Strictness Analysis — Another Method

Michael Winikoff
Supervisor: Dr. Harald Søndergaard

Honours Thesis
November 11, 1993
Department of Computer Science
University of Melbourne
Parkville 3052
Australia
winikoff@cs.mu.oz.au

Abstract

Strictness analysis is a compile time analysis applicable to lazy functional programming languages. Strictness analysis can be used to significantly improve uniprocessor implementation speed and to derive non-speculative parallelism.

Many methods for doing strictness analysis have been proposed. Most of these are based on abstract interpretation and require fixpoint iteration.

In this thesis we survey existing methods and then go on to define a new method which does not require fixpoint iteration.

The new method is significantly more efficient than methods based on abstract interpretation. The abstract domain used is not fixed and can be expanded to obtain better accuracy. Our method is also easy to prove correct.

We discuss further work necessary for the new method to realise its potential as a framework for *fast* and *practical* analyses.

Acknowledgments

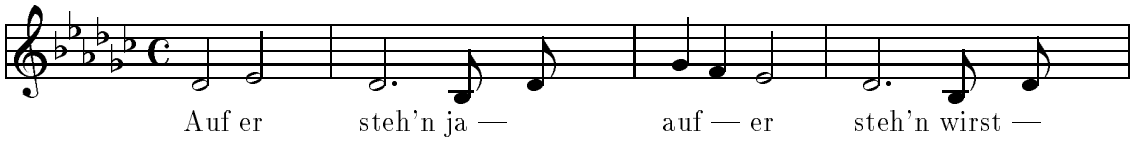
I would like to thank ...

- Lorraine Johnston and Bert Thompson for independently providing me with published material on abstract reduction based methods.
- The honours students for creating a great slaving environment and support system.
- MUCS[1] for helping me cope with stress.
- Malcolm Herbert for `choral-chat` — a much needed source of support and procrastination.
- Alistair Moffat for his sense of humour and helpfulness.
- Bert Thompson and Tim Gabric ("The MHS gang of three") for many interesting discussions over pizza.
- My brothers for sometimes turning down their music.
- My sister for being cheerful.
- Tuli for providing a natural example of laziness and for purring a lot.
- Mahler for his second symphony — always a spirit lifter.
- Harald Søndergaard for support, help, proof reading and above all for being a great guy.
- My parents for patience, tolerance, support and for being wonderful people.

"That which does not kill you makes you stronger"

— An anonymous honours student describing honours

Langsam. Misterioso



[1] Melbourne University Choral Society

Contents

0 Introduction

Strictness analysis is a compile time analysis applicable to *lazy functional* programming languages. Strictness analysis is important both to efficient sequential implementations and to simpler parallel implementations.

Strictness analysis has seen much work within the functional programming research community beginning with Mycroft's 1981 thesis [Myc81] where strictness analysis was first proposed and was done using *abstract interpretation*. The language that could be handled by Mycroft's method was fairly simplistic. Further work in strictness analysis focused on extending abstract interpretation based strictness analysis to more realistic languages.

As work progressed it became apparent that abstract interpretation based methods were rather slow. This saw work on so called *frontiers* — clever representations for the abstract functions used.

As abstract interpretation failed to yield a *practical* analysis some researchers turned to investigating other bases for strictness analysis. The first alternative we are aware of was a strictness analysis method based on *type inference* techniques [KM87].

This thesis gently introduces lazy functional programming languages and strictness analysis. After surveying existing analysis methods we go on to present a new method we have developed. This new method is faster than existing analysis proposals and, we feel, simpler than type inference based methods. The new method is closely related to abstract interpretation and inherits many of its nicer properties including easy provability while avoiding its inefficiency.

Section 1 explains functional programming and laziness, arguing why they are desirable. Section 2 introduces strictness analysis and explains what it buys. In Section 3 we explain the major proposed methods for strictness analysis in some detail. We then go on in section 4 to present our new method and give examples. Our implementation of the new method is described in section 5. We then discuss optimisations to our method in section 6. Section 7 describes possible further work. Section 8 concludes.

We shall assume that the reader is familiar with elementary lattice theory. A brief summary of the notions and terms we use can be found in appendix A. We shall also be assuming familiarity with the λ-calculus and with representation of functions within the lambda calculus — particularly with the $\mathcal{Y}$ combinator and its use.

