Jones on a 13000 line Haskell program – the source code of this analyser – reveal that only a very few polymorphic functions, for example `map` and `foldr`, are used at a large number of instances. Certainly, monomorphisation does not cause any noticeable performance problems for this analyser. A much better reason for disliking monomorphic analyses is that they significantly complicate life when modules appear on the scene.

*My apologies for the dog-rough typesetting. This will be fixed in later versions of the paper.

1.1 Overview of paper

The remainder of the introduction is devoted to an overview of the analyser, henceforth referred to as **Anna**.

Section 2 deals with technical preliminaries. In particular, we examine how types in the source program are mapped to abstract domains suitable for the control of parallel evaluation.

The abstract interpretation used forms the subject of section 3. A variant of the projection analysis described in section 6 of [Hug87], the technique is a mixed backwards and forwards analysis, with the aim of performing backwards analysis. By supplying just enough "forwards" information to turn functions into first class citizens, the analysis deals sensibly with higher-order functions, partial applications and functions inside data structures.

Section 4 presents, in detail, the elaborate term rewriting system used to detect fixed points in the recursive domain equations generated by the abstract interpreter.

Functional programmers have long observed that large parts of the programs they write can be mechanically transformed to have no higher-order functions. Anna exploits this to good effect, transforming away as many higher-order functions as possible before analysis. Common higher order functions which encapsulate particular forms of recursion, such as `foldr`, are trivially removed. Even more difficult forms, like the monads so beloved in certain quarters [Wad92] are transformable. Nevertheless, some difficulties remain. Section 5 discusses all this in detail. We also look briefly at the monomorphiser.

Finally, in section 6, all these goodies are drawn together with a discussion of the system's performance, and of related and further work.

1.2 Overview of the analyser's front end

Anna is a large Haskell program, consisting of more than 13000 lines. Operation is simple. Anna reads a source program on the standard input, and performs extensive transformations on the program, printing it out at various points on the way. Finally, the strictness information is generated and printed.

The language accepted is a subset of Haskell. Missing features are anything to do with overloading, modules or arrays. Because of these, offering up arbitrary Haskell programs for analysis is a difficult task: all overloading has to be resolved by hand, a tedious business. The lack of module support does not prove much of a problem, since multimodule inputs are simply concatenated into one massive program, modulo solving the odd renaming problem along the way. Anna knows nothing about the Haskell prelude, and the relevant parts of this too need to be inserted into the inputs. A very few operations are taken as primitive: `(+)`, `(-)`, `(*)` and `(/)`, all of type `Int -> Int -> Int`, comparisons `(<)`, `(<=)`, `(==)`, `(/=)`, `(>)` and `(>=)`, of type `Int -> Int -> Bool`, and conversion functions `chr` and `ord` of type `Int -> Char` and `Char -> Int` respectively. A valid program must supply a binding for `main`, but unlike a Haskell program, this may be of any type.

Despite this meagre collection of primitives, Anna knows about most of the built-in Haskell types, including booleans, characters, strings, lists and tuples. Although some important features of Haskell are missing, the subset allows Anna to be fed real-world programs of considerable complexity, albeit after some considerable massaging.

Probably the best way to think of Anna is as a framework for trying out new analysis techniques. Hence, the system logically consists of two parts: the analysis proper, and the supporting framework. The interface between the two is reasonably clean, so changing the nature of the analysis can be done without much upheaval. This section focusses on the supporting framework.

Because we want to exercise the analyses on functional programs of realistic size, the supporting framework is necessarily large and complex. Indeed, the analysis part is currently the smaller of the two. The framework contains a goodly part of what one might expect to find in a full-scale compiler for the same language:

- Following the parsing stage, desugaring and pattern matching transformations are carried out. These produce **Core**, a minimal functional language used as an intermediate form in the Glasgow Haskell compiler [PHHP93], and typical of the intermediate forms of various other compilers, for example the Chalmers Haskell-B Compiler [Aug87]. All further transformations prior to strictness analysis proper are Core-to-Core transformations.

- A dependancy analysis phase splits the program up into minimal mutually recursive groups, and marks non-recursive bindings as such. All subsequent transformations are required to maintain dependancy order.

- A crude but effective Core simplification pass removes unused bindings, and substitutes in constant bindings only used once. This helps to clean up the rather messy output of the desugarer. The former feature is useful for debugging the analyser. Because a binding for `main` must be supplied, the simplifier will eventually remove all bindings not reachable from `main`. If the analyser is seen to malfunction, arbitrary subsections of the input program can be discarded simply by changing the body of `main`, until what remains is small enough to make debugging viable.

- Removing nested environments makes subsequent transformations and analyses simpler. To this end, the program is flattened out by a modified Johnsson-style lambda-lifter [Joh85], followed by another dependancy analysis pass.

- The program is now typechecked, using a standard Milner-Hindley inferencer derived from Chapter 9 of Peyton Jones' book [Pey87]. Every node in the Core tree has a type expression attached. Although a complete annotation is rather expensive, it is essential for subsequent passes.

- The single most complicated transformation, higher-order function removal (also known as specialisation or firstification) now follows. The present naive implementation, described in Section 5, is slow but correct. Most if not all of the higher-orderness of typical programs can be removed. This transformation is complicated by the need to maintain type annotations correctly.

- Finally, the program is monomorphised. This pass is quick and relatively painless, even though a third trip through the dependancy analyser is subsequently required.

Most compilers would want to mangle the output of the desugarer in quite different ways to generate good code. Fortunately, it is easy to see how the output of the strictness analyser proper pertains to the desugared program. Only two transformations give much trouble:

- Lambda-lifting simply moves bindings from inner levels to the top level, and adds extra parameters. With a little bookkeeping, it is possible to keep track of where nested bindings ended up, so that strictness information can be related back to them.
- Higher-order function removal will only ever remove higher-order functions which have become irrelevant because of specialisation. All first-order functions are preserved. We are really only interested in deriving evaluation transformers for the first order functions. This is because the demand propagated across a higher-order function largely depends on what the higher-order parameter is. So exploiting demand propagation across higher-order functions means runtime manipulation of evaluation transformers, a serious complication for parallel graph reduction systems.

Building and maintaining the framework is a tiresome, time consuming task. One could also argue all that effort was unnecessarily, because the Glasgow Haskell team have specifically designed their compiler as a basis for experiments like this, and valiantly supported those brave enough to take them up [PHHP93]. In retrospect, there are three reasons why Anna was not built into Glasgow Haskell:

1. At the time work on Anna begun, in the summer of 1991, Glasgow's compiler (version 0.02) was in still in the process of development.
2. Until recently, the analyser was relatively feeble, so the need to feed it realistic Haskell programs has only recently arisen.
3. The most important reason, though, is this: Anna had been developed using Mark Jones' marvellous interactive environment, Gofer. Merging Anna into the Glasgow Haskell world would have meant compiling with a Haskell compiler and this would easily have put an order of magnitude on the edit-compile-run cycle time.

As Anna becomes more and more powerful, the incentive to build it into a real compiler grows. This is definitely a long term objective.

2 Technical preliminaries

2.1 Some terminology

The analyser's front end produces a **Core syntax tree**, in which every node is decorated with its type. This is fed to the **abstract interpreter** proper, which translates to an abstract form: **recursive domain equations**. The **fixpointer** solves these equations by iterating to their greatest fixed points, detecting equality of adjacent approximations by reducing them to **normal form** using the **term rewriting system**, and comparing those normal forms.

There are two kinds of abstract entity.

- **Contexts** denote an amount of evaluation that should be applied to a data structure or function. These are sometimes referred to as **demands** or **backwards values**, but we will stick with **context** where possible. We later introduce a Haskell type `Context` to model contexts.
- **Abstract values** amount to some trickery we will introduce to deal with higher order functions. An alternative name, which is again avoided where possible, is **forward value**. The corresponding Haskell type is `AbsVal`.

This paper is primarily concerned with discovering how source language functions behave viz-a-viz contexts. Nevertheless, the output of the abstract interpreter is one abstract value per Core function. Contexts and abstract values intertwine, so the `Context` and `AbsVal` types are mutually recursive. The abstract interpreter itself is defined as the function `Z` in section 3.6.1.

Contexts and abstract values are, in a sense, strongly typed. Each context is a member of a particular **context domain**, and most operations on contexts are only meaningful if their operands are drawn from particular domains. Abstract values are also strongly typed. Although the domains for abstract values are, strictly speaking, different from context domains, we will ignore abstract value domains. Instead, we only consider context domains, henceforth referred to simply as **domains**, and pretend that for each domain there is a family of contexts, and a family of abstract values.

For each Milner-Hindley type, there is a corresponding domain. In general, there may be many different types which map to the same domain. The next section defines, informally, this mapping. We then refine the mapping slightly in section 2.2.4, and formalise it in section 2.2.5.

2.2 Domains for projection analysis

A primary aim of these analyses is to generate information useful for exploiting a parallel machine. To this end, we use domains which are best viewed as a generalisation of the evaluation transformers introduced by Burn [Bur87]. These are introduced by example.

2.2.1 Base types

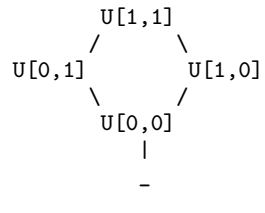
Base types `Int` and `Char` are mapped to a two point domain `2 = {0, 1}`, with `0` meaning "do not evaluate this" and `1` meaning "evaluate fully". In this case only, full evaluation is the same as evaluation to weak head normal form (WHNF).

2.2.2 Non-recursive structured types

Consider the interpretation of a familiar non-recursive structured type: `(Int, Int)`. We need to model the evaluators for the components of the pair separately, so there must be a product involved: `(2 x 2)`. An evaluator corresponding

to any such point would first have to evaluate the pair closure to WHNF, so it could get its hands on the individual components. So we really need a fifth point representing an evaluator which does nothing at all. The overall interpretation is `Lift (2 x 2)`.

At this point it is convenient to introduce a notation for points to be used throughout this paper. The bottom point of the above domain is written as an underscore, `_`. The other four are written in the form `U[x, y]` where the `U` stands for "go up the `Lift`", and the `x` and `y` are the relevant product components. The overall collection of evaluators is thus written `{_, U[0,0], U[0,1], U[1,0], U[1,1]}` with the following ordering:



How does this generalise to arbitrary non-recursive structured types? Well, very simply. A non-recursive structured type is modelled by the single lifting of the product of whatever its type variables are bound to. Further details are irrelevant. That's because we observe the guiding rule that all objects corresponding to a particular type variable are treated as a single entity. This rule is imposed for the purpose of keeping things reasonably straightforward. For example, given:

```
data Foo a b = MkFoo a b
             | MkA a
             | MkB b

data Grok a b c = MkGrok a b c
                | GrokodileDundee a a a b b c
```

a value of `(Foo Int Int)` is mapped to `Lift (2 x 2)`, and `(Grok Int Int Int)` to `Lift (2 x 2 x 2)`. More complicated parameterisations give rise to more complicated domains. The type `(Grok Int (Foo Int Int) (Grok Int Int Int))` has a 91 point domain `Lift (2 x Lift (2 x 2) x Lift (2 x 2 x 2))`.

It is worth understanding that the number of product components is equal to the number of type variables, and entirely unrelated to the number of parameters of any particular constructor. A context `U[1,0,0]` applied to an object of type `(Grok Int Int Int)` means: evaluate the object to the first constructor. Then, if it is a `MkGrok`, evaluate the first argument. Otherwise, it must be a `GrokodileDundee`, so evaluate the first three parameters. We treat the first argument to `MkGrok` and the first three of `GrokodileDundee` as a single entity because they all correspond to the same type variable, `a`, in the declaration.

2.2.3 Recursive structured types

So far, things are reasonably straightforward. But defining evaluators for recursive types is a minefield, partly because there are so many alternative formulations [Wad87] [WH87]. As it happens, the formulation used in Anna is a trivial variation of the rule for non-recursive types, but justification is not so easy.

The rule is identical to the non-recursive case, except for the following modification: the single lifting of the product, written `Lift`, is replaced by a double lifting, `Lift2`. Now, given the pseudo-declaration

```
data [a] = []
         | a : [a]
```

it is easy to see that the domain for `[Int]` is `Lift2 (2)`, a four point domain corresponding precisely to the interpretation for that type made by Wadler [Wad87] and later justified by Burn [Bur87]. Extending the notation of the previous section, we write the points in this domain as `{_, U_, UU[0]` and `UU[1]}`, understanding them to denote the evaluators which Burn called `{E0, E1, E2` and `E3}`:

- `_`: Do not evaluate at all (`E0`).
- `U_`: Evaluate as far as the first constructor, that is, to weak head normal form (`E1`).
- `UU[0]`: Evaluate the entire structure of the list (`E2`).
- `UU[1]`: Evaluate the entire structure of the list, and all the elements (`E3`).

In general, a recursive type of `n` parameters has evaluators of the form `{_, U_` and `UU[x1 ... xn]}`. The `UU[x1 ... xn]` points denote evaluating the entire structure, and then applying evaluator `x1` to each object corresponding to the first parameter, `x2` to objects corresponding to the second parameter, and so on. We will see, in Section 4, that this conceptual partitioning of all recursive domain points into three sections is crucial to the working of the term rewriting system used to detect fixpoints. Similarly, the non-recursive points may be partitioned into two: `{_` and `U[x1 ... xn]}`.

The guiding principle, originally stated by Wadler, is to model the recursive types by letting the sub-evaluators in `UU[...]` values be representative of the least defined element of that type in the structure. Imagine we have a list of type `[(Int, Int)]`, which induces domain `Lift2 (Lift (2 x 2))`, and we know that evaluator `UU[U[0,1]]` is the strongest that can safely be applied (that is, without danger of non-termination) to the list. Now suppose we obtain another list for which `UU[U[1,0]]` is the strongest safe evaluator, and append it to the original. What is the best evaluator that can be applied to the new list? It cannot be either of the originals, since that risks non-termination. The most we can evaluate any particular element whilst remaining safe is `U[0,0]`, so the best that can be applied to the list as a whole is `UU[U[0,0]]` – the greatest lower bound of the values for the original lists. Wadler summarised this by stating that a list is characterised "by its least defined element" but we need to be more precise: a list is characterised by the greatest evaluator that can safely be applied to any element, even if a stronger evaluator could be applied to specific elements. The same principle generalises to structured types of any number of parameters, with the greatest-lower-bound characterisation occurring independently for each parameter.

This abstraction, whilst simple, assumes that programs treat all elements of the same type inside a structure in the same

way. For example, it assumes list processing functions treat all elements in the list the same way. Functions not playing along with this may induce bad, but safe, results. Consider:

```
tail (x:xs) = xs
```

If we apply a `UU[1]` evaluator to `(tail zs)`, what can we evaluate `zs` with? Unfortunately, not `UU[1]`, since the element that `tail` throws away might just have been the one-and-only non-terminating `Int` in the list. Erring on the side of safety thus restricts the evaluator for `zs` to `UU[0]`, and loses all the potential parallelism in evaluating the elements in the rest of the list. One upshot of this, also noted by Wadler, is that defining functions directly by pattern-matching is essential to get good results. In the example below, the analyser gives a much better result for `sum1` than `sum2`, despite them having identical strictness properties.

```
sum1 []      = 0
sum1 (x:xs)  = x + sum1 xs

sum2 xs      = if    null xs
               then  0
               else  head xs + sum2 (tail xs)
```

A related defect is the inability of these domains to capture the notion of head strictness. A head strict function is one which evaluates the first item in a list whenever it evaluates the list as far as the first constructor, and discovers it to be non-nil. Head strictness is useful in a sequential implementation, so an extension of the domains to capture these properties would increase the useful scope of this analyser.

2.2.4 Modifying the notation

The above mapping assigns domain `Lift ()` to all enumeration types, for example the familiar type `data Bool = False | True`. Observe that `Lift ()` is isomorphic to the two-point domain used for base types, and rightly so. After all, we could, conceptually, define

```
data Int = ... -3 | -2 | -1 | 0 | 1 | 2 | 3 ...
```

and we'd certainly expect a two-point domain for it!

This isomorphism can be used to simplify the domain structure, and thus the strictness analysis itself. We forget about domain `2`, and instead map base types `Int` and `Char` to `Lift ()`, with points `{_, U[]}` replacing `{0, 1}`. Henceforth the new notation is used.

2.2.5 Summary

Let `D(t)` denote the domain for some type `t`. Let `typeName` be the name of some structured type, and `te1 ... ten` be some arbitrary type expressions. `D` is defined as:

```
D(Int)   = Lift ()
D(Char)  = Lift ()

D(typeName te1 ... ten)
   = Lift  (D(te1) x ... x D(ten))
     if typeName denotes a non-recursive type
   = Lift2 (D(te1) x ... x D(ten))
     if typeName denotes a recursive type
```

The concept of a function-valued context seems rather meaningless, and is left undefined until section 3.2.

2.2.6 Restrictions on structured types

The astute reader may have noticed the examples above have been rather restricted. In particular, none of them had constant types as an argument to any constructor. But this situation is commonplace, for example as the `Int` conveying balancing information in the following tree declaration:

```
data AVLTree a b
   = ALeaf
   | ANode Int (AVLTree a b) a b (AVLTree a b)
```

Instead of extending the domain-generating rules to cover such cases, it is simpler to factorise out the offending `Int`, generating a second type `AVLTree2`:

```
data AVLTree2 i a b
   = ALeaf2
   | ANode2 i (AVLTree2 i a b) a b (AVLTree2 i a b)
```

`(AVLTree2 Int a b)` is an isomorphic type to `(AVLTree a b)`, at the same type being a type for which we know how to generate an abstract domain. Conveniently, Milner-Hindley typecheckers are amenable to such substitutions.

Finally, observe that there are certain types for which it is hard to devise a sensible set of evaluators, for example:

```
data Foo a b
   = MkFoo a b (Foo (Foo a b) (Foo a b))
```

The solution adopted in Anna is simply to deem these illegal. We require that, in a type declaration, arguments on constructors are either simple variables (here, `a` or `b`), or a simple recursive instance of the structure (here, `Foo a b`). These restrictions seem inconsequential for real programs, and have been reported quite independantly in [KHL91].

2.3 The Core datatype

The extensively mangled source program eventually passed to the abstract interpreter is a type-annotated tree, representing a simple functional language. Each node in the tree carries with it an annotation. The Haskell data type used is parameterised over both the type of the annotations, and the type of the identifier names:

```
type AnnExpr a b
   = (b, AnnExpr' a b)

data AnnExpr' a b
   = AVar     a
   | AConstr  a
   | ALit     Int
   | AAp      (AnnExpr a b) (AnnExpr a b)
   | ALet     Bool [AnnBind a b] (AnnExpr a b)
```

```
    | ACase    (AnnExpr a b) [AnnAlt a b]
    | ALam     [a] (AnnExpr a b)

type AnnBind a b
   = (a, AnnExpr a b)

type AnnAlt a b
   = (a, ([a], AnnExpr a b))
```

The first and second parameters on an `AnnExpr` type are for the identifier and annotation types respectively. For example, if the type of identifiers is `Id`, and that of type expressions `TExpr`, the type of the corresponding Core expression is `(AnnExpr Id TExpr)`.

Most cases are straightforward. The `AVar` term represents an identifer, whilst `AConstr` represents a constructor name. Literal values are represented by `ALit`, and applications by `AAp`. Lambda terms are represented by `ALam`, which can bind an arbitrary number of formal parameters.

That leaves the two tricky ones. A let/rec expression is represented by `ALet`, which has a boolean flag indicating whether this is a recursive binding, a list of bindings, and a main expression in which those bindings can be used. Each binding is an identifier paired with the value it is bound to. Case expressions, denoted by `ACase`, contain a switch expression, and a list of alternatives. Each alternative is a triple of constructor name, constructor arguments and the appropriate right-hand side. The use of nested pairs is more convenient for coding purposes.

All phases downstream of the lambda-lifter exploit certain assumptions about the form of the program. The most important are that there are no nested `ALets`, that the program is in dependancy order, and that no identifier is undefined or multiply defined in the same scope.

2.4 Compiling parallel code

Evaluation transformers are supposedly exploited by compiling multiple copies of each function, up to one copy for each context in which the result might be demanded. For each copy, compile-time analysis indicates how much demand propagates to the parameters of the function, and thus how much the arguments to the call may be evaluated before the call. In this manner, demand propagation is preserved as far as this style of static analysis makes possible. Of course, all this is done in pursuit of the overall goal: maximising available parallelism. An equally important issue, not discussed here, is how to avoid excessive fine-grain parallelism.

All well and good, but the potential for code explosion renders a naive implementation impractical. Consider a function delivering a result of the contrived type `(Grok Int (Foo Int Int) (Grok Int Int Int))` discussed in section 2.2.2. Since the domain has 91 points, it would appear necessary to compile 90 versions of the code, omitting the version for no demand at all on the output. Burn's early work simply ignored the problem by restricting itself to lists of `Int`, for which at most 3 copies of code are required. Quite what to do about complex types, which induce product domains, or non-trivial instantiations of lists, is not clear.

This unsatisfactory state of affairs can to some extent be alleviated by restricting ourselves to compiling just a subset of all the possible versions of each function. Then, when the output of a function is demanded in a context for which no version has been compiled, the version used is that compiled for the greatest demand less than the demand we required. Observe that the choice of alternative is not necessarily unique, but, provided we compiled in a fully sequential (that is, WHNF demand) version, an alternative is at least guaranteed to exist. Of course, some potential parallelism may well be lost: such is the price for restricting the code explosion to a tolerable magnitude.

The central question, then, is which versions to compile code for. One person who has ventured into this quagmire is Mintchev. For his MSc dissertation [Min92], Mintchev built a simulation of a parallel graph reduction machine, which understands three levels of demand: none at all, weak head normal form demand, and full demand. An immediate advantage is that these points apply to all structured types, including tuples and complex instantiations of types, and are thus more widely applicable than Burn's scheme. Encouragingly, even with so few evaluators, Mintchev found that substantial amounts of parallel activity were generated, validating his approach. Recently it has been suggested that a fourth evaluator might be profitably included: evaluation of the entire structure of a recursive type, but no evaluation of the components. This makes no sense, of course, in a non-recursive type, or, alternatively, one can regard it as equivalent to the WHNF evaluator, in this case.

A further complication is what to do about polymorphic functions. We may compile three versions of the `reverse` function, working from the evaluators of the simplest instance, but then what do we do given an evaluator `UU[U[1,0]]` applied to a use of `reverse` at non-base instance `[(Int, Int)] -> [(Int, Int)]`? Suffice it to say that a possible solution is only to compile versions of polymorphic functions based on their evaluators for simplest instances, and use safe approximation techniques based on Conc maps to handle the non-base instances. See [HH91], section 5, for an introduction to Conc maps. My MSc dissertation [Sew91] indicates how Conc maps are useful in matters of polymorphism, a theme explored further in [Sew93].

Now, if the compiler is only going to make code for a few of all the possible evaluators for a function returning an object of complex type, what is the point of doing strictness analysis with the full complement of evaluators? After all, this amounts to doing a detailed analysis, then throwing away most of the detail in the final answer. It would certainly be much quicker just to work with those few evaluators we are really interested in. Nevertheless, doing that risks losing intermediate detail, and, ultimately, parallelism, compared with the expensive approach. Building an abstract interpretation for the el cheapo approach might also be rather difficult, and the interpreter would have to be rewritten every time the particular subset of interesting evaluators changed.

A final interesting caveat pertains to higher-order functions. As explained towards the end of section 1.2, it looks difficult to exploit parallelism in higher-order functions if we do not want to engage in complicated manipulation of evaluation transformers at run-time. One can therefore reasonably argue that the higher-order removal transformation (firstification) described in section 5 enhances parallelism. What firstification does is to discover statically some of the functional parameters passed to higher-order functions, and specialise them accordingly, generating first-order replace-

ments. These can then be parallelised in the normal way, without having to resort to complicated run-time machinery. Maybe, then, this transformation should be incorporated as a matter of course into good parallelising compilers.

3 The abstract interpretation

3.1 Preliminaries

3.1.1 The notion of forwards and backwards

Semantic analyses of functional languages seem to fall into two camps: forwards and backwards. To see the intuitive meaning of this, consider a function application:

```
(f x y z)
```

A forward analysis generates information about `f` which tells us properties of the application `(f x y z)` if we know the properties of the individual arguments, `x`, `y` and `z`. In other words, the analysis propagates information *forwards* through functions. Forward analyses tend to be expensive because they have to consider all possible interactions between arguments. On the other hand, one gets a very detailed picture of what is going on.

A backward analysis, by contrast, generates information about `f` which tells us the properties of the individual arguments `x`, `y` and `z` if we know some property of the application `(f x y z)`. That is, a backwards analysis propagates properties *backwards* through functions. Backward analyses may be cheaper to do, but they may also give less detailed results.

Now, for reasons which will shortly become apparent, Anna does a combined forward and backward analysis. The properties which Anna propagates forwards through functions are the abstract values, whilst those which flow backwards are contexts. To set the stage, observe critically that *Anna's main purpose is to determine the backwards behaviour of the source language functions.* The presence of forward (abstract) values is a necessary evil which enables us to deal cleanly with higher-order functions. The discussion which now follows makes more sense if you keep a clear notion that abstract values correspond to a forwards flow of information, whilst contexts correspond to a backwards flow.

3.1.2 A fundamental problem with backwards analysis

Backwards strictness analysis would be straightforward, were it not for the higher-order nature of the language under analysis. To see the problem, consider `apply`:

```
apply f x = f x
```

Given some demand on a use of apply, `(apply g y)`, what demand may be propagated to `y`? Without knowing how `g` propagates demand to its argument, the only safe answer is "none". However, knowing what `g` is implies having a forward flow of information, as well as the backward flow of demand we started with.

Things look grimmer when we put functions inside data structures, then fish them out and apply them:

```
1 + (head xs (y+1))
```

where `xs :: [Int -> Int]`. There is no way to tell what demand could be propagated to `y`.

The solution really lies in building a combined backwards and forwards analysis. Wray [Wra85] made a start on the problem, but it took the work of Hughes [Hug87] to generalise Wray's results to the point of general applicability. The resulting analysis is rather hard to understand, so, rather than attempting a head-on assault, we look first at underlying issues, starting off with some new concepts.

3.2 Function contexts

Dealing with functions properly means turning them into first-class citizens for the purposes of the abstract interpreter. Section 2.2 discussed the notion of demand (or context) on a data structure. We now extend the notion of context to functions.

Since a context really denotes a demand for evaluation, the idea of a function context seems pretty meaningless: after all, how can a function be evaluated? But imagine we defined a function context as a pair, containing the abstract value of the argument, and the context for the result of an application of the function. By making the analysis fully curried from now on, we can consider all functions as having just one argument. Then, for example, the context on a function of two arguments, such as `apply`, looks like:

```
(abstract value of first argument,
   (abstract value of second argument,
    context on result)
)
```

Such a scheme would solve the problem outlined above, by supplying the value of the first parameter, allowing demand to be propagated onto the second parameter. This abstract value of the first parameter is just the relevant context function, but what of the abstract value of the second, non-functional parameter? Well, we are simply not interested in it, so we map it to the 1-point domain.

The world now becomes populated by two species of values:

- **Contexts** (also called backwards values, evaluators or demands). In what follows, we often say "context" or "context on" when it would be more natural to say "demand" or "demand on". This convention has been adopted because of a wish to have just one term for each concept.

  Contexts are the first kind of abstract entity referred to in section 2.1. For non-function values, they are as discussed in section 2.2. For function values, they are a pair which we write as `(Fnc a c)`, where `Fnc` reminds that this is a **FuNction Context**, `a` is the abstract value of the argument and `c` is the context on the result. Henceforth, variables denoting contexts or context maps have 'c' as their first letter.

- **Abstract values** (also called forwards values). These are the second kind of abstract entity described in section 2.1. They are designed purely to convey contexts to any place involving a call to an unknown function, such as in the two problematic examples above. The abstract values of non-function objects are always irrelevant and are mapped to a 1-point domain, whose single point is denoted `#`, for the time being. The abstract

Defining equation	Disassembles a ...	producing the ...
`FncA (Fnc a c) = a`	functional context	abstract value of the argument
`FncC (Fnc a c) = c`	functional context	context on the result
`FvalA (Fval c a) = a`	functional abstract value	abstract value map: argument to result
`FvalC (Fval c a) = c`	functional abstract value	context map: result to argument

Table 1: Selector functions for functional entities

value of a function-valued object is also a pair (but quite unrelated to `Fnc` pairs), written `(Fval c a)`, with `Fval` reminding us this is a **Functional abstract VALue**. The two components are both maps. The first component, `c`, maps the context on the function to context on the argument, whilst `a` maps the abstract value of the argument to the abstract value of the result. Variables denoting abstract values or abstract value maps have 'a' as their first letter.

Notice how the two kinds of values are mutually recursive. The overall output of the abstract interpreter is one abstract value per Core function. Each abstract value contains enough information to propagate demand from the overall result to each of the arguments, even in the presence of functional parameters. These concepts are confusing, so some examples are in order. First, define four selectors `FncA`, `FncC`, `FvalA` and `FvalC` to disassemble `Fnc`s and `Fval`s, with the behaviour shown in Table 1. Hopefully, their names will serve as a reminder of their meaning.

Let's start with the simplest function imaginable: `id :: Int -> Int`. The only remotely interesting thing we can say about `id` is that it simply propagates the context on its result to the context on its argument. So, supposing we now write down a mapping from the context on the result to the context on the argument, we get:

```
(\c -> c)
```

Let's be clear what this is. It's *not* a context, and it's also *not* an abstract value. It's a map from contexts to contexts.

But that's not good enough. We said earlier that Anna produces one abstract value per Core function. So what do we produce for `id`? For a start, since `id` is a function, we must get a functional abstract value: an `Fval` term. It must look like:

```
id = Fval context_map
          abstract_value_map
```

Now, the context map, as we just mentioned, maps the context on `id` to the context on `id`'s argument. And the context on `id`, since `id` is a function, must be a function context, of the form `(Fnc a c)`, where `c` is the bit we're really after. This gives a context map of `(\c -> FncC c)`, so we've now got:

```
id = Fval (\c -> FncC c)
          abstract_value_map
```

What of the abstract value map? It tells us what the abstract value of `id`'s result is given the abstract value of `id`'s argument. But, for this instance of `id`, the result type is `Int`. All non-function types have a corresponding abstract value, denoted `#`, in a 1-point domain. So we don't actually care what the abstract value of `id` is – it can only be `#` anyway. That means, after installing the abstract value map, we could write either of the following, although the second is a little clearer:

```
id = Fval (\c -> FncC c)
          (\a -> a)

id = Fval (\c -> FncC c)
          (\a -> #)
```

If you are confused, go no further! It is better to return to the start of this section, consider again the meanings of contexts and abstract values, and iterate until the example makes sense.

Moving on to `(+) :: Int -> Int -> Int` gives:

```
(+) = Fval (\c1 -> FncC (FncC c1))
           (\a1 -> Fval (\c2 -> FncC c2)
                        (\a2 -> #))
```

This time currying comes into play. That's why the term which maps the abstract value of the first argument to the abstract value of the result returns a `Fval` term: the "result" here has type `Int -> Int`. Clearly, `(+)` simply propagates context on the overall result to both arguments, which is why the context maps for the two arguments are `(\c1 -> FncC (FncC c1))` and `(\c2 -> FncC c2)`. If this seems a little mysterious, bear in mind that both `(FncC (FncC c1))` and `(FncC c2)` refer to the context on the final result. That's because `c1` binds to a context in `Int -> Int -> Int`, which is necessarily of the form `(Fnc # (Fnc # cc))` where `cc` is the context on the final result. Similarly, `c2` is a context of type `Int -> Int`, having the form `(Fnc # cc)` where `cc` is again the context on the final result.