1 Lazy Functional Programming

1.1 Functional Programming

1.1.1 What?

A *functional programming language* is a programming language that is based on the notion of functions, where computation consists of evaluating expressions and (user defined) functions.

An *impure* language is one that has side effects and/or assignment. A *pure* language is one that has neither side effects nor assignment.

Functional programming languages are members of the class of *declarative* programming languages — languages based on sound mathematical principles. This class includes logic programming and specification languages as well as functional languages. Examples of impure functional languages are LISP [WH81], Scheme and SML. Examples of pure functional languages are Miranda[2] [Hol91] and Haskell.

An excellent overview of functional programming is [Hud89].

1.1.2 Why?

Declarative languages have long been advocated by researchers as being superior to more conventional sequential imperative languages. Programs, it is claimed, are shorter and are easier to write, debug and maintain.

Why should this be so?

Part of the answer is that languages like Miranda and Haskell are both modern and well designed. They have a strong type system, pattern matching, list comprehensions and various other features. Furthermore they are symbolic languages, providing convenient idioms for the manipulation of lists, trees etc. All of which stand in sharp contrast to languages like `C`.

In addition having been designed by people of a theoretical background the languages tend to be carefully and solidly designed with due accord given to such principles as orthogonality and first class citizenship. As a result these languages tend to be simpler and more general than many other languages.

These advantages are due to the simple fact that declarative languages are well designed, a well designed imperative language would possess many of these advantages.

The rest of the answer however, is that the lack of side effects and the existence of pleasant mathematical properties buys further advantages. The single major language feature that is made possible by the outlawing of impure constructs (such as assignable variables and side effects) is *laziness*. We shall discuss this in the following section.

In addition, the simple semantics possessed by these languages significantly simplify the construction of various sophisticated tools such as program analysers, partial evaluators, program transformation tools and declarative debuggers.

Perhaps the most important advantage however is a software engineering one, in the absence of side effects coupling is reduced and it is significantly easier to combine code and to test code in isolation.

[2] Miranda is a Trademark of Limited Research Software

1.2 Laziness

1.2.1 What?

An evaluation strategy[3] for a functional programming language is *call by name* or *non strict* if subexpressions are only evaluated if they are needed. An evaluation strategy for a functional programming language is *lazy* or *call by need* if it is non strict and furthermore arguments to a function are evaluated at most once.

From the point of view of programming style it is important that arguments to both functions and data constructors (such as `cons`) not be needlessly evaluated.

For example under a non strict evaluation strategy we have that[4]

$$(\lambda\, x.\, \lambda\, y.y)\ \bot\ 3 \quad = \quad 3$$

$$head\ (cons\ 3\ \bot) \quad = \quad 3$$

1.2.2 Why?

Laziness is desirable despite the required sacrifice of side effects since it gives you a number of significant advantages. For a full discussion on why laziness and functional programming are good see [Hug90b]

Advantages conferred by lazy evaluation include

- Infinite data structures
- Glue
- The ability to model deterministic communicating processes
- Efficient argument passing
- Substitutivity properties
- A good fit with denotational semantics

Infinite Data Structures

This is is perhaps the most often heard argument. The ability to define infinite data structures is an additional piece of expressiveness.

It is useful in that it allows the separation of the termination condition from the generation of values — we can define an infinite list and its termination condition separately. This gives the functionality of generators without requiring any additional language constructs.

Glue

Laziness is extremely useful in allowing us to combine sections of programs without having to worry about potential inefficiency due to unneeded evaluation. A good counter example is coroutining logic programming languages where taking the conjunction of two perfectly well defined predicates can lead to nontermination.

[3] The rule used to decide in which order redexes (reducible subexpressions) are reduced

[4] We use $\bot$ to represent the undefined value

Communicating Processes

Lazy lists can be used to model streams.

We can then write functions over streams that are mutually recursive. The results derived by lazy evaluation are faithful to the communicating process interpretation.

The canonical example is the hamming sequence. While the solution in a lazy functional programming language is extremely short and simple the problem is actually quite hard to solve in a conventional imperative language [Dij79].

$$\begin{aligned} hamming \quad = \quad & 1 : (merge3 \quad (map\ (*2)\ hamming) \\ & \qquad\qquad\ (map\ (*3)\ hamming) \\ & \qquad\qquad\ (map\ (*5)\ hamming)) \end{aligned}$$

A special case of this paradigm are the standard Unix notion of pipes.