Now for something altogether more adventurous: the familiar `apply` function, at type `(Int -> Int) -> Int -> Int`. This example is easier to follow if one bears in mind that `(apply f x)` reduces immediately to `(f x)`, so any context applied to the former expression also applies directly to the latter. What the rather formidable term below does is to route the context from the result of calling `apply` to the result of calling the higher-order parameter.

```
apply = Fval (\c1 -> Fnc (FncA (FncC c1))
                         (FncC (FncC c1)))
             (\a1 -> Fval (\c2 -> (FvalC a1)
                                  (Fnc (FncA c2)
                                       (FncC c2))
                                  (\a2 -> (FvalA a1) a2))
```

First of all, consider what the function context `c1` will get bound to must look like: `(Fnc a_ho (Fnc # c_final))` where `a_ho` us the abstract (or forward) value of the functional parameter and `c_final` is the context on the result of applying this functional parameter to something. Now, term `(\c1 -> Fnc (FncA (FncC c1)) (FncC (FncC c1)))` maps context on `apply` to context on the first parameter. As this is a functional parameter, it makes sense that this expression is built from a `Fnc`. So just what context is propagated to the functional parameter? Well, the abstract value must be the same as the abstract value of the second parameter to `apply`, and this value (which must be `#`) is extracted by the term `(FncA (FncC c1))` Similarly, the context on the result of the functional parameter must be the same as the context on the overall result of `apply`, given by `(FncC (FncC c1))`.

Everything else is easier to follow. Variable `a1` will get bound to the abstract value of the functional parameter, which must be a `Fval` term. So `(FvalC a1)` returns the map used by the functional parameter to translate context on itself to context on its first argument. The map is applied to the same function context as was built in the preceding paragraph, except that references to `(FncC c1)` are replaced by `c2`, which is the same thing.

Finally, the abstract value of the result is given by applying the abstract value map of the functional parameter, `(FvalA a1)`, to the abstract value of the second parameter, `a2`.

Two improvements are possible. Firstly, the abstract value of the result must simply be `#`, since the result type is `Int`. Secondly, examination of the definition of `FncA` and `FncC` shows that `(Fnc (FncA c) (FncC c))` is equivalent simply to `c`. The improved version is:

```
apply = Fval (\c1 -> FncC c1)
             (\a1 -> Fval (\c2 -> (FvalC a1) c2)
                          (\a2 -> #))
```

The mechanism for dealing with functions and applications is the hardest part of the abstract interpreter to understand. A little time spent making sense of this last example is a wise investment.

Why is it necessary to propagate demand into functional parameters? Well, consider:

```
add1 x = apply (+ x) 1
```

If demand isn't propagated into `apply`'s functional parameter, there will be no demand on term `(+ x)` and none on `x`, giving the impression that `add1` is not strict, when really it is.

3.3 More about abstract values

All non-function expressions yield an abstract value in a unit domain. However, value `#`, used in the examples above, is too indiscriminating. The Haskell declaration for abstract values looks (almost) like this:

```
data AbsVal
   = ANonRec  [AbsVal]
   | ARec     [AbsVal]
   | Fval     Context AbsVal

   | AbsVar   Id
   | AbsLam   Id AbsVal
   | AbsAp    AbsVal AbsVal

   | FncA     Context
   | FvalA    AbsVal
   | SelA     Int AbsVal
   | AMeet    [AbsVal]
```

The `ARec` and `ANonRec` terms define abstract values for recursive and non-recursive types, respectively. In both cases, the associated list of `AbsVal`s are the abstract values of the parameters of the type. For example, a term of type `[(Int, Int)]` has abstract value `(Rec [NonRec [NonRec [], NonRec []]])`, given that `Int` is treated as an enumeration and thus maps to `(NonRec [])`. It is important to realise that this value is still unitary, like `#`, but has the added advantage that it can be disassembled to reveal its unitary subcomponents, as necessitated by the abstract interpretation of `case` statements. Constructor `SelA` is used for this, with meaning:

```
SelA n (ARec    [a1 ... an ... ak])  = an
SelA n (ANonRec [a1 ... an ... ak])  = an
```

`Fval`, `FncA` and `FvalA` were introduced in the previous section. `AbsVar`, `AbsLam` and `AbsAp` allow references to abstract-valued variables, and for the creation and application of abstract-valued mappings. Observe that we often omit `AbsVar` and `AbsAp`, when the meaning is obvious, and abbreviate `(AbsLam a e)` to `(\a -> e)`.

Consider again

```
1 + (head xs (y+1))
```

where `xs :: [Int -> Int]`. We expect `xs` to have been bound to an abstract value which can supply a sensible context-mapping function. Once again, we characterise the list by the least element, this time the least context function, in it. So, supposing

```
xs = [id, id] where id x = x
```

the abstract value of `xs` will be:

```
ARec [  Fval (\c1 -> FncC c1)
             (\a1 -> ANonRec [])  ]
```

The effect of the `head` function is to wrap `SelA 1` around this term, making the abstract value of `id` available where it is needed. But, now, if `xs` were defined as

```
xs = [id, const] where id x     = x
                       const x  = 42
```

we need to be more cautious. Since the abstract interpretation cannot distinguish items in lists, we must arrange that the function which emerges from the list represents the weaker evaluator: `const`. That requires the list as a whole to have value:

```
ARec [  Fval (\c1 -> _)
             (\a1 -> ANonRec [])  ]
```

The upshot of all this is that the abstract value of a list containing functions is characterised by the least function in the list, with the principle extending analogously to all other structures. In order to carry that out, a greatest-lower-bound operation is needed for abstract values. This is what the `AMeet` term is for.

3.4 More about contexts

This is a good point at which to wheel in the Haskell declaration for contexts. Unfortunately, it is even more cumbersome than the `AbsVal` declaration. Nevertheless:

```
data Context
   = Stop1
   | Up1       [Context]
   | Stop2
   | Up2
   | UpUp2     [Context]
   | Fnc       AbsVal Context

   | FncC      Context
   | FvalC     AbsVal

   | CJoin     [Context]
   | CMeet     [Context]

   | CtxVar    Id
   | CtxLam    Id Context
   | CtxAp     Context Context

   | SelU      Int Context
   | SelUU     Int Context
   | CaseU     Context Context Context
   | CaseUU    Context Context Context Context

   | DefU      Context
   | DefUU     Context
```

The first six are for building literal contexts. `Stop1` and `Up1` pertain to points in `Lift (D1 x ... x Dn)`, with `Stop1` representing the bottom point `_`, and `(Up1 [x1 ... xn])` representing the point `U[x1 ... xn]`. Similarly, `Stop2`, `Up2` and `(UpUp2 [x1 ... xn])` represent the points `_`, `U_` and `UU[x1 ... xn]` in the domain `Lift2 (D1 x ... x Dn)`. `Fnc` is used for building function-valued contexts, as discussed in section 3.2. Finally, `DefU` and `DefUU` exist to help the term rewriting system, as described in section 4.4.

`FncC` and `FvalC` were also discussed in section 3.2. `CJoin` and `CMeet` unsurprisingly denote the least upper and greatest lower bounds of their respective argument lists.

`CtxVar`, `CtxLam` and `CtxAp` are exact equivalents to the `AbsVar`, `AbsLam` and `AbsAp` discussed in section 3.3. They provide a way to reference context-valued variables, and allow the creation and application of context-valued maps. Once again, note that we often omit `CtxVar` and `CtxAp`, when the meaning is obvious, and abbreviate `(CtxLam c e)` to `(\c -> e)`.

Far and away the most interesting constructs are the last four. `CaseU` and `CaseUU` allow partial disassembly of values in `Lift (D1 x ... x Dn)` and `Lift2 (D1 x ... x Dn)` respectively, in the manner discussed in section 2.2.3. The exact semantics are:

```
CaseU Stop1   x y  = x
CaseU (Up1 _) x y  = y

CaseUU Stop2     x y z  = x
CaseUU Up2       x y z  = y
CaseUU (UpUp2 _) x y z  = z
```

Note that the switch values are restricted to being in domains `Lift (D1 x ... x Dn)` and `Lift2 (D1 x ... x Dn)` respectively. Switch values from any other domain constitute an ill-formed context. `CaseU` and `CaseUU` terms denote a mapping from their switch expressions to one of the alternatives. As such, a well-formed `CaseU` or `CaseUU` must denote a monotonic mapping, so we impose the semantic constraint that $\mathtt{x} \sqsubseteq \mathtt{y} \sqsubseteq \mathtt{z}$.

As you might suspect, `SelU` and `SelUU` are selectors in the spirit of `SelA`, discussed in section 3.3. Semantics are:

```
SelU  n (Up1   [x1 ... xn ... xk]) = xn
SelUU n (UpUp2 [x1 ... xn ... xk]) = xn
```

But there is a very strong semantic constraint here: it is illegal to apply `SelU` or `SelUU` to a value unless that value is provably equivalent to an `Up1 [...]` or `UpUp2 [...]` value respectively. This means, for some arbitrary context `c`, the following expressions are likely to be ill-formed:

```
SelU  n c
SelUU n c
```

The one and only way to make them well-formed is to wrap the appropriate species of `Case` term around them, leaving the `Sel` in the greatest-value arm:

```
CaseU  c (...whatever...) (SelU n c)

CaseUU c (...whatever...) (...whatever...)
                          (SelUU n c)
```

In both cases, the term `(Sel n c)` may not appear in any place marked "`...whatever...`". Note that the `Sel` term may appear anywhere within the greatest-value arm, and is not restricted to the top level, as this example seems to suggest.

3.5 Constructor functions and case statements

The source-language trappings of structured types give rise to some of the more interesting parts of the abstract interpreter, and warrant a section to themselves. First, though, some terminology. A structured type is defined like this:

```
data typeName v1 ... vk = C1 t11 ... t1m
                        |     ...
                        | Cn t1n ... tnm
```

This defines a type called `typeName`, paramterised by type variables `v1` to `vk`, with constructors `C1` to `Cn`. The type expressions `t11` to `tnm`, which form the arguments to the constructors, are heavily constrained in the manner discussed in section 2.2.6: they may only be either one of the type variables, `v1 ... vk`, or a direct recursive call to the type: `(typeName v1 ... vk)`.

Because of this constraint, each constructor argument in a valid definition can be classified either as a recursive call `Rec`, or as one of the type variables, `Var n` where `n` is a number denoting which variable. For example, the definition

```
data AVLTree i a b
   = ALeaf
   | ANode i (AVLTree i a b) a b (AVLTree i a b)
```

can, in principle, be rewritten as

```
data AVLTree (of 3 type variables)
   = ALeaf
   | ANode (Var 1) Rec (Var 2) (Var 3) Rec
```

We now define two strange functions, `argkind` and `update`, to assist in the discussion below. Neither are meant to be implementable. Rather, they serve as convenient notational devices, and are best illustrated by example. They are both meaningless unless the particular constructor application they are associated with is stated.

`argkind` tells us what part of a data type a given constructor argument corresponds to: either a certain type variable, or a recursive instance of the type. For example, bearing in mind the declaration above, given the constructor application `(ANode i l a b r)`:

```
argkind i = Var 1
argkind l = Rec
argkind a = Var 2
argkind b = Var 3
argkind r = Rec
```

`update` replaces a particular value in a supplied list with another value. It finds out which location to update by using `argkind`, expecting an answer of the form `(Var i)`, whereupon `i` is used as the location. It is invalid to use `update` in a way which would cause the call to argkind to return `Rec`. Again, using the constructor application `(ANode i l a b r)`:

```
update i "my"   ["the", "cat", "sat"]
   = ["my",  "cat", "sat"]

update a "dog"  ["the", "cat", "sat"]
   = ["the", "dog", "sat"]

update b "ran"  ["the", "cat", "sat"]
  = ["the", "cat", "ran"]
```

But

```
update l x xs
update r x xs
```

are both illegal since `argkind l` = `argkind r` = `Rec`.

The example used `update` to replace words in a list thereof to emphasise `update`'s polymorphic nature. Note that `update` is always used with a constructor wrapped round the final list argument. This constructor is re-attached to the result:

```
update i "my"
     (SomeConstructor ["the", "cat", "sat"])
   = (SomeConstructor ["my",  "cat", "sat"])
```

For the sake of clarity, this inconsequential detail is henceforth ignored.

Recall from section 2.2.5 that function `D` returns the domain associated with a particular type. Four more handy functions of similar ilk are `top`, `bot`, `topfv` and `whnf`. The first two simply generate the greatest and least contexts in a particular domain. `topfv(D)` generates the greatest abstract value in domain `D`.

```
top (Lift  (D1 x ... x Dn))
   = Up1   [top(D1) ... top(Dn)]
top (Lift2 (D1 x ... x Dn))
   = UpUp2 [top(D1) ... top(Dn)]
top (Ds -> Dt)
   = (\c -> top(Ds))

bot (Lift  (D1 x ... x Dn))
   = Stop1
bot (Lift2 (D1 x ... x Dn))
   = Stop2
bot (Ds -> Dt)
   = (\c -> bot(Ds))

topfv (Lift  (D1 x ... x Dn))
   = ANonRec [topfv(D1) ... topfv(Dn)]
topfv (Lift2 (D1 x ... x Dn))
   = ARec    [topfv(D1) ... topfv(Dn)]
topfv (Ds -> Dt)
   = Fval (\c -> top(Ds))
          (\a -> topfv(Dt))
```

`whnf(D)` is the weak head normal form evaluator for domain `D`. This only makes sense for certain values of `D`:

```
whnf (Lift  (D1 x ... x Dn))
   =   Up1 [bot(D1) ... bot(Dn)]
whnf (Lift2 (D1 x ... x Dn))
   = UpUp2 [bot(D1) ... bot(Dn)]
```

Finally, for the record, a Core `case` expression looks like

```
case switchExpression of
   C1 p11 ... p1m -> rhs1
         ...
   Cn p1n ... pnm -> rhsn
```

where it is assumed that all constructors are present. This is assured by the pattern-matching phase of the desugarer.

The four following sections document the flow of abstract values and contexts through constructor applications and `case` expressions. In some ways, the two are opposites: constructor applications build structures, whilst `case` expressions disassemble them. An interesting duality arises from this. The flow of abstract values though case expressions is uncannily similar to the flow of contexts values through constructors, and vice versa.

3.5.1 Constructor functions: abstract value propagation

How do abstract values flow through a constructor? The discussion of section 3.3 implied that the the `(:)` function must behave something like:

```
(:) = \x xs -> AMeet [xs, ARec [x]]
```

Observe that the apparently polymorphic nature of this definition is incidental. In general, given an arity-n constructor `C` and arguments `a1 ... an` where `(C a1 ... an) :: tau`, the forward behaviour of C is:

```
\a1 ... an -> AMeet [e1 ... en]

ei = ai
     if argkind ai = Rec

   = ARec (update ai ai topfv(D(tau)))
     if   argkind ai == Var x
     and  C is from a recursive type

   = ANonRec (update ai ai topfv(D(tau)))
     if   argkind ai == Var x
     and  C is from a non-recursive type
```

Nullary constructors simply acquire the top abstract value of the relevant domain (bear in mind that, for a domain not containing function spaces, this is the same as the bottom point). The `[]` case for `[Int]`, for example, is:

```
[] = topfv(D( [Int] ))
   = topfv( Lift2 (Lift ()) )
   = ARec [ANonRec []]
```

The motive in all this is to ensure that the abstract value of a constructor application is characterised, for each parameterising type, by the least value of that type.