Efficient Argument Passing

If an argument to a function may not be needed and is costly to evaluate we would like to avoid evaluating it until it turns out to be required. Doing this in a strict language involves modifying the caller to pass a "suspension" and modifying the callee to "force" the suspension.

Another example of this are `let` clauses where in a strict language care must be taken so variables are not defined before they are certain to be needed.

A typical solution is to convert a value declaration of the form

```
val x = <long computation>
```

to one of the form

```
fun x () = <long computation>
```

and replace references to `x` with the function application `(x ())`. This type of modification breaches modularity and is unpleasant and unnecessary under lazy evaluation.

Substitutivity Properties

It turns out that simple substitutivity properties fail in the absence of laziness. A simple (contrived) example is

$$\text{let } x \ = \ \bot \text{ in } 3$$

This should ideally be equivalent to the result of substituting for x in the body of the let clause. However under an eager evaluation strategy the former will fail to terminate whereas the substituted form is just 3.

Denotational Semantics

Lazy languages are very suitable for the implementation of denotational semantics of languages. Typically the denotational equations can be transcribed directly and run.

1.2.3 Why Not?

There are two arguments against the use of lazy evaluation.

1. We must sacrifice side effecting constructs and variable assignment to be able to have laziness
2. Efficiency

Side Effects and Variables

Firstly a word of explanation on why side effects and laziness are not compatible. Consider a lazy language with a `print` statement that has the side effect of printing something to the screen. In the expression `(f (print "Hello") (print "World"))` we need to know whether `f` actually requires its arguments in order to know whether one, both or neither of the strings get printed. Worse, in the case that `f` evaluates both arguments we need to know which argument gets evaluated first to be able to predict whether we get the message `HelloWorld` or `WorldHello`.

A similar example can be constructed for variable assignment. That mixing assignments and expressions is a bad idea can be seen by looking at the semantics of `C` code such as

```
i = 3;
i = (i++) + (++i);
```

Since the `C` language definition does not clearly specify an evaluation order the semantics of abominations like this are not clearly defined[5].

It is a tribute to the expressiveness of lazy functional languages that they can easily regain the lost expressiveness by simulating a global state. The technique is called the *monadic* style and is beyond the scope of this thesis. Interested readers may see [Wad92]

Efficiency

The single main argument against laziness is the devil of efficiency. Laziness is wonderful, however it is no free lunch — lazy implementations must construct suspensions and force them resulting in a significant run time overhead. Worse still, even simple programs that do not make essential use of laziness exact this runtime penalty.

This problem can be solved.

The solution is strictness analysis.

The idea is to analyse the program and detect which functions do and don't make use of laziness. The non-lazy functions can be compiled using that knowledge thus avoiding most of the overheads.

[5] This example actually gives different results on different compilers

2 A Brief Introduction to Strictness Analysis

In this section we briefly overview strictness analysis and abstract interpretation before plunging into the details in the next section. Introductions to strictness analysis can also be found in [CP85, Hug90a] and in chapter 22 of [Pey87]

2.1 What is Strictness Analysis?

As mentioned previously we are interested in finding out when we can "safely"[6] evaluate the arguments to a function call before evaluating the function's body.

There are two feasible formalisations:

1. A function is strict in an argument if it forces the evaluation of that argument in the process of computing a result (intuitively if it "needs" the argument)
2. A function is strict in an argument iff
   (argument undefined) $\Rightarrow$ (function result undefined)

These may appear at first glance to be identical. There is in fact a subtle difference between them that surfaces in the case of non terminating functions. The function

$$f\ x\ =\ \bot$$

is considered strict in x under the second formalisation but not under the first.

Since in this case the function will fail to terminate under lazy evaluation we clearly cannot *introduce* nontermination so it is safe to evaluate x strictly. We use the second definition of strictness.

It should be noted that strictness does not attempt to detect the case when an argument is not used. We are interested only in the two cases

- The argument is definitely needed
- The argument is not definitely needed

Having defined strictness, strictness analysis is, as one might expect, a (compile time) analysis that detects strictness.