As an example, consider an object of type `(AVLTree Int Int Int)`. Contexts for that type are drawn from the domain `Lift2 (Lift () x Lift () x Lift ())`. We expect the abstract value returned by both the `ALeaf` and `ANode` constructors to be of the form `ARec [ii, aa, bb]` where `ii` represents the least abstract value of any object corresponding to type variable `i` in the type definition, and similarly for `aa` and `bb`. So, at this instantiation, the abstract value behaviour of the constructors is:

```
ALeaf = ARec [ANonRec [], ANonRec [], ANonRec []]

ANode
= \i l a b r ->
  AMeet
  [ ARec [i,          ANonRec [], ANonRec []],
    l,
    ARec [ANonRec [], a,          ANonRec []],
    ARec [ANonRec [], ANonRec [], b         ],
    r
  ]
```

Non-recursive types are dealt with in an exactly analogous manner. For example, the behaviour of the pairing constructor at type `Int -> Int -> (Int, Int)` is

```
(,)
= \x y ->
  AMeet
  [ ANonRec [x,          ANonRec []],
    ANonRec [ANonRec [], y         ]
  ]
```

If your instincts tell you this is much ado about nothing, you are correct. Since all these examples build structures without embedded function spaces, the result values are unitary, and may be written:

```
[]  =          ARec [ANonRec []]
(:) = \x xs -> ARec [ANonRec []]

(,) = \x y -> ANonRec [ANonRec [], ANonRec []]

ALeaf = ARec [ANonRec [], ANonRec [], ANonRec []]
ANode = \i l a b r ->
        ARec [ANonRec [], ANonRec [], ANonRec []]
```

3.5.2 Constructor functions: context propagation

The name of the game here is to say what context propagates from a non-nullary constructor to its arguments. Intuiting first on `[Int]`, `(:)` exhibits the following behaviour:

Demand on (x:xs)	Demand on x	Demand on xs
UU[U[]]	U[]	UU[U[]]
UU[_]	_	UU[_]
U_	_	_
_	_	_

A `UU[U[]]` context causes evaluation of the entire structure of the list, and all the `Int`s in it too. So we may propagate `U[]` to `x` and `UU[U[]]` to the tail of the list. The same reasoning explains the propagation of a `UU[_]` context. Now, what of `U_`? This evaluator simply evaluates to WHNF, that is, the, first constructor, and gives up. So zero context may be propagated to either head or tail. Similarly, zero context propagates from zero context on `(x:xs)`.

Is there a pattern here? The context on `xs` is that same as the context on `(x:xs)` except at the WHNF point, whilst the context on `x` is "`y`" in the `UU[y]` cases, and none otherwise. This latter operation could be regarded as dropping the double-lifting, and selecting the first product component. Writing the context on `(x:xs)` as `alpha`, context on `x` and `xs` respectively could be written as:

```
DropUU 1 alpha
ZapWHNF alpha
```

Implementing `DropUU` and `ZapWHNF` directly causes major problems in the term-rewriting system. Fortunately, the `CaseUU` and `SelUU` primitives can be used instead:

```
DropUU n alpha = CaseUU alpha _ _ (SelUU n alpha)
ZapWHNF alpha  = CaseUU alpha _ _ alpha
```

Analogising the informal argument leads to a general rule. Given an arity-`n` constructor `C` and arguments `a1 ... an` where `(C a1 ... an) :: tau`, context `alpha` on the constructor application produces context on `ai` as follows:

```
ai  = ZapWHNF alpha
      if   argkind ai == Rec

    = DropUU x alpha
      if   argkind ai == Var x
      and  C is from a recursive type

    = DropU x alpha
      if   argkind ai == Var x
      and  C is from a non-recursive type
```

The `AVLTree` example at instance `(AVLTree Int Int Int)` behaves as follows for a context `alpha` applied to `(ANode i l a b r)`:

```
Variable       Demand
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
i              DropUU 1 alpha
l              ZapWHNF  alpha
a              DropUU 2 alpha
b              DropUU 3 alpha
r              ZapWHNF  alpha
```

Context propagation for non-recursive types behaves in a similar manner, except that it is no longer possible to generate `ZapWHNF`, and the drop-select operator only drops one point, instead of two. This operator, called `DropU`, is implemented as

```
DropU n alpha  = CaseU alpha _ (SelU n alpha)
```

For a context `alpha` applied to `(x, y) :: (Int, Int)`, the contexts propagated to `x` and `y` are `(DropU 1 alpha)` and `(DropU 2 alpha)` respectively.

Translation of `DropU`, `DropUU` and `ZapWHNF` into the `Case` and `Sel` primitives requires some passing around of domains, so that the appropriate kind of bottom values can be manufactured.

3.5.3 Case expressions: abstract value propagation

The task here is to figure out what abstract values to attach to constructor variables in a case expression, given the abstract value of the switch expression. The solution is remarkably similar to propagation of contexts to constructor arguments, and follows a theme which should be becoming familiar. Given a case expression

```
case sw of
   ...
   C a1 ... an -> rhs
   ...
```

and an abstract value associated with `sw` of `fsw`, the abstract value associated with `ai` is

```
ai  = fsw
      if   argkind ai == Rec

    = SelA x fsw
      if   argkind ai == Var x
```

Let the abstract value of the switch expression be denoted `fsw`. For `[Int]` we have:

```
case sw of
   []     ->  rhs1
   (x:xs) ->  rhs2
```

giving bindings of

```
x  --->  SelA 1 fsw
xs --->  fsw
```

`(AVLTree Int Int Int)` gives:

```
case sw of
   ALeaf            -> rhs1
   ANode i l a b r  -> rhs2

i  --->  SelA 1 fsw
l  --->  fsw
a  --->  SelA 2 fsw
b  --->  SelA 3 fsw
r  --->  fsw
```

Finally, `(Int, Int)` gives:

```
case sw of
   (x, y)  -> rhs1

x  --->  SelA 1 fsw
y  --->  SelA 2 fsw
```

3.5.4 Case expressions: context propagation

This section establishes how context on a `case` expression propagates to context on the switch expression. First, a subsidiary result.

Forwards propagation of contexts though constructors

Given a constructor application `(C a1 ... an) :: tau`, the method of section 3.5.2 can tell us how context on this application maps to context on `a1 ... an`. However, we now need to run the process in reverse. Given some contexts `c1 ... cn` on `a1 ... an`, we want to find the greatest context `alpha` that may be put on the application, constrained so that the contexts that section 3.5.2 indicates would then propagate to `a1 ... an` are less than or equal `c1 ... cn` respectively.

The following scheme is offered, again without justification. If `C` is from a recursive type:

```
alpha = CJoin [Up2, CMeet [e1 ... en]]

ei = ai
     if   argkind ai == Rec

   = update ai ai top(D(tau))
     if   argkind ai == Var x
```

If `C` is from a non-recursive type, `(argkind ai)` cannot be `Rec`, so this simplifies to:

```
alpha = CMeet [e1 ... en]

ei = update ai ai top(D(tau))
     if   argkind ai == Var x
```

Finally, for nullary constructors, like `[]`:

```
alpha = ctop(D(tau))
```

Examples: given `(a1, a2) :: (Int, Int)`, we get

```
alpha = CMeet [ Up1 [c1,    U[] ],
                Up1 [U[],   c2] ]
```

`(a1:a2) :: [Int]` gives

```
alpha = CJoin [ Up2,
                CMeet [ UpUp2 [c1],
                        c2 ]
              ]
```

`[] :: [Int]` gives

```
alpha = UpUp2 [Up1 []]
```

`(ANode a1 a2 a3 a4 a5) :: (ATree Int Int Int)` gives

```
alpha = CJoin
        [ Up2,
          CMeet [ UpUp2 [c1,    U[],   U[] ],
                  c2,
                  UpUp2 [U[],   c3,    U[] ],
                  UpUp2 [U[],   U[],   c4  ],
                  c5 ]
        ]
```

Using the lemma

And now to return to the main theme. At this point, it's necessary to introduce a function we will see a lot more of later. The function `C` tells us how much context is propagated to a variable `x` when context `alpha` is propagated to some arbitrary expression `e`. Of course, if `x` does not occur free in `e`, the answer is none. We write this as

```
C x [e] rho alpha
```

with the `e` in square brackets to emphasise that `C` regards it as a syntactic object. As becomes apparent later, `C` also requires an environment `rho` which supplies abstract values for all free variables in `e`.

Recall that a `case` expression looks like this:

```
case sw of
   C1 p11 ... p1m -> rhs1
   ...
   Cn p1n ... pnm -> rhsn
```

Now, given context `alpha` overall, what is the context on `sw`? The first step is to find the context on `p11 ... pnm`. These context are given by:

```
(C p11 [rhs1] rho1 alpha)   ...
(C p1m [rhs1] rho1 alpha)

...

(C p1n [rhsn] rhon alpha)   ...
(C pnm [rhsn] rhon alpha)
```

For each particular constructor, the original environment `rho` is augmented with abstract value bindings for the variables associated with that constructor, generating `rho1 ... rhon`. These values are derived from the abstract value of the switch expression, as described in section 3.5.3.

The next step is to figure out what context can be safely applied to each constructor, knowing the contexts on their individual arguments. The method described in the lemma is applied, once for each constructor, to the contexts for `p11 ... pnm` just computed, giving `alpha1 ... alphan`. These values are combined to give the overall context on `sw` as:

```
CMeet [alpha1 ... alphan]
```

Using `CMeet` to merge these values reflects the fact that we cannot know which alternative will be selected at compile time. The best safe value which can be obtained is the least of any of alternatives.

Unfortunately, there is one special case where this formulation is wrong. When the switch expression is of a recursive type, like `[Int]`, one finds that propagating zero context onto the `case` expression produces non-zero context on the switch expression. This unsafe result can be traced to the case for recursive types being of the form `alpha = CJoin [Up2, ...]`, which imposes a minimum value of `Up2` on the context contributed by each constructor. Simply throwing away the `Up2` clamping bit causes more problems than it solves. A better solution is to explicitly impose the required condition that zero overall context produces zero context on the switch expression. Recalling that `alpha` is the overall context, this is done by writing

```
CaseU alpha _ (CMeet [alpha1 ... alphan])
```

or

```
CaseUU alpha _ (CMeet [alpha1 ... alphan])
               (CMeet [alpha1 ... alphan])
```

depending on the domain of `alpha`. The possible duplication of the `(CMeet [alpha1 ... alphan])` term is regrettable, and could potentially cause major performance problems. Section 6.1 shows how these may be avoided.

The complications in this business seem endless. We have just created yet another problem. Consider:

```
let  id y = y in
     case e of
        []      -> id
        (x:xs)  -> id
```

This `case` expression returns a function, which is perfectly legitimate. But the overall context on it, `alpha`, will be a function context, and it is quite meaningless to scrutinise such a value with `CaseU` or `CaseUU`. A little thought reveals a simple solution. The `case` expression returns a function, which will, eventually, be applied to something. What really matters is the context on the final result of that application: if non-zero, it means the `case` expression will eventually have to be entered, in order to generate a function which in turn generates some result to satisfy the demand. So, all we need do, if `alpha` is a function context, is test the final context

encapsulated in alpha, rather than alpha itself. Getting the final context out of an n-arity function context is easily done by wrapping n FncC selectors round it. So context on the switch expression, in terms of alpha, now looks like:

```
CaseUU (FncC (FncC ..... (FncC alpha) .....))

       _
       (CMeet [alpha1 ... alphan])
       (CMeet [alpha1 ... alphan])
```

The number of FncCs is equal to the arity of alpha, if alpha happens to be a function context. case expressions returning functions seem to be rarities, so usually there will be zero FncCs. The corresponding modification of the CaseU version is obvious, and it only remains to say that choosing between the two now depends on the final context encapsulated in alpha when alpha is a function context.

As our long journey through the forest of supporting machinery comes to a close, so the final destination draws into sight: the definition of the abstract interpreter proper. We pause but briefly to take respite in the following example, then embark upon the final straight: section 3.6.

```
case vs of
   []      -> 0
   (x:xs)  -> x
```

Clearly, vs :: [Int] and the overall type is Int. So a context alpha placed on the result must be in domain Lift (), with the resulting context on vs in Lift2 (Lift ()). Section 3.5.4 indicates that the [] case contributes context UpUp2 [Up1 []]. Now, propagating alpha to the (:) alternative puts context alpha on x and Stop2 (that is, none) on xs. Combining these two, again using section 3.5.4, shows that the context propagated by this alternative is:

```
  CJoin [Up2, CMeet [alpha, Stop2]]
= CJoin [Up2, Stop2]
= Up2
```

This gives overall context on vs as:

```
  CaseU alpha Stop2
              (CMeet [UpUp2 [Up1 []], Up2])

= CaseU alpha Stop2 Up2
```

That's intuitively correct: with no demand on the resulting Int, there's no (Stop2) demand on the incoming list. Otherwise, we may evaluate the list to WHNF (Up2), that is, to the first constructor. It is a pity these domains can't tell us about the head-strictness here: given non-zero demand, it's obvious we can not only evaluate to the first constructor, but can also evaluate the first element of the list if it is non-empty.

3.6 Defining the abstract interpreter

Section 3.5.4 introduced the context-finding function C. We now augment this with Z, the abstract interpreter itself. C takes any Core expression, a context on that expression, and a variable, and returns the resulting context on the variable. Z takes any Core expression, and returns the abstract value of that expression. Since the forward and backward flows of information are heavily intertwined, C and Z are mutually recursive. In a call to C orZ

```
C x [e] rho alpha
Z   [e] rho
```

x is a variable, e is a Core expression, alpha is a context, and rho is an environment binding all free variables in e to abstract values. As implemented, both functions carry an extra parameter used to help generate new variable names. C also carries the domain of x so it can generate the appropriate bottom value when needed. Recall also that a Core expression is a pair, the first part of which is the type of the expression, and the second the expression proper.

3.6.1 Definition of Z

The abstract value of a literal is a value in the appropriate one-point domain.

```
Z (tau, ALit n) rho  = ANonRec []
```

Variables have their values looked up.

```
Z (tau, AVar v) rho  = rho v
```

Applications are a little more tricky. First, the abstract value of the function is created. From that, the abstract-value-map is extracted using FvalA,, and applied to the abstract value of the argument to give the abstract value of the result.

```
Z (tau, AAp f e) rho
   = AbsAp (FvalA (Z f rho)) (Z e rho)
```

Lambda terms are a lot more tricky. Let a and c denote new variables.

```
Z (tau, ALam [x] e) rho
   = Fval (CtxLam c (C x e rho_c (FncC (CtxVar c))))
          (AbsLam a (Z e rho_a))
     where
        rho_c = rho {x -> FncA (CtxVar c)}
        rho_a = rho {x -> AbsVar a}
```

An Fval is returned. Its first component is a map from the function context c on (\x.e) to the context on parameter x. Bear in mind that c will get bound to a term of the form (Fnc aa cc), where aa is the abstract value supplied for x, and cc is the context on e. So the context on x is found by finding C of x in e, with rho augmented by binding x to aa, that is, to FncA (CtxVar c), and with the context on the body of the function, e, equal to cc, that is, FncC (CtxVar c).

The second Fval component maps the abstract value a of x to the abstract value of e. This is easily done by computing Z of e, with rho modified to bind x to AbsVar a.

The ACase case is quite easy:

```
Z (tau, ACase sw [(cname1, (pars1, rhs1)) ...
                  (cnamen, (parsn, rhsn))])
   rho
   = AMeet [Z rhs1 rho1 ... Z rhsn rhon]
```

The augmented environments rhoi (1 <= i <= n) are obtained by extending rho to provide bindings for parsi in

view of the value of `Z sw rho`, using the method of section 3.5.3.