2.2 What Does it Buy You?

- Sequential implementations — avoid creating and later forcing closures [LGY87, SNvGP91]
- Parallel implementations — detect non-speculative parallelism [KPRS92]
- Improved space usage

[6] That is without introducing nontermination

Sequential Implementations

As noted previously the implementation of laziness has overheads due to the associated manipulation of suspensions. In the case that the suspension is actually evaluated then knowing in advance that it will be evaluated enables us to avoid the unnecessary work of creating and then forcing the suspension.

Another overhead is repeated forcing — if an argument is not known to be evaluated in advance then any reference to it has to check whether it has been evaluated *even after it has been evaluated.*

In the case that an argument is strict we know that it has been evaluated and any reference to it in the body of the function can avoid the "has it already been evaluated?" test.

Parallel Implementations

The parallel implementation of functional languages is a whole research area in its own right. In brief, due to the absence of side effects it is possible in a function call of the form $(f\ e_1\ e_2)$ to evaluate e_1 and e_2 in parallel. Things aren't however that simple. It is possible that f doesn't actually require e_2. In this case we would be wasting work in evaluating e_2 and worse, possibly be introducing non-termination.

One can solve the first problem by assigning a lower priority to "speculative" evaluation. One would begin the parallel evaluation with a "speculative" priority. We need however to be able to

1. Raise the priority of a subexpression when and if we discover it is needed (Eg. e_1)
2. Kill processes which we have discovered to be unnecessary (essentially garbage collect processes)

The complexities and overheads of managing speculative evaluation are significant.

Strictness analysis offers a solution in that if we know that f is strict in its first argument we can safely begin the parallel evaluation of f's first argument.

Strictness analysis allows us to detect non-speculative parallelism, thus enabling parallel implementations of lazy functional languages to avoid the complexities of speculative evaluation.

Better Space Usage

There is one further overhead associated with creating suspensions and that is space. In some cases a suspension can contain a pointer to a large graph representing the unevaluated subexpression. This can defeat tail recursion optimisation and make an algorithm run in non-constant space.

For example, if we have the tail recursive function definition

```
sum  []    n  = n
sum (m:ms) n  = sum ms (m+n)
```

Then in the absence of strictness information we will have a large graph being created in suspensions consisting of many '+' nodes and the elements of the list. Only after the entire list is traversed will evaluation of the subgraph be triggered and its nodes become available for reuse.

If it is known that `sum` is strict in its second argument then we can evaluate the additions before the recursive call to `sum` thus avoiding this problem.

2.3 Examples

1. $f\ x\ y\ =$ if $x = 0$ then 0 else y
   This is strict in x but not in y. Intuitively x is needed in the conditional whereas y may not be required.

2. $f\ x\ y\ =$ if $x = 0$ then y else $y + 1$
   In addition to being obviously strict in x this is strict in y since y is needed regardless of which arm of the conditional is selected.

3. $add\ x\ y\ =$ if $x = 0$ then y else $add\ (x - 1)\ (y + 1)$
   The function *add* is obviously strict in x. What is not so obvious is that it is strict in y. The 'then' case obviously needs y. What about the 'else' case?
   This is an example of recursion. To solve this we need to compute the *least fixpoint* of *add* using the ascending Kleene sequence.

2.4 Complications for Strictness Analysis

As may be apparent, strictness analysis is conceptually simple. There are however, a number of factors that complicate the analysis.

- Recursion
- Higher Orderness
- Non Flat Domains
- Polymorphism

Recursion

Recursion is the central problem. To analyse recursive functions we must compute an ascending series of approximations to obtain the result. This is the major reason why strictness analysis using abstract interpretation is inefficient.

Higher Orderness

Functional languages are higher order – that is, they allow functions to take functions as arguments and to return functions. This forces us to expand our domains to include (abstract) functions.

Non Flat Domains

Naive analyses assume that data is either defined or undefined. This is the case only for atomic data types such as integers, characters, floats etc. When dealing with composite data structures (such as lists, trees and tuples) there are many degrees of definedness. If we ignore these we get very poor analysis results.

Polymorphism

In the context of functional languages polymorphism (almost always) refers to parametric polymorphism. This is when a function is (partially) independent of its argument's type and can be applied to a variety of types. For instance the length function can be applied to lists of anything. An example of a *monomorphically* typed language is Pascal.

A number of proposed methods for strictness analysis only handle monomorphically typed languages. While handling polymorphism is not difficult it is a point to be aware of when designing strictness analysis methods.

Furthermore, the "obvious" or naive way of handling polymorphism is to define it in terms of a number of monomorphic instances. This is hideously inefficient[7] [Sew93].

2.5 Abstract Interpretation

There are a number of analysis methods that have been suggested. Since many of them are outgrowths of abstract interpretation we feel it prudent to briefly outline the concepts behind abstract interpretation before continuing.

The essential idea behind abstract interpretation is quite simple — rather then carrying out a computation in all its gory detail we can simulate it by eliding details thus making it tractable.

The canonical example is the rule of signs — rather then multiplying together two large numbers we can abstract them to their signs and "multiply" the signs. Another example is the casting out of nines.

Salient points to note are

- We are replacing the real (often complex and intractable) domain with a simpler but less informative domain. Quite often the original domain will be infinite and the abstract domain finite.
- For each operation on the real domain we have a corresponding operation on the abstract domain. We demand that these operations be safe in some sense.
- The results given by the abstract interpretation may be inaccurate but are guaranteed to be safe. An abstract interpretation may return a "don't know" result, however if the abstract interpretation returns that some property holds then we are guaranteed that indeed, this is the case.

2.6 Research Areas in Strictness Analysis

There are three main research areas within strictness analysis.

1. Extending abstract interpretation based analysis to "real"[8] languages
2. Algorithms for fixpoint computation
3. Exploration into methods of strictness analysis not based on abstract interpretation

[7] As in close to three orders of magnitude
[8] Non flat, higher order, polymorphic

Extending Abstract Interpretation to "Real" Languages

As noted before, certain language features such as higher-orderness, non-flat data types and polymorphism are a challenge in terms of obtaining an efficient yet accurate analysis.

Much work has been done on extending basic strictness analysis as originally proposed by Mycroft [Myc81] (first order, flat, monomorphic) to real languages. Researchers have addressed both the handling of higher order functions [HY86, BHA85, BHA86] and the handling of non-flat data [Hug85, Wad87, Hug87]. Good handling of non-flat languages has proven to be interesting and has spawned a number of interesting methods including backwards abstract interpretation & evaluator transformers [Bur87, Bur91] and projections [WH87, Hug89].

These issues are discussed in the next section.

Fixpoint Algorithms

Abstract interpretation involves a fixpoint iteration over an abstract function. This involves an equality comparison between functions. Hence we need to be able to represent and compare functions.

Unfortunately functions representations have a worst case exponential complexity. An example is truth tables which are exponential and grow too large even for modest numbers of arguments. It should be noted that truth tables represent functions over a two point domain — most strictness analysers use much larger domains.

Work has been done on finding representations that in typical cases are compact and that allow equality tests and the computations associated with fixpoint iteration to be done rapidly [MH87, PC87, Hun89].

Another research thrust has been work on heuristics for speeding up analysis.

We will not be covering this research area.

Alternative Approaches

Abstract Interpretation based methods are inefficient. Some researchers feel that the best way to get an efficient strictness analysis algorithm is to avoid abstract interpretation. Alternative approaches to strictness analysis are *type inference* based methods[KM87, LM91, Wri91, Jen91] and more recently a method based on *abstract reduction* [Nöc92, Nöc93a, Nöc93b].

The emergence of alternative methods has seen some work comparing the power of various methods [DW90, HL90, Jen90, NM92]. This work is of a theoretical nature and comprise a comparison of the power inherent to the different formalisms. Much of it is inconclusive and is the subject of current investigation. In our comparisons we shall prefer to be informal and focus on the differences in the *practical algorithms* rather than the differences between the underlying formalisms.

3 Survey of Analysis Methods

3.1 A Brief History

The first formulation of strictness analysis was in terms of abstract interpretation [Myc81]. As a result abstract interpretation based methods have seen the most work. The mid eighties saw work on extending Mycroft's work to higher-order and non-flat languages. Type inference based methods were proposed as an alternative to abstract interpretation in 1987.

3.2 Abstract Interpretation Based Methods

3.2.1 Examples

Consider a function that we wish to analyse for strictness say[9] ...

$$add\ (x, y)\ =\ \text{if } x = 0 \text{ then } y \text{ else } add\ (x - 1, y + 1)$$

We use the abstract domain $(\bot\ ,\ \top)$ and use it as a basis for constructing an abstract function. The ordering on the abstract domain is $\bot\ \leq \top$. The intuitive interpretation of the domain is that the $\bot$ represents a result which is undefined and $\top$ represents a possibly defined value.