Finally, the `AConstr` case. Although sections 3.5.1 and 3.5.2 completely document abstract value and context flows through constructors, we as yet have no way of creating abstract values for constructors. Starting from a general constructor application

```
C e1 ... en
```

we desire to build

```
Fval (\c1 -> f1 (FncC^n c1))
     (\a1 -> Fval (\c2 -> f2 (FncC^(n-1) c2))
     (\a2 -> ...
             ...
             ... -> Fval (\cn -> fn (FncC^1 cn))
                         (\an -> aresultant) ...))
```

where `FncC^i e` means `FncC` applied `i` times to `e`. Observe that each use of `FncC` here is of the form `FncC^i cj` where `i + j == n + 1`, and so all these terms simply denote the context on the result of the constructor application. What section 3.5.2 provides is a way to compute the `n` context maps, `f1 ... fn`. Section 3.5.1 generates a term of the form

```
\a1 ... \an -> aresultant
```

and between them, that's all that's needed. As this is rather confusing, here's a couple of examples. For `(:) :: Int -> [Int] -> [Int]`:

```
Fval (\c1 -> DropUU 1 (FncC (FncC c1)))
     (\a1 -> Fval (\c2 -> ZapWHNF (FncC c2))
                  (\a2 -> ARec [ANonRec []]))
```

For `(,) :: Int -> Int -> (Int, Int)`:

```
Fval (\c1 -> DropU 1 (FncC (FncC c1)))
     (\a1 -> Fval (\c2 -> DropU 2 (FncC c2))
                  (\a2 -> ANonRec [ANonRec [],
                                   ANonRec []]))
```

3.6.2 Definition of `C`

Propagation of a context onto a constant has no effect:

```
C x (tau, ALit n) rho alpha
   = bot (domain-of-x)
```

The variable case is:

```
C x (tau, AVar v) rho alpha
   = if    x == v
     then  alpha
     else  bot (domain-of-x)
```

As before, the application and lambda cases are a bit mind bending.

```
C x (tau, ALam [y] e) rho alpha
   = C x e rho2 (FncC alpha)
     where
        rho2 = rho {y -> FncA alpha}
```

Here, `alpha` is a function context being applied to `(\y.e)`. Assuming that `x` and `y` are not the same variable (the lambda-lifter assures this), context on `x` in `(\y.e)` can be found from the context on `x` in `e`. Since `alpha` is a function context, `FncA alpha` is an abstract value which `y` is bound to, generating `rho2`. `FncC alpha` is the context on `e` itself.

```
C x (tau, AAp f e) rho alpha
   = CJoin [ C x f alpha_f rho,
             C x e alpha_e rho ]
     where
        alpha_f = Fnc (Z e rho) alpha
        alpha_e = CtxAp (FvalC (Z f rho)) alpha_f
```

To deal with applications, observe that `x` may occur in both the function and argument expressions, so we need to collect up the contexts from `f` and `e`, and "add them together" using `CJoin`. The only problem is figuring out what context propagates to `f` and `e`. Recall that a function context consists of the abstract value of the argument, and the context on the result. Hence, the context on `f` must be `Fnc (Z e rho) alpha`. The context on `e` is equal to the context that `f` would propagate to its argument, and we know that the context on `f` is `alpha_f`. So, we build the abstract interpretation for `f` with `Z f rho`, extract the context map using `FvalC`, and apply that to `alpha_f`. All told, that's `Ctx (FvalC (Z f rho)) alpha_f`.

Due to the heroic efforts of section 3.5.4, the rather complicated `case` clause is stated quite succinctly:

```
C x (tau, ACase sw [(cname1, (pars1, rhs1)) ...
                    (cnamen, (parsn, rhsn))])
     rho alpha
   = CJoin [ C x sw rho alpha_sw,
             CMeet [ C x rhs1 rho1 alpha ...
                     C x rhsn rhon alpha
                   ]
           ]
```

Here, `alpha_sw` is the context on `sw`, given `alpha` context on the `case` expression itself, as computed by the method of section 3.5.4.

As before, the `rhoi (1 <= i <= n)` are obtained by extending `rho` to provide bindings for `parsi` in view of the value of `Z sw rho`, using the method of section 3.5.3.

Variable `x` can appear in both the switch expression, and any of the alternatives. To deal with the former, context on `sw` is computed as per section 3.5.4, and this context propagated into `sw`. Context for `x` in alterative `i` is `C x rhsi rhoi alpha`, but since we can't say which alterative will actually be selected, we must take the greatest lower bound over all alternatives. Finally, the switch and alterative contexts are once again "added" using `CJoin`. As with the `Z` clause for `case` rho is extended to provide bindings for the variables associated with each constructor.

Finally, the `AConstr` case. All the actual work of dealing with context flow through constructors is done in the corresponding `Z` clause. All we need do here is observe that `x` is never free in any constructor, and so return zero context:

```
C x (tau, AConstr c) rho alpha
   = bot (domain-of-x)
```

4 The term rewriting system

4.1 Introduction

For each Core function, the abstract interpreter produces an `AbsVal` term. Recursive groups of terms require fixpointing, which is done in a straightforward manner. The initial approximation for a function in domain `D` is `atop(D)`, so the fixpointing produces the greatest fixpoint. Although it might seem a little unusual to seek the greatest fixed point, bear in mind that this approach represents starting off from a dangerous value, `atop(D)` and iterating one's way to safety. In forward analyses in the style of [Sew91], danger is represented by the least point in the domains, and fixpointing yields the least fixed point. In any case, this discussion is rather academic, since we can claim to be looking for least fixpoints here too simply by turning all the domains upside-down – as they are finite, complete lattices, such a trick is quite allowable.

The term rewriter exists because of the need to compare approximations during fixpointing. For non-recursive terms, there is, strictly speaking, no need to use the rewriter. Nevertheless, because what emerges from the abstract interpreter is usually grossly redundant, all terms are subject to rewriting, and the recursive ones are subsequently fixpointed.

What the rewriter does is to transform each possible term into a normal form, such that semantically equivalent forms map to the same normal form. Detection of fixed points is then a simple matter of detecting syntactic equality of the normal forms. For higher order terms, unfortunately, this implies an ability to solve the halting problem. We therefore deal with higher order functions as described in section 5, and restrict ourselves to generating unique normal forms for the abstract interpretations of first order functions, something which is, fortunately, decidable.

The term rewriter proper is an elaborate system which generates normal forms by applying many local transformations to a term. When no more transformations can be applied, the term is deemed to be in normal form. Each kind of allowable transformation is encapsulated in a so-called rewrite rule. Each rule must implement a semantically invariant transformation. Section 3.2 introduced a few equalities, which, when given a directionality, become rewrite rules:

```
FncA (Fnc a c)     ===>   a
FncC (Fnc a c)     ===>   c
FvalA (Fval c a)   ===>   a
FvalC (Fval c a)   ===>   c
```

Most rules are complicated by the presence of side-conditions:

```
Fnc (FncA c1) (FncC c2)   ===>   c1
provided
   c1 == c2
```

These examples illustrate the problem of whether to simplify terms starting from the leaves (innermost-first) or from the root (outermost-first). Since, in the second example, the rule only applies if subterms `c1` and `c2` are provably equal, innermost-first rewriting seems necessary. But the same strategy applied to `FvalC (Fval c a))` could waste a lot of effort simplifying `a`, only to throw it away, so outermost-first might give better performance.

Providing the rules are finitely confluent and terminating, both approaches still give the same normal forms. Observe however that whatever approach is used, multiple passes over the tree will, in general, be needed to arrive at normal form. The decision can therefore be based purely on whichever scheme gives better performance. Experimentation showed that outermost-first rewriting was up to ten times slower than innermost-first for realistically sized terms emitted by the abstract interpreter. Although it would be foolish to claim that this is always so, the evidence suggested an innermost-first scheme would usually be much quicker, so an innermost-first scheme[1] was adopted.

4.2 Performing a single simplification pass

Because the `AbsVal` and `Context` types are mutually recursive, the term rewriter proper consists of two functions of type `AbsVal -> AbsVal` and `Context -> Context`, each of which performs multiple innermost-first simplification passes with an auxiliary function. When stability is reached, it means normal form has been achieved. This section discusses how those auxiliary functions work. For simplicity, they are treated as a single function, called `simp`, working on the union of `AbsVal` and `Context`, called `Term`.

To maximise performance, each pass of `simp` tries to do as much as possible, so as to minimise the number of passes required. Measurements showed the vast majority of terms reach normal form in one pass, and no term has been observed to require more than three passes.

The individual rewrite rules are classified into groups (represented as lists) by the root symbol of the term which they rewrite. The mechanism which directs the application of rewrite rules ensures that each rule is only applied to terms possessing the relevant root symbol. Each rule is implemented as a function of type `Term -> Maybe Term`, where:

```
data Maybe a = Nothing | Just a
```

As becomes clear shortly, we need to know whether the application of a rewrite rule has had any effect. We could make each rule have type `Term -> Term` and compare the term before and after application, but this seems abominably inefficient, because the rule itself "knows" when it has made a change. Therefore, we encode that knowledge in the return value by passing back `Nothing` if there is no change. Observe that the returned `Maybe Term` value is instantly disassembled using a Haskell case expression, to find out whether the rule has succeeded. Therefore, a Haskell implementation which returns constructors in registers, like Glasgow Haskell [PJ92], never actually builds the `Nothing` or `Just` closure in the heap, a pleasing little efficiency.

Let `t` denote a term, and `rulesfor(t)` denote the list of

[1] One of the sharper wits in the functional programming community, on reading an early draft, commented:

> How could you let such a wonderful example of self-reference go by unremarked? I thought it was absolutely marvelous that you decided to use an innermost-first scheme in the term rewriter which is, after all, the whole point of Anna's existance in the Real World outside itself!

rewrite rules relevant to the root symbol of `t`. `simp(t)` is computed as follows:

```
simp(t)
   = schedule(t_inner_simp)
     where
        t_inner_simp
           = t with simp applied to t's subterms

schedule(t)
   = rewrite_with(rulesfor(t), t)

rewrite_with([], t)
   = t

rewrite_with((rule:rules), t)
   = case (rule t) of
        Nothing -> rewrite_with(rules,t)
        Just t2 -> schedule(t2)
```

Firstly, `t`'s subterms are simplified, giving `t_inner_simp`. This is passed to intermediary `schedule`, which examines the root symbol to determine the relevant list of rewrite rules. `schedule` passes the rules and its argument to `rewrite_with`, which works its way through the list of rules. If it runs out of rules, it simply returns the term. But if there is a rule, it is applied to the term. This either has no effect, in which case the next rule is tried, or it produces a new term `t2`. Now `t2` may well have a different root symbol, which would invalidate all the remaining rules. So rewriting of `t2` is continued by passing it back to `schedule`.

The net effect of `schedule(t)` is thus to keep applying rewrite rules to the root of `t` until no applicable rules can be found. This process deals properly with changes in the root symbol. Observe that the call to `schedule` from `rewrite_with` is not necessary for correctness. We could simply return `t2` at this point. What this would mean is that any possible rewrites of `t2` would be delayed until the next simplification pass, rather than being done straight away. So omitting the re-schedule implies more simplification passes and a serious loss of efficiency.

4.3 Dealing with lambdas and applications

The presence of `AbsLam`, `AbsAp` and `AbsVar` terms introduces the need to perform lambda calculus-like substitution. What follows applies equally to the dual constructions `CtxLam`, `CtxAp` and `CtxVar`. In particular, `simp` needs to be able to deal with terms of the form `(AbsAp (AbsLam v e) a)`. Naturally, we can reach directly for the blunderbuss solution: devise a function `subst(e,v,a)` to replace free occurrences of `v` in `e` with `a`, and employ it in the rewrite rule:

```
AbsAp (AbsLam v e) a   ===>   subst(e,v,a)
```

Two defects are apparent. Firstly, since `simp` is committed to doing innermost-first simplification, both function and argument are simplified extensively before substitution begins. Our hands are now tied: we cannot make the lambda/apply term reduction any lazier. Inefficiency is the second complaint. This scheme demands a complete substitution pass over `e` for every argument.

An altogether nicer solution is to forget about `subst` and the rewrite rule. Instead, we equip `simp` with an environment `env` which binds `Abs`-variables to values. Now, give `simp` a couple of special cases. These omit the usual simplification of subterms, and bypass the general rewriting mechanism. In this way we regain precise control over the order of rewrites, and no separate substitution passes are needed. Variables are simply looked up:

```
simp env (AbsVar v) = env v
```

On encountering `(AbsAp f a)`, we need to try and turn `f` into an `(AbsLam v e)`. The obvious way to do this is by applying `simp` to `f`, but this would be a big waste of time if `f` is in that form already. So there is a special check for this case. The environment is then augmented with a binding for `v`, and simplification continues with `e`. By choosing to bind `v` to `a` or `simp env a`, we can again vary the strictness of the scheme. The latter choice gives better performance, so the special case for `(AbsAp f a)` is:

```
simp env (AbsAp f a)
   = let sa = simp env a
         sf = simp env f
     in
     case f of
        AbsLam v e
           -> simp env{v :-> sa} e
        other
           -> case sf of
                 AbsLam v2 e2
                    -> simp env{v2 :-> sa} e2
                 other
                    -> AbsAp sf sa
```

If `f` simply refuses to be rewritten into an `AbsLam`, the term has its subterms simplified and is then returned as-is. This is consistent with how normal cases are dealt with, since there are no more `AbsAp` rewrite rules.

An `AbsVar` construct can refer not just to variables bound by a surrounding `AbsLam`, but also to the abstract values of other functions. To deal with these, we "preload" the `Abs`-environment with suitable bindings before starting simplification. Finally, note that the dual `Ctx`-constructions are dealt with in the same way, so `simp` carries two environments, rather than just one. The only difference is that a `CtxVar` can only refer to `CtxLam` bound variables. These two environments are henceforth referred to as `aenv` and `cenv` respectively.

4.4 Avoiding infinite branching

4.4.1 A naive approach

Section 3.4 introduced the `CaseU` and `CaseUU` constructions as one of the fundamental mechanisms for disassembling contexts. A serious problem which becomes apparent as soon as one starts fixpointing is the potential for infinite branching. Fixpointing produces expressions like

```
CaseU e (CaseU e w x) (CaseU e y z)
```

which is equivalent to:

```
CaseU e w z
```

We can get round this by designing the normal form so that for a term `(CaseU e a b)`, neither subterm `a` nor `b` may do a `CaseU` on `e`. To achieve this normalisation requires using partial knowledge about the value of `e` when simplifying `a` and `b`.

To implement this, we could adopt the following scheme. Give `simp` yet another environment, `selenv`, which maps switch expressions seen in surrounding `CaseU` and `CaseUUs` to partial information about their value. When a nested `Case` expression is encountered, look up its switch value in `selenv`. If there is a corresponding entry, this `Case` expression must be examining a context which has already been looked at, so the `Case` expression is replaced by whichever arm the table entry says is correct. For example, given a call

```
simp selenv (CaseU e (CaseU e w x) (CaseU e y z))
```

simplification of `(CaseU e w x)` is done with `selenv` binding `e` to `Stop1`, and simplification of `(CaseU e y z)` is done with `selenv` binding `e` to some value of the form `Up1 [...]`. This partial information about `e` immediately allows the system to reduce the two subterms to `w` and `z` respectively. Propagation of information about `CaseUU` selector values is done analogously.