To avoid confusion it is worth noting that the intuitive interpretation of the domain in abstract interpretation is the reverse of the interpretation used in semantics. In abstract interpretation the accuracy increases as we move down the lattice – $\bot$ represents the most accurate information and $\top$ represents a complete absence of information. In semantics $\bot$ represents "undefined" and moving up the lattice adds definition (the domains used in giving semantics to functional languages typically do not have a top element).

We will follow Mycroft's convention and use a superscript hash to denote abstraction. Thus $add^{\#}$ is the abstract function corresponding to add.

Our first step in defining an abstraction of add is

$$add^{\#}\ (x, y)\ =\ \text{if } x =^{\#} 0^{\#} \text{ then } y \text{ else } add^{\#}(x -^{\#} 1^{\#}, y +^{\#} 1^{\#}))$$

The abstraction of a constant is $\top$ since a constant is not undefined. Upon examination the abstract functions corresponding to the equality test and to subtraction are both $\wedge$ defined by

$$x \wedge y \stackrel{\text{def}}{=} \text{ if } (x = \top) \text{ and } (y = \top) \text{ then } \top \text{ else } \bot$$

We shall also find the lattice operation $\vee$ useful, its definition is

$$x \vee y \stackrel{\text{def}}{=} \text{ if } (x = \top) \text{ or } (y = \top) \text{ then } \top \text{ else } \bot$$

The abstract function for if-then-else is slightly more complex

$$(\text{if } x \text{ then } y \text{ else } z)^{\#} \stackrel{\text{def}}{=} x\ \wedge\ (y\ \vee\ z)$$

We can then rewrite our abstract function as

$$add^{\#}\ (x, y)\ =\ (x \wedge 1) \wedge (y \vee (add^{\#}\ (x \wedge 1, y \wedge 1)))$$

Since $\top \wedge x = x$ we can simplify to obtain

$$add^{\#}\ (x, y) = x \wedge (y \vee (add^{\#}\ (x, y)))$$

[9] We use a tuple of arguments since we wish to avoid currying and higher orderness at the moment

We now have an abstract function, that is, a function on the abstract domain that in some sense corresponds to the original function. This "some sense" is the notion of *safe approximation* – we can use the abstract function to determine the strictness of the original function. The information obtained is not guaranteed to be accurate however it is guaranteed to err on the safe side. A contrived example where we get inaccurate information is

$$f\ x\ =\ \text{if } 0 = 1 \text{ then } 3 \text{ else } x$$

This will in all cases require that x be defined however an analysis will return $\top$ representing "I don't know".

The question of whether *add* is strict in x reduces to whether $add^{\#}\ (\bot, \top) = \bot$. Substituting for x and y in the body of *add* we obtain $(\bot\ \wedge\ (\top\ \vee\ add^{\#}\ (\bot, \top)))$ which is just $\bot$ – hence *add* is strict in x.

What about y? We now are interested in whether $add^{\#}\ (\top, \bot)$ is $\bot$. Substituting and simplifying yields $add^{\#}\ (\top, \bot)$ as the reader may verify. In order to solve this we take the *least fixed point* of *add*.

$$\begin{array}{lcl} add_0^{\#}\ (\top, \bot) & = & \bot \\ add_1^{\#}\ (\top, \bot) & = & add_0^{\#}\ (\top, \bot) \\ & = & \bot \end{array}$$

This yields the solution that *add* is indeed strict in y.

3.2.2 Handling Higher-Order and Non-Flat Programs

The method considered so far is limited to flat, first order languages. We now look at removing these restrictions.

Handling a higher order language is conceptually quite simple — we extend the domain to include functions as well as first order data items. Unfortunately this makes for a rather large domain making the fixpoint iteration expensive.

Extending to non-flat domains is a little more complex. Although we can easily extend the domain to handle degrees of definedness it turns out that getting useful information involves knowing what questions to ask.

The rest of this subsection is based on [Bur87, Bur91].

Consider a domain $0 < 1 < 2 < 3$ where the correspondence between the abstract and concrete domains is

0 $\bot$

1 Infinite lists, $\bot$ itself and lists with a tail of $\bot$

2 All lists containing $\bot$

3 All lists

Let us step back for a moment and consider the meaning of the strictness test $(f\ 3\ 0)\ =\ 0$. For reasons which shall soon become apparent we shall write the test in the equivalent form $(f\ 3\ 0)\ \leq\ 0$.