`selenv` is augmented each time a `CaseU` or `CaseUU` is "gone past". A problem is what happens when we go past a `(CtxLam v e)`, since this would invalidate any keys in `selenv` containing free variable `v`. Remember that the keys are arbitrary expressions, rather than mere variables. An expensive solution is to filter out all (key, value) pairs which refer to `v`, but that's overkill. It is cheaper to completely empty `selenv` at every `CtxLam`. This doesn't lose information because the abstract interpreter never builds context expressions where we need to maintain selector information across `CtxLam` boundaries. For example, it never builds anything like:

```
CaseU s1 (\c1 -> ... (CaseU s2 ....))
         (\c2 -> ... (CaseU s2 ....))
```

4.4.2 Generalising the scheme

A little thought shows our solution, whilst perfectly workable, is too weak. We need a more general way to propagate so-called "`selenv` information" around, as can be seen by considering:

```
CMeet [e, UpUp2 [Stop1, Stop1]]
```

Initially, it looks like nothing more can be done with this. But if, by looking in `selenv`, we can show that `e` has an `UpUp2 [...]` value, then:

```
  CMeet [e, UpUp2 [Stop1, Stop1]]
= UpUp2 [Stop1, Stop1]
```

What we really need is a general mechanism for propagating `selenv` information. To be fully general, we will have to search `selenv` for each term `simp` encounters. This process can be rolled into the general mechanism of `simp`, by searching `selenv` after `simp` runs out of applicable rewrite rules. We expect to discover nothing about the vast majority of terms, in which case `simp` acts as before. But, for a lucky few, `selenv` tells us a little about the term: it is either `Stop1`, `Stop2`, `Up2`, `Up1 [...]` or `UpUp2 [...]`. In the first three cases, we can obviously replace the term with the relevant value, but the other two are problematic. How can we exploit partial information like this? Conceptually, we need to add a footnote to the value saying, for example, "P.S. This value is known to be `UpUp2 [...]`", and modify the rewrite rules to take account of such footnotes.

This all sounds rather clumsy, but there is a neat solution. Recall section 3.4 introduced `DefU` and `DefUU`. `Def`s stand for "definitely", and are intended as a way of attaching such a footnote to a value. The intuitive reading of `(DefU e)` is "I'm not sure what the exact value of `e` is, but I do know it's an `Up1 [...]` value". So now, on discovering from `selenv` that a term `c` has an `Up1 [...]` or `UpUp2 [...]` value, we merely need to wrap `c` in `DefU` or `DefUU` respectively. All that remains to do is modify rewrite rules to take account of `DefU` and `DefUU` as appropriate. This mechanism subsumes the previous one. Consider again:

```
simp selenv
     (CaseU e (CaseU e w x) (CaseU e y z))
```

Ignoring possible changes to `w`, `x`, `y` and `z`, `simp` tranforms this to:

```
CaseU e (CaseU Stop1 w x) (CaseU (DefU e) y z)
```

Application of the rewrite rules

```
CaseU Stop1    a b   ===>   a
CaseU (DefU e) a b   ===>   b
```

yields the desired result:

```
CaseU e w z
```

Recall the other example, in which `selenv` binds `e` to an `UpUp2 [...]` value:

```
CMeet [e, UpUp2 [Stop1, Stop1]]
```

After wrapping `DefUU` around `e`, the following sequence of rewrites is possible:

```
  CMeet [DefUU e, UpUp2 [Stop1, Stop1]]

= UpUp2 [ CMeet [SelUU 1 e, Stop1],
          CMeet [SelUU 2 e, Stop1] ]

= UpUp2 [Stop1, Stop1]
```

Again, the desired result is obtained. All we had to do is include a rewrite rule derived from this:

```
CMeet [ UpUp2 [x1, x2],  UpUp2 [y1, y2] ]
   ===>   UpUp2 [ CMeet [x1, y1],
                  CMeet [x2, y2] ]
```

By modifying the rule so that one of the initial terms is `(DefUU e)`, and bearing in mind the meanings of `DefUU` and `SelUU` (see section 3.4), one can easily show that:

```
CMeet [ DefUU e,  UpUp2 [y1, y2] ]
   ===>   UpUp2 [ CMeet [SelUU 1 e, y1],
                  CMeet [SelUU 2 e, y2] ]
```

All in all, a rather elegant solution to a tricky problem. There is just one final caveat. Consider:

```
simp selenv (CaseU e a b)
```

If we cannot find a value for `e` in `selenv`, the `CaseU` expression may still be removable by the following means. Find in `selenv` a key `k` for which we can prove that $\texttt{k} \sqsubseteq \texttt{e}$, and for which `k` is bound to some `Up1 [...]` value. So `e` must also bind to some `Up1 [...]` value, so we can replace `(CaseU e a b)` by `(CaseU (DefU e) a b)`. `CaseUU`s are, as ever, analogous. So we might be able to do just a little bit better my taking monotonicity of keys into account when searching `selenv`.

4.5 Avoiding an exponential explosion

Although we have avoided non-termination via infinite branching, another insidious problem lurks: terms which expand exponentially for a while, before shrinking back to a compact normal form. Such behaviour causes the term rewriter to run out of memory simplifying seemingly insignificant expressions. The problem manifests itself, once again, with `CaseU` and `CaseUU` terms. The normal form requires that the switch expression cannot itself be a `CaseU` or `CaseUU`, giving rise to some rules of the form:

```
CaseUU (CaseUU a b c d) e f g
===>
CaseUU a (CaseUU b e f g)
         (CaseUU c e f g)
         (CaseUU d e f g)
```

The problem occurs because of the way `rewrite_with` attempts to apply rewrite rules to the root term until no more can be found. If `a` is itself a `CaseUU` term, `rewrite_with` will immediately reapply the rule, trebling the expression size again. It would be better to look to see if we can do some simplifications on the `(CaseUU b e f g)`, `(CaseUU c e f g)` and `(CaseUU d e f g)` terms *before* selecting another rewrite rule for the root term. There's a very good chance we can, because it is likely that we already know enough about `b`, `c` and `d` to eliminate their associated `CaseUU`s. It may also turn out that `a` is the same as `b`, `c` or `d`, and this is helpful too.

Implementing this is not only easy, but essential. When `rewrite_with` detects that a rewrite rule has created a `CaseUU` term, it does not immediately seek out another rewrite rule for the root term. Instead, it tries to rewrite the subterms as much as possible, and only then looks again at the root term. This minor modification proves very successful at avoiding exponential explosions.

4.6 Type-specific `AbsVal` optimisation

The abstract values (`AbsVals`) of all non-function-space objects are points in a one point domain. Therefore, for any `AbsVal` at all, if we can determine that the object's domain is non-functional, we can manufacture an equivalent value from `ARec` and `ANonRec`. This is extremely useful. The definition of `AbsVal`, presented in section 3.3, is augmented so we can identify the domain for any term. This is done by tagging each `AbsVal` with a context domain value, except for the `ARec` and `ANonRec` cases, where the domain is obvious.

The `simp` action for `AbsVal`s now begins by extracting the context domain, and building a literal replacement if appropriate. Let `domainof(a)` be the domain of `a`.

```
simp a
   = let a_ctx_domain = domainof(a)
     in
     if    unitary_ctx_domain(a_ctx_domain)
     then  unit_value(a_ctx_domain)
     else  (... do as before ...)

unitary_ctx_domain (Lift  (D1 x ... x Dn))
   = unitary_ctx_domain(D1) && ... &&
     unitary_ctx_domain(Dn)

unitary_ctx_domain (Lift2 (D1 x ... x Dn))
   = unitary_ctx_domain(D1) && ... &&
     unitary_ctx_domain(Dn)

unitary_ctx_domain (Ds -> Dt)
   = False

unit_value (Lift  (D1 x ... x Dn))
   = ANonRec [unit_value(D1) ... unit_value(Dn)]

unit_value (Lift2 (D1 x ... x Dn))
   = ARec    [unit_value(D1) ... unit_value(Dn)]
```

This works fine, but, as usual, we can do a little better. Presented with a term already composed entirely of `ARec` and `ANonRec`s, the scheme returns a copy of the term, giving a potential loss of sharing. A small modification detects such terms and returns them as-is.

4.7 Improving the representation of contexts: `ApplyET`

For many trivial-looking functions, the abstract interpreter emits a remarkably cumbersome and unintuitive-looking term. Examination of terms from first order functions shows a way to cut down their sizes. Since all abstract values pertaining to the arguments and result of a first order function are unitary, the only thing one can ask about it is how the context on the result propagates to each argument. Supposing we have `f`, a first order function of m arguments, and we want to know what context propagates to the n'th argument if the result is demanded in context `alpha`. At present, we get a large term of the form:

```
CtxAp  (FvalC (AbsAp (GetA ... (AbsAp (FvalA f)
                                      a1)
                             ...
                       ai)
       ))

       (Fnc aj ... (Fnc am alpha) ...)
```

where $\texttt{i} = n - 1$ and $\texttt{j} = n + 1$. What this does it to use the `(AbsAp (GetA ...))` construction $n-1$ times to supply the first $n-1$ abstract value arguments, which exposes the

```
FsqDiff
 = (Fval (\c1 -> (CJOIN
                   (ApplyET#0 F+ (ApplyET#1 F* (FncC (FncC c1))))
                   (ApplyET#0 F- (ApplyET#0 F* (FncC (FncC c1))))))
         (\a1 -> (Fval (\c2 -> (CJOIN
                                 (ApplyET#1 F+ (ApplyET#1 F* (FncC c2)))
                                 (ApplyET#1 F- (ApplyET#0 F* (FncC c2)))))
                     (\a2 -> (ANonRec [])))))

FsqDiff
 = (Fval (\c1 -> (CJOIN
                   {(FvalC F+)
                    (Fnc (ANonRec []) (Fnc (ANonRec []) {(FvalC {*(FvalA F*) (ANonRec [])*})
                                                         (Fnc (ANonRec []) (FncC (FncC c1)))})))}
                   {(FvalC F-)
                    (Fnc (ANonRec []) (Fnc (ANonRec []) {(FvalC F*)
                                                         (Fnc (ANonRec []) (Fnc (ANonRec [])
                                                                                (FncC (FncC c1))))})))}))
         (\a1 -> (Fval (\c2 -> (CJOIN
                                 {(FvalC {*(FvalA F+) (ANonRec [])*})
                                  (Fnc (ANonRec []) {(FvalC {*(FvalA F*) (ANonRec [])*})
                                                     (Fnc (ANonRec []) (FncC c2))})}
                                 {(FvalC {*(FvalA F-) (ANonRec [])*})
                                  (Fnc (ANonRec []) {(FvalC F*)
                                                     (Fnc (ANonRec []) (Fnc (ANonRec []) (FncC c2)))})})))
                     (\a2 -> (ANonRec [])))))
```

Figure 1: Abstract interpretation of `sqDiff`, with and without using `ApplyET`

n'th context map. This is then applied to `alpha` wrapped up in a chain of `(Fnc ...)` constructions which supply the remaining $m - n - 1$ abstract value arguments.

This seems an enormously wasteful way to say what amounts to:

```
ApplyET n f alpha
```

That is, "extract the n'th context map from `f` and apply it to the context `alpha`". Well, almost. In fact, `f`'s n'th context map expects to be applied not directly to `alpha`, but to the term `(Fnc aj ... (Fnc am alpha) ...)`. It looks at first like we need to include the abstract values `aj` to `am` in the `ApplyET` term. Fortunately, that is avoidable: since they are all unitary, we should never need to know what they are. Instead, we simply record in the `ApplyET` term how many of these "trailing" arguments there are. When the term rewriter finally gets hold of `f`'s n'th context map, it uses this number to build a suitable "dummy term"

```
(Fnc Error ... (Fnc Error alpha) ...)
```

to which it applies the relevant context map. In effect, we avoided storing those trailing arguments, and faked them instead, using `Error` terms, because we can guarantee they will never be used. From this it follows that it is an error for `Error` to appear in the normal form of any term.

Use of `ApplyET` shrinks many terms dramatically, and enhances the time and space performance of the analyser. As an example, Figure 1 shows the abstract interpretation of

```
sqDiff x y = (x + y) * (x - y)
```

with and without using `ApplyET`. Of course, when the definitions of `(+)`, `(-)` and `(*)` are substituted in, both terms reduce to the same thing. Note that `(CtxAp e1 e2)` and `(AbsAp e1 e2)` are written as {`e1 e2`} and {`*e1 e2*`} respectively.

4.8 Normal forms and termination properties

Showing that the term rewriting system always terminates, and produces normal forms, is important. For a system with as many rewrite rules and complications as this, producing a correctness argument is a formidable task. We hope to include one in a later version of this paper. Also to be included will be a listing of the rewrite rules, along with their associated proofs of correctness. There are at present in the region of sixty rewrite rules.

5 Firstification and monomorphisation

5.1 Introduction

Although first order functions are easily handled by term-rewriting based fixpointing, higher order recursive functions give trouble, as typified by foldr:

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

Naively fixpointing this gives an series of approximations in which a term involving functional parameter `f` is applied ever more times to an initial term. The term rewriter cannot

show that two approximations are the same, so a fixpoint is apparently never reached. What's really going on is that the fixpoint of `foldr` depends on the fixpoint of `f`.

There are only two ways round this. The first is to iterate enough times to be sure that the fixpoint is certainly reached. Work by Nielson and Nielson [NN92] gives safe lower bounds on the number of iterations needed. Unfortunately, the expense of doing this makes it unattractive. Note also that this approach demands monomorphisation.

The second solution requires us to supply a value for `f` before fixpointing `foldr,` so that we are, in effect, no longer dealing with a higher-order function. There are numerous ways to do this, some rather obscure in that they partially substitute in functional parameters as part of the fixpointing process [**?**]. By contrast, Anna adopts a completely straightforward approach: transform the source program. Program transformation is a popular subject, and various papers describe higher-order function removal (also known as firstification or specialisation) [CD91] [Nel]. The scheme presented below is based on work by George Nelan [Nel].

Not all functional parameters can be removed. For example, recursive functions which have accumulating functional parameters are not transformable, at least with the scheme below:

```
f g x = if    x == 0
        then  g 1
        else  x * f (\y -> g x + y) (x-1)
```

We justify this design decision on the basis that the vast majority of functions that people really write can be firstified, and the vast majority of the rest can be handled by a secondary mechanism outlined in section 6.4. In doing this we implicitly appeal to the measurement-lead approach to design discussed in the introduction. We only need to remove functional parameters for recursive functions, but, as becomes clear, this means firstifying non-recursive functions too.

5.2 Firstification by examples

For the moment, let's use `foldr` as a running example. Given a use of `foldr`, like

```
sum xs = foldr (+) 0 xs
```

we can unfold functional parameter `(+)` and identity `0` into `foldr`, giving a new function `foldrSpec`:

```
sum xs = foldrSpec xs

foldrSpec []      = 0
foldrSpec (x:xs)  = x + foldr (+) 0 xs
```

And now, folding the body of `foldrSpec` gives what we really want:

```
sum xs = foldrSpec xs

foldrSpec []      = 0
foldrSpec (x:xs)  = x + foldrSpec xs
```

The key to success here is the ease with which that last fold was done. In general, folding is a tricky business, with no guarantee of termination. However, by restricting the functions we deal with, we can guarantee to make the fold step terminating, and trivial to carry out. The restriction is that the function must pass along all parameters which we want to specialise in the same position in recursive calls as they appeared in the arguments. For example, in `foldr`, both `a` and `f` satisfy this.

In fact, we only want to substitute in functional parameters. So the transformation of `sum` is really:

```
sum xs = foldrSpec 0 xs

foldrSpec a []      = a
foldrSpec a (x:xs)  = x + foldrSpec a xs
```

A little terminology. The function or recursive group of functions - for example, `foldr` - for which functional parameters are being removed is called the **target group**. And a function containing a call to a target group is called a **source**.

The restriction on valid targets seems to be this: the function must pass along all functional parameters unchanged in all recursive calls. Generalising this to deal with mutually recursive targets requires the notion of a **constant argument set**. Calling a recursive group in general causes calls within the group, and a constant argument set gathers together arguments which are guaranteed to have the same value for every sub-call. For example, given

```
f x y z  = f y y z + g z x
g a b    = f a b a + g a b
```

a little thought shows `f`'s third argument and `g`'s first argument are always the same, giving a constant argument set written `{f.3, g.1}`. A group can have more than one set, as the following trivial example shows:

```
f x y  = g x y
g a b  = f a b
```

Constant argument sets can be computed using a simple abstract interpretation described below. What is important here is that for a recursive target group to be specialisable, all functional parameters in the group must be constant arguments. Certain other constraints also apply. We return to these later.

The following example breaks our nice scheme:

```
map f []      = []
map f (x:xs)  = f x : map f xs

addN n xs = map (+ n) xs
```

Folding and unfolding as above gives

```
addN n xs = mapSpec xs

mapSpec []      = []
mapSpec (x:xs)  = (x+n) : mapSpec xs
```

which is clearly wrong because `n` is undefined in `mapSpec`. What we need to do is pass along all free lambda-bound variables in the specialising value `(+ n)` as new parameters, in a style reminiscent of lambda-lifting:

```
addN n xs = mapSpec n xs

mapSpec n []      = []
mapSpec n (x:xs)  = (x+n) : mapSpec n xs
```

Nevertheless this still allows us to get our knickers in a twist. In the following (admittedly contrived) example, we may select either `inc` or `g` as a source to transform first:

```
apply f x  = f x
g a b      = apply a b

inc y      = g (+ 1) y
```

Doing g first gives:

```
applySpec x  = a x
g a b        = applySpec b

inc y        = g (+ 1) y
```

Now we need to introduce free variables of the specialising value `a` as a new parameter to `applySpec`:

```
applySpec a x  = a x
g a b          = applySpec a b

inc y          = g (+ 1) y
```

Whereupon it should be eminently clear that we've achieved exactly nothing! Our mistake was to select a source for which the specialising value had function-valued free variables, since passing them as new parameters to `applySpec` means `applySpec` is still a higher-order function. The moral is clear: only transform sources for which the free lambda-bound variables of the specialising value are not higher-order. A second obvious constraint on source calls is that the call should be sufficiently applied for all function-valued parameters to be visible.

In what follows, we assume that naming issues are dealt with correctly. In particular, the free variables in an specialising value need to be renamed before they can be safely inserted as new parameters of the specialised target.

The algorithm below requires the program to be stratified into minimal mutually recursive groups, and topologically sorted. We adopt the usual convention that the program is written top-to-bottom, with any given function referring only to its own group, if recursive, and groups above it. The root expression is right at the bottom.

Specialisation of target groups may result in some groups becoming unreachable from `main`. These groups should be removed. A more difficult problem is how to insert a new group, resulting from specialisation of an existing group, into the program so as to maintain dependancy. It turns out that two different behaviours are possible. In the usual case, the new group "splits" away from the source group:

```
letrec    map f []      = []
          map f (x:xs)  = f x : map f xs
in let
          square x      = x * x
in let
          squareList xs = map square xs
in ...
```

Since the specialised target `mapSpec` refers to `square`, we must place it after `square`. Taking this argument to its conclusion shows we should place any "splitting" group immediately before the source group:

```
let       square x        = x * x
in letrec
          mapSpec []      = []
          mapSpec (x:xs)  = square x :
                              mapSpec xs
in let
          squarelist xs   = mapSpec xs
in ...
```

Sources giving rise to "joining" specialisations are unusual:

```
letrec    map f []      = []
          map f (x:xs)  = f x : map f xs
in letrec
          squareList xs
             = map (head.squareList.unit) xs
in ...
```

Here, the definitions of `(.)`, `head` and `unit` are unimportant. Because the specialising value `(head.squareList.unit)` refers to the source group from which it originates, the resulting specialisation of `map` will also refer to `squareList`, and that in turn means the specialisation should be merged into the source group:

```
letrec
   squareList xs   = mapSpec xs
   mapSpec []      = []
   mapSpec (x:xs)  = (head.squareList.unit) x :
                     mapSpec xs
in ...
```

Since only recursive source groups can refer to themselves, specialisations corresponding to sources in non-recursive groups never exhibit this "joining" behaviour.

5.3 An algorithm

The procedure below is repeated until no more valid (source, target) pairs can be found. As shown by Nelan [Nel], the order in which these pairs are selected makes no difference. Unused targets should be deleted, but again it doesn't make any difference when. As a running example, we take:

```
letrec
   map1 f g (x:y:xys)  = f x : g y : map2 g f xys
   map1 f g _          = []

   map2 g f (x:y:xys)  = g x : f y : map1 f g xys
   map2 g f _          = []
in let
   addmul p q xs       = map1 (+ p) (* q) xs
```

1. Find a valid target group, and a valid source group which refers to the target group.

   ```
   Target group   = {map1, map2}
   Source group   = {addmul}, refers to map1
   ```

2. If the target group is recursive, compute its constant argument sets (see section 5.4). In reality, this computation has to be performed at step (1). If non-recursive, manufacture a "fake" singleton set listing all higher-order parameters as constant.

```
Const arg sets  = { {map1.1, map2.2},
                    {map1.2, map2.1} }
```

3. Invent a new set of names for the specialised target group.

```
New names  = {map1Spec, map2Spec}
```

4. Determine, from the constant argument set, which arguments in the source call site are the specialising values. Extract their free lambda-bound variables and rename.

```
Original:
 specialising values = {(+ p), (* q)}
 free variables      = {p, q}

Renamed:
 specialising values = {(+ pnew), (* qnew)}
 free variables      = {pnew, qnew}
```

5. Rebuild the source call by deleting the specialisation values, inserting free variables as new parameters, and changing the called function to its new name, as determined in step (3).

```
Rebuilt source call  = map1Spec p q xs
```

6. Build the specialised target group, starting with a copy of the original target group. For each recursive call inside the group, modify that call in a similar way to how the source call was modified in step (5).

```
Rebuilt target group:

map1Spec pnew qnew (x:y:xys)
   = (x + pnew) : (y * qnew) :
                  map2Spec pnew qnew rest
map1Spec pnew qnew _
   = []

map2Spec pnew qnew (x:y:xys)
   = (x * qnew) : (y + pnew) :
                  map1Spec pnew qnew rest
map2Spec pnew qnew _
   = []
```

7. Determine whether the specialised target group should split or join, by finding out whether the specialisation values contained any reference to the source group. Augment program appropriately.

```
Specialisation vals {(+ p), (* q)} don't
refer to {addmul}, so new group splits.
```

A valid non-recursive target group must consist of a single higher order function. A valid recursive target group satisfies all the following:

- All functions in the group have at least one functional parameter.
- Each functional parameter in the group is a member of exactly one of the group's constant argument sets. This implies that all intra-group calls must be sufficiently applied to expose all functional arguments.
- Each constant argument set must mention exactly one argument for each function in the group. This disallows certain contrived pathological cases which would otherwise seriously complicate the algorithm.

A valid source call site satisfies the following:

- The site is a call to a valid target group.
- The application must have sufficient arguments to supply all higher order (specialisable) arguments.
- For each specialisable argument, none of the free lambda-bound variables may be, or contain, a function space.

Although it looks easy on paper, implementing this algorithm is tricky. Taking into account the mechanisms for detecting constant arguments and maintaining type annotations, the Haskell implementation approaches 1500 lines of code.

5.4 Computing constant argument sets

A simple abstract interpretation is used. Each function call in the group is abstracted to expose the parameters it passes along:

```
f x y z  = f y y z + g z x
g a b    = f a b a + g a b
```

Phrased abstractly, this becomes:

```
f: calls f [#2, #2, #3]
   calls g [#3, #1]

g: calls f [#1, #2, #1]
   calls g [#1, #2]
```

Now we iterate to a fixed point, gathering a complete set of the possible values of each argument. There is a list for each function, and each list contains a set of possible values for each argument. Initially, each argument can only be itself:

```
F0 = [ {f.1}, {f.2}, {f.3} ]
G0 = [ {g.1}, {g.2} ]
```

At each iteration, new approximations are computed by using the abstract versions of functions to look up possible argument sets in the existing approximation. Also, the existing approximation is merged in wholesale:

```
F1 = F0 U [ {f.2}, {f.2}, {f.3} ]
        U [ {g.1}, {g.2}, {g.1} ]
   = [ {f.1, f.2, g.1}, {f.2, g.2}, {f.3, g.1} ]

G1 = G0 U [ {f.3}, {f.1} ]
```

```
        U [ {g.1}, {g.2} ]
   = [ {f.3, g.1}, {f.1, g.2} ]

F2 = F1 U [ {f.2, g.2}, {f.2, g.2}, {f.3, g.1} ]
          [ {f.3, g.1}, {f.1, g.2}, {f.3, g.1} ]
   = [ {f.1, f.2, f.3, g.1, g.2},
       {f.1, f.2, g.2}, {f.3, g.1} ]

G2 = G1 U [ {f.3, g.1}, {f.1, f.2, g.1} ]
        U [ {f.3, g.1}, {f.1, g.2} ]
   = [ {f.3, g.1}, {f.1, f.2, g.1, g.2} ]
```

Eventually this process stabilises, giving the following possible argument values:

```
f.1  could have value  f.1, f.2, f.3, g.1, g.2
f.2  could have value  f.1, f.2, f.3, g.1, g.2
f.3  could have value  f.3, g.1
g.1  could have value  f.3, g.1
g.2  could have value  f.1, f.2, f.3, g.1, g.2
```

So we have two candidate constant argument sets: `{f.1, f.2, f.3, g.1, g.2}` and `{f.3, g.1}`. We reject the first because it does not mention each function exactly once. Deducing that `{f.1, f.2, f.3, g.1, g.2}` is a constant argument set is correct, but only under the conditions that, for the initial call into the group, `f.1 == f.2 == f.3`, if `f` was called, or `g.1 == g.2` for an initial call to `g`. This leaves `{f.3, g.1}` as the sole constant argument set for this recursive group.

A glaring flaw is the inability to abstract function calls which do anything more than pass parameters unchanged. In this case, a special value `Unknown` is used to denote that we cannot be sure what the value of this argument is. For example

```
f x y = f x (y+1) + f y x
```

abstracts to

```
f: calls f [#1, Unknown]
   calls f [#2, #1]
```

During fixpointing, `Unknown` annihilates any other values in a set. For example, a set `{f.1, g.2, Unknown}` is equivalent simply to `{Unknown}`. `Unknown` represents an argument of uncertain origin, so we disallow any constant argument set containing it.

5.5 Preserving type annotations

Since Anna works with type annotated Core expressions, we need to go to a little trouble to transform the annotations too. At first glance, this looks like a simple matter of modifying function types pertaining to specialised functions, by deleting types of specialised arguments and inserting the types of free variables being passed as extra parameters. This is indeed correct, but there's more to it. Recall the previous definition of `foldr`. The typechecker infers:

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

Given the usual Haskell definition of `(++) :: [c] -> [c] -> [c]`, we can define

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

which specialises to:

```
foldrSpec a []      = a
foldrSpec a (x:xs)  = x ++ foldrSpec a xs

concat = foldrSpec []
```

Merely adding and deleting argument types gives `foldrSpec` an apparent type `b -> [a] -> b`, which is too general. We need to unify the type of specialising value `(++)`, `[c] -> [c] -> [c]` with the type of the functional parameter it replaces, `a -> b -> b`, and apply the resulting substitution `{a -> [c], b -> [c]}` to the annotations on `foldrSpec`. This gives `foldrSpec :: [c] -> [[c]] -> [c]`, as required.

Such trickery should not come as a complete surprise. After all, the Milner-Hindley type rules for an application of `f :: (T1 -> T2)` to `a :: T3` require unification of `T1` with `T3`. That's effectively what is going on here.

The need to preserve type annotations is a major nuisance from the implementation viewpoint, because more time is spent in fixing up types than doing the transformation proper. Work to improve efficiency is a priority. The scheme described above is a first implementation in which correctness was more important than efficiency.

5.6 Monomorphisation

By comparison with firstification, monomorphisation is simple. Monomorphisation is a two-phase process. The first pass conducts what amounts to a depth-first search from `main` to discover all required instances. The second pass clones code, changes function names accordingly and restores dependancy order.

5.6.1 Collecting the instances

I am indebted to Mark Jones for suggesting the following algorithm. We carry a set `instances` to accumulate the eventual result, and a stack `toVisit` recording places we need to visit. Elements of `instances` and `toVisit` are pairs of (function name, type expression) specifying a particular instance of a function. The type expressions are always monomorphic.

Since `main` may be of any type, we trivially monomorphise it by substituting any type variables with `Int`. This gives a single initial value for `toVisit`, with `instances` initially being empty. The final value of `instances` is then `search(instances, toVisit)`, where:

```
search(instances, toVisit)
   = if    [toVisit is empty]
     then  instances
     else
     let   next = head toVisit
     in    if    next `elem` instances
                 [We've already been here]
     then  search(instances, tail toVisit)
     else  [Get the function specified by next.
            Find out what instances of other
```

```
            functions are called.  Add these
            instances to (tail toVisit) giving
            toVisitAug, and then do]
            search({next} U instances, toVisitAug)
```

For example, given

```
id x   = x
f x    = id x
main   = id 42 + ord (f 'c')
```

the algorithm runs through these states:

```
     instances                        toVisit
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

{}                                [(main, Int)]

{(main, Int)}                     [(id,   Int -> Int),
                                   (f,    Char -> Char)]

{(main, Int),                     [(f,    Char -> Char)]
 (id,   Int -> Int)}

{(main, Int),                     [(id,   Char -> Char)]
 (id,   Int -> Int),
 (f,    Char -> Char)}

{(main, Int),                     []
 (id,   Int -> Int),
 (id,   Char -> Char),
 (f,    Char -> Char)}
```

This gives two instances for `id`. Because the abstract interpretation of types maps both `Int` and `Char` to the two point domain, only one of those instances is needed for analysis purposes. In general, we can exploit the fact that many different types are assigned the same domain to reduce the program expansion caused by monomorphisation. Far and away the easiest way to do this is to transform the type annotations to domain annotations before monomorphisation, using the results of section 2.2.5. Type variables are simply replaced by domain variables.

5.6.2 Cloning code

This is fairly trivial. Each function is duplicated once per instance, with appropriate new names. The function bodies have their call sites modified to refer to appropriately named instances in previous groups, and this one, if recursive. Then the bodies have the type variables in their annotations substituted appropriately for each different instance required. The entire program can be processed in a single top-to-bottom pass.

The only slightly tricky problem is rebuilding recursive groups so as to maintain dependancy. For non-recursive clones, this is easy, but because of the existance of certain contrived recursive functions, maintaining dependancy in the recursive case can be complicated. We avoid these problems by lumping all the clones arising from a recursive function group into a single `letrec`, and passing the program a third time though the dependancy analyser.

Despite these complications, the monomorphiser is extremely quick and does not prove a significant limitation on performance.

6 Discussion

This section draws together the detailed technical threads expounded in the previous three sections, by presenting some performance results, and looking at related and further work. But we begin by looking at some performance issues.

6.1 Putting it all together

Throwing realistically sized programs at the analyser revealed some performance problems which were traced to niceties in the interface between the abstract interpreter and the term rewriter.

Performance problems appear when the term rewriting system is fed a gigantic term to simplify. Usually, such terms reduce to something quite trivial. It is important to realise that the abstract interpreter will generate absolutely enormous terms, especially from source text which has deeply nested function calls or deeply nested `case` expressions, both of which are quite common. For example, the bigger programs mentioned in section 6.2 generated terms which, when pretty-printed in the style used in this paper, covered literally tens of A4 pages.

Analysis of a program proceeds as follows. First, the program is passed in its entirety through the abstract interpreter, generating a corresponding collection of recursive equations (or terms). These equations are hugely redundant and are simplified individually, without reference to each other. Finally, the fixpointing system travels along the groups of equations, accumulating an environment of "solved" equations. A solved equation is self-contained: it does not refer to the abstract interpretation of any other function. Solving non-recursive equations is a simple matter of substituting in the abstract interpretations of other functions, and simplifying. For recursive functions, this stage is followed by fixpointing.