Let us say that we know that $(f\ 3\ 0)\ \leq\ 0$ and that we are in a context where we are demanding that the value of $(f\ 3\ x)$ is not completely undefined, that is $(f\ 3\ x)\ >\ 0$. We can then make use of monotonicity and reason as follows:

Known Property:	$f\ 3\ 0\ \leq\ 0$
Desired Result:	$f\ 3\ x\ >\ 0$
Required Property:	$x > 0$

The correspondence between a conclusion of the form $x > n$ for some n and the evaluation strategy is that if have a required property that $x > n$ for some n then we can safely evaluate x *as long as we can guarantee that if x is indeed greater then n then the evaluation will terminate.*

For example, in the above example we can evaluate x using any strategy that is guaranteed to terminate for $x \in \{1, 2, 3\}$.

The point is that using the same strictness test as before — phrased in terms of $\top$ (3) and $\bot$ (0) we can only say that for f's result to be defined x must be greater then 0.

Translating this to safe evaluation, all we can do safely with x before the call to f is evaluate it to *head normal form.* For lists however we would like to be able to evaluate them fully, to normal form. The strictness question asked until now however, only checks for evaluation to head normal form. The extra domain elements then, haven't really bought us anything.

The solution to this problem is to consider various degrees of demand on both function results and arguments. Rather than just considering strictness of the form

$$f\ 3\ 0\ =\ 0$$

we look at the more general case

$$f\ 3\ v\ \leq\ r$$

The interpretation that can be drawn from this question is that given that we demand that $(f\ 3\ x)\ >\ r$ and given we know that $(f\ 3\ v)\ \leq\ r$ we can safely evaluate x using a strategy that is guaranteed to terminate *if* $x > v$.

An Example

Our example is the append function defined as

$$\begin{array}{lcl} ap\ []\ ys & = & ys \\ ap\ (x : xs)\ ys & = & x\ :\ (ap\ xs\ ys) \end{array}$$

The abstract function corresponding to append is

$ys \backslash xs$	0	1	2	3
0	0	1	1	1
1	0	1	1	1
2	0	1	2	2
3	0	1	2	3

Given the table we can easily see that the following are valid

- $ap\ 3\ 2\ \leq\ 2$
- $ap\ 2\ 3\ \leq\ 2$

Thus if $(ap\ x\ y)$ is required to be completely defined (greater than 2) then we can evaluate x *and* y using a strategy that is guaranteed to terminate if they are greater then 2. Since being greater then 2 characterises completely defined lists we can safely evaluate to normal form.

3.2.3 Contexts, Evaluator Transformers and Backward Analysis

A *context* is simply a demand. We might demand that $(f\ x) > 2$ for instance.

An *evaluator* is an evaluation strategy that corresponds to a domain element in that evaluation is guaranteed to terminate if the value it is evaluating is larger or equal to the domain element. Thus the evaluator associated with 1 would evaluate to head normal form, the evaluator associated with 3 would evaluate to normal form, the evaluator associated with 2 would evaluate the spine of a list and the evaluator associated with 0 would do no evaluation.

The idea behind *backward analysis* is that rather than thinking in terms of functions which take arguments and return results we can look at making demands on functions' results and seeing what the resulting demands on the arguments are. We then think in terms of *evaluator transformers* — functions that take a demand and return the demand on their argument[10].

3.2.4 Advantages and Drawbacks of Abstract Interpretation Based Methods

+ Abstract interpretation based methods have a solid theoretical background.
+ Abstract interpretation is conceptually simple
+ Correctness proofs are easy. Since the groundwork has been done all that a proof involves is showing the safety of individual operations.
– Handling higher order functions and non flat domains involves significant complications. For instance Wadler and Hughes projections (which are a formalisation of evaluators and demands) are significantly less simple then simple forward analysis.
– Abstract interpretation based methods are inefficient.

3.3 Type Inference

This class of methods is based on the insight, first reported in [KM87], that one can view strictness properties as a form of type. We can then find strictness information using a type inference algorithm. Other work on this approach is [LM91, Jen91, Wri91] This class of methods is interesting for a number of reasons.