What really ruins performance is not fixpointing, but the initial simplification of terms which emerge from the abstract interpreter. This problem was largely alleviated by giving the abstract interpreter a little more intelligence, in the form of:

- The type-specific `AbsVal` rewrites described in section 4.6.
- A knowledge of key `Context` rewrite rules. In particular, the rules for `case` statements generate large expressions involving `CMeet`, `CJoin`, `Stop1` and `Stop2`. A handful of rules embodying simple facts such as `CMeet [... Stop2 ...] == Stop2` were added.

  Between them, the sizes of terms generated were sometimes cut by two orders of magnitude, and performance was much improved.

A second performance problem was traced to the initial simplification of terms emerging from the abstract interpreter. These terms are simplified without reference to each other. It turns out to be better to simplify a term only when we know the solved values of the other terms it refers to, because knowing these values makes the final term much smaller. In effect, this is achieved simply by omitting this simplification pass altogether. For at least one input, analysis time was cut by a factor of five.

Program	Lines	For a complete run			Analysis only		Compiling with `ghc-0.10`	
		Time	Claim	Space	Time	Claim	Time	analysis time as %
concat	< 10	0.42 s	1.485 M	47.2 k	0.18 s	0.586 M	0.92 s	19 %
zip3	< 10	0.52 s	1.793 M	56.1 k	0.17 s	0.570 M	1.00 s	17 %
wang	385	23.62 s	85.396 M	908.8 k	9.15 s	15.861 M	43.61 s	21 %
wave4main	619	43.40 s	116.207 M	2897.7 k	21.62 s	43.239 M	199.29 s	11 %
ag2hs	1047	208.95 s	275.245 M	9653.8 k	126.42 s	135.335 M	100.68 s	126 %

Table 2: Some performance figures for **Anna**

6.2 Absolute performance results

Five test programs were used. The first two, `concat` and `zip3`, are utterly trivial and were included as comparison against figures presented in [Sew93]. `wang` and `wave4main` are taken from Pieter Hartel's benchmark suite [HL92]. The biggest one, `ag2hs`, is preprocessor for a dialect of Haskell augmented with attribute grammar [Joh87] facilities, written by David Rushall. The analyser was compiled with Chalmers Haskell-B 0.999.4, with flags `-fpbu -O`, and run using an eight megabyte heap for all except `ag2hs`, which required a sixteen megabyte heap. A generational garbage collector was employed. Tests were run on a lightly loaded Sun Sparc 10/31, and each test was performed at least three times. Times are user CPU seconds.

Simply measuring overall run times is not particularly helpful, because we really want to establish how expensive this analyser would be if employed in a Haskell compiler. It seems reasonable to consider the "border" between the front end and the analyser itself as the point where the typechecker produces a type-annotated Core tree, since, at least in Glasgow Haskell, the compiler produces this tree anyway. So we present figures not only for a complete run, but also for the analysis phase proper. The latter category covers firstification, monomorphisation, abstract interpretation and fixpointing, all of which are legitimate analysis expenses.

To assess whether or not we are approaching the right ballpark, we timed Glasgow Haskell 0.10 compiling the programs into C, and compared those times with the analysis phase time of Anna. The compiler options were `-O2 -C`. Compiler semispace sizes were 3 megabytes for the small two, and 8 megabytes for the big three: this turned out to be plenty. The times reported for Glasgow Haskell are for the compiler proper, that is, that part of the compiler which is itself written in Haskell, and which translates the output of the Yacc parser into C.

Table 2 presents the figures. The maximum residency figures were obtained using a copying collector with heap sizes set only just big enough. This quantity is omitted for the analysis-only figures because of the difficulties of deciding on how to divide space expenses between the front end and the analysis phase.

For the big three, times are, very roughly, divided equally between the front end and analysis phases. `ag2hs` has a relatively large analysis time in comparison to its size. This is because it makes considerable use of lazy pattern matching, which translates to a large quantity of complex Core expressions. These in turn generate some large, complex sets of equations for the fixpointer to solve. A technique mentioned in section 6.4 might help here. For the larger problems, space consumption is of concern. Much, if not the majority, of the space used is related to front-end processing, and it seems likely that the analysis itself is relatively cheap on space. Further investigation with a heap profiler is necessary.

The results of comparing analysis time with a run of Glasgow Haskell on the same program are intriguing. The tests are at least fair in the sense that both Anna and the Haskell compiler are written in Haskell, so neither has an unfair advantage. Just by themselves, it is a little unusual that `ghc` compiled `ag2hs` in almost half the time it took for `wave4main`. It may be that the heavy use of numeric overloading in `wang` and `wave4main` has slowed down `ghc` as it will have had to generate and optimise large quantities of dictionary handling code. `ag2hs`, by comparison, is mostly string handling: there is little overloading in it. Anna has a naive view of the Haskell numbers – it only knows about `Int`, so it will not have seen any such numeric overloading. In order to make Anna accept these two programs, we had to strip out the extensive type signatures which had been placed there expressely to eliminate numeric overloading. These factors may well have conspired to give Anna a remarkably good relative showing for `wang` and `wave4main`, although it is hard to believe they account for all the difference between 11% (`wave4main`) and 126% (`ag2hs`).

Because `wang` and `wave4main` are machine-generated Haskell, the expressions in them are reasonably simple and small. By comparison, the desugared version of `ag2hs` contained some very large expressions and some quite complicated structured types. Watching the behaviour of Anna on this example, it is clear that the majority of the analysis time is spent fixpointing a single large group of about twenty functions which arose from the extensive use of lazy pattern matching. It seems plausible that this particular group did not cause any similar difficulty to `ghc`, and it may also be possible that `ghc`'s desugarer did a better job than Anna's in translating the pattern matching. Nevertheless, the disparity in relative analysis/compile costs between `ag2hs` and the other two big examples is a warning that we should not read too much into these measurements beyond the perhaps heartening conclusion that we are indeed approaching the right ballpark for analyser performance.

6.3 Related work

Mycroft's original work [Myc80] on applying abstract interpretation to the analysis of functional programs sparked off intense work on forward analyses. A forward strictness analysis tells us the definedness of a function application given the definedness of the arguments. Landmark papers include the Burn-Hankin-Abramsky work [BHA85] which put higher order analysis on a firm theoretical footing, and Wadler's paper [Wad87] which showed how one might deal sensibly with sum-of-products types. Implementors made much of finding fixpoints using the Frontiers algorithm, massaging it extensively to deal with higher order functions [HH91], sum-of-products types [Sew91] and polymorphism [Sew93]. Despite this and other trickery [HH92] [Sew92], frontiers failed to deliver usable performance for high-definition strictness analysis for anything other than trivial inputs, and there are good theoretical reasons for believing the situation cannot be improved.

Starting at around the same time, another school of thought was developing backwards, or projection, analyses. A backwards analysis shows how the semantic quantity in question - here, demand for evaluation - propagates from a function application to the individual arguments. Hughes [Hug90] argues that backwards analyses are inherently more efficient than forward ones, because the function spaces with which the analyses deal are smaller in the backwards case. Projection analysis deals easily with sum-of-products types, and captures certain properties, such as head-strictness, that seem to elude forward analyses. A good reference for projection analysis is [WH87]. Later work showed how to do make non-flat projection analysis polymorphic [Hug], and a successful non-flat, polymorphic projection analyser was built into Glasgow Haskell [KHL91].

Despite these successes, projection analyses have a fundamental inability to deal with higher order functions. Following the lead of Wray [Wra85], Hughes defined a mixed analysis which was forwards for the higher order bits and backwards for everything else [Hug87]. Doing this gives an analysis which deals with higher-orderness whilst retaining the inherent efficiency of backward analysis. Recently, other workers have begun to explore the relationship between forward and backward analysis [Bur90] [HL90] [DW90]. The analysis described in this paper is a modification of Hughes' original mixed analysis.

Meanwhile, people have been looking at other ways of solving recursive domain equations. There has been a discernable shift towards term oriented approaches. Ferguson and Hughes developed "concrete data structures" (CDSs) [**?**] based on Curien's work on sequential algorithms [Cur86]. CDSs deal with higher-orderness by regarding a higher order function as containing a CDS interpreter for each functional parameter. This is really a disguised way of substituting in functional parameters before fixpointing. Whether or not CDSs can deliver a viable fixpointing mechanism remains to be seen. Early implementations hinted at space problems, but these may now have been solved [Hug93]. CDSs can also be viewed as a higher-order generalisation of the minimal function graph scheme originally described by Neil Jones [JM86]. Minimal function graphs are used in the Semantique analyser [KHL91] built into Glasgow Haskell [PHHP93].

The term rewriting based fixpointer described here was, in part, inspired by Charles Consel's strictness analyser in the Yale Haskell compiler [Gro92]. Consel's paper [Con91], which seems to have passed by almost unnoticed, described a successful, if simple, strictness analyser solving fixpoint equations by term rewriting. In view of how well this and Consel's system work, it is perhaps a pity that Peyton Jones et al made disparaging remarks about term-based fixpointing in their seminal frontiers paper [PC87].

6.4 Further work

Anna's performance is encouraging. Nevertheless, there's still a long way to go before evaluation transformer information can be generated automatically in production compilers. Three avenues of development need to be persued.

- **Enhancement of applicability.** Anna's most worrying limitation is her inability to deal with higher order functions which cannot be firstified. A possible partial solution is to iterate these (or, more precisely, just the nasty bits) as many times as is necessary to guarantee a fixpoint. The work of Nielson and Nielson [NN92] gives the magic number of iterations needed. For many common forms, this number is reasonably low, and it seems reasonable to expect this approach to yield worthwhile results.

  It is also necessary to remove some of the excessive restrictions on user-defined data types discussed in section 2.2.6. This does not appear to be particularly difficult. Kubiak et al [KHL91] managed this quite successfully.

- **Enhancement of performance.** The refinements of section 6.1 have done a lot to improve the system's performance. Nevertheless, some programs we tried recently - in excess of a thousand lines - run more slowly than one would like. Investigations are being made.

  Fixpointing large groups of functions could concievably be accelerated by reducing the group to a "minimal form" first. For example, given

  ```
  a = ... a ... b ...
  b = ... c ...
  c = ... c ... d ...
  d = ... a ...
  ```

  we can remove `b` and `d` by substituting them into `a` and `c` respectively. This halves the number of functions in the group being fixpointed. Once the solutions to `a` and `c` have been generated, we obtain values for `b` and `d` by straightforward back-substitution.

  Note that this technique may be used in any situation involving fixpointing mutually recursive groups of equations. The idea stems from an analogy with the Gauss-Jordan method for solving simultaneous linear equations. In this case, a recursive group can only be reduced to the point where every equation in the group refers directly to itself – no further. After that, fixpointing is unavoidable. Whether or not this renders a speedup depends on the relative costs of substitution, back substitution and fixpointing.

- **Dealing with modules**. Modules are an unmitigated nuisance for many kinds of high powered semantic analyses and optimisations. In particular, modules cause big difficulties for any kind of what John Young termed "collecting interpretations" [You89]. A collecting interpretation is essentially a global analysis. Many compile time optimisations are limited by the module structure. For example, some of the more recent schemes for compiling overloading efficiently [Jon93] [Aug93] require global analysis for full applicability. The point of all this is that the monomorphisation and firstification transformations used in Anna also require a global view.

  There is an urgent need to devise sophisticated compilation systems which maintain enough intermodule communication to make global analyses possible. Development of such a framework would benefit not only strictness analysis, but many aspects of compile time optimisation. Such a compiler might work by dumping a lot of information into a module's interface file, enough to do whatever analyses we need. This would really just be an extension of the schemes used already in the Chalmers and Glasgow compilers, which dump function arity and rudimentary strictness information into interface files. The question is not really whether we could construct such a system, but whether the quantity of information dumped into interface files could be limited sufficiently to render the scheme practical.

Acknowledgements

Thanks to Bill Mitchell for advice on building term rewriting systems, and to Barney Hilken for an invaluable insight regarding separate compilation systems. Mark Jones provided many interesting comments about monomorphisation and firstification, and outlined the instance-collecting algorithm of Section 5.6.1. Geoffrey Burn and Denis Howe were sufficiently brave to experiment with the implementation, and provided useful feedback.

Denis Howe read an early draft in minute detail. His extensive and sometimes amusing[2] comments proved very helpful in making the presentation clearer.

References

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II*. PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.

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

[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.

[Bur90] G.L. Burn. A relationship between abstract interpretation and projection analysis. In *17th Annual ACM Symposium on the Principles of Programming Languages*, pages 151–156, San Francisco, 17–19 January 1990. ACM.

[Bur91] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Research Monographs in Parallel and Distributed Computing. Pitman in association with MIT Press, 1991. To appear.

[CD91] Wei-Ngan Chin and John Darlington. Removing higher-order expressions by program transformation. Chin is at NUS, Singapore and Darlington at Imperial, London. Possibly published, February 1991.

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

[Cur86] P.-L. Curien. *Categorical Combinators, Sequential Algorithms And Functional Programming*. Research Notes in Theoretical Computer Science series. Pitman Publishing Limited, London, 1986.

[DW90] Kei Davis and Philip Wadler. Strictness analysis in 4d. In *In proceedings of the 1990 Glasgow Ayr FP Workshop (??)*, 1990.

[Gro92] The Yale Haskell Group. The yale haskell users manual, version y2.0-beta, August 1992.

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

[HL90] R.J.M. Hughes and J. Launchbury. Towards relating forwards and backwards analyses. In *Proceedings of the Third Annual Glasgow Workshop on Functional Programming*, pages 145–155, Ullapool, Scotland, 13–15 August 1990.

[HL92] 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.

[2] See the other footnote.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[Hug] John Hughes. Projections for polymorphic strictness analysis. In *In Springer Verlag LNCS 389*. Year and conference not established.

[Hug87] John Hughes. Backwards analysis of functional programs. Technical Report CSC/87/R3, University of Glasgow, Department of Computing Science, March 1987.

[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.

[Hug93] John Hughes. Private communication regarding cdss, 1993.

[JM86] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs: Abridged version. In *Unknown, but definitely in an ACM proceedings*, 1986.

[Joh85] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, Nancy, France, September 1985.

[Joh87] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 154–173. Springer-Verlag LNCS 274, September 1987.

[Jon93] Mark Jones. Partial evaluation for dictionary-free overloading. Unpublished draft. Yale University, Department of Computer Science, 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.

[Min92] Sava Mintchev. A parallel stg-machine. Master's thesis, Department of Computer Science, University of Manchester, UK, September 1992.

[Myc80] A. Mycroft. Theory and practice of transforming call-by-need into call-by-value. In *4th International Symposium on Programming*, pages 269 –281, Paris, April 1980. Springer-Verlag LNCS 83.

[Nel] George C. Nelan. Firstification. Date unknown, but must be 1992 or after. Based on Nelan's PhD thesis. Arizona State University.

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

[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.

[Pey87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd, London, 1987.

[PHHP93] S.L. Peyton Jones, Cordelia V Hall, Kevin Hammond, and Will Partain. The glasgow haskell compiler: a technical overview. In *Proceedings of the UK Join Framework for Information Techology, Keele*, 1993.

[PJ92] S.L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, April 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.

[Sew92] Julian Seward. Polymorphic, higher order strictness analysis using frontiers. Unpublished paper, 1992.

[Sew93] Julian Seward. Polymorphic strictness analysis using frontiers. In *Proceedings of the Symposium on Partial Evaluation and Semantics based Program Manipulation, PEPM93, Copenhagen, Denmark*, June 1993.

[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.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico*, 1992.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 385–407. Springer-Verlag LNCS 274, September 1987.

[Wra85] S. C. Wray. A new strictness detection algorithm. In *Proceedings of the Workshop on Implementations of Functional Languages, Aspenas, Sweden. Available as Chalmers PMG report 17*, 1985.

[You89] J.H. Young. *The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages.* PhD thesis, Department of Computer Science, Yale University, May 1989.