1. In abstract interpretation we distinguish between backward and forward analysis depending on the direction of the flow of data. Type inference is a multi directional process — data flows in both directions due to the use of constraints.
2. There is no need for fixpoint iteration.

Since type inference is not as widely understood as abstract interpretation we give an introduction to general type inference which is presented so as to be suitable as background for the type inference papers cited. Our discussion of *strictness analysis* type inference is brief and we refer the reader to the cited papers for the details.

[10] Rather then have these functions return multiple results (one for each argument) we abstract each function of arity n to n functions

3.3.1 How Does Type Inference Work?

The rules for type inference are in figure 1. Before continuing it should be noted that our presentation is non-standard since we would like to keep things general so we can move from inferring types to inferring strictness properties with minimal pain.

The first three rules suffice to check types. Given a conclusion of the form

$$\vdash \lambda f.\lambda x.(f\ (f\ x)) : (\alpha \to \alpha) \to \alpha \to \alpha$$

we can easily work backwards and end up with a proof tree (such as the one in figure 3).

Indeed this type of tree is often presented as an example of the inference rules at work. One thing which tends to baffle people first seeing this is that it appears as if we start off knowing the types since the leaves of the tree are typically of the form

$$f : \alpha \to \alpha,\ x : \alpha \vdash x : \alpha$$

The explanation is that when these inference rules are implemented as an algorithm what actually happens is that we start off by assuming that a variable introduced by a λ has type α (a fresh type variable). Subsequent use of the coerce rule will impose constraints upon the type of the variable.

Intuitively the coerce rule says that if I have that M is of type α and I need it to have type β then I can fudge that *provided* that I impose the constraint $\alpha \leq \beta$.

Consider now how we would go about implementing a type inference *algorithm*. We start off with an expression and work our way *backwards* through the rules. The structure of the expression determines which of the first three rules is applicable. This would make the selection of the rule to apply at each step nice and deterministic except that the coerce rule can be applied at any time.

It turns out that the coerce rule is only ever needed before an application so we can build its use into the application rule. Having done this the selection of the inference rule to apply is determined entirely by the top level structure of the expression being type checked.

The algorithm is in figure 2. It traverses the expression and returns a type and a set of constraints. Once this is done we need to check that the constraints C are actually consistent. The full algorithm then, consists of the steps:

1. Use the algorithm in figure 2 to give a type and a set of constraints.

$$(C, \alpha) = \mathit{type}\ \{\}\ \mathit{expression}$$

2. Check the set of constraints for consistency.

Recursion is handled by translating function definitions to a non-recursive form using the $\mathcal{Y}$ combinator and treating the $\mathcal{Y}$ combinator as a constant having type

$$\mathcal{Y} :: (\alpha \to \alpha) \to \alpha$$

The nice thing about this algorithm is that it is independent of the domain — by changing the definition of $\leq$ and the algorithm used to check the consistency of a constraint set we can get a classical type inference algorithm or a strictness analyser using type inference.

Figure 3 gives an example of a type inference. The lambda expression being type checked is the "twice" function.

Figure 4 shows an example of the type inference *algorithm* in action on the same expression.

$$, , x : \alpha \ \vdash \ x : \alpha \ \text{(var)}$$

$$\frac{\Gamma \vdash M{:}\alpha \rightarrow \beta \quad \Gamma \vdash N{:}\alpha}{\Gamma \vdash (M\ N){:}\beta} \ \text{(ap)}$$

$$\frac{\Gamma[x/\alpha] \vdash E{:}\beta}{\Gamma \vdash \lambda x.E{:}\alpha \rightarrow \beta} \ \text{(lam)}$$

$$\frac{\Gamma \vdash M{:}\alpha \quad \alpha \leq \beta}{\Gamma \vdash M{:}\beta} \ \text{(coerce)}$$

Figure 1: Type Inference Rules

$type$	::	$env \rightarrow exp \rightarrow (constraintSet, type)$
$type\ ,\ x$	=	$(\{\}, lookup\ ,\ x)$
$type\ ,\ (e\ e')$	=	$(\{\beta \leq (\beta' \rightarrow \alpha)\} \cup C \cup C', \alpha)$
	where	
		α is a new type variable
		$(C, \beta)\ =\ type\ ,\ e$
		$(C', \beta')\ =\ type\ ,'\ e'$
$type\ ,\ (\lambda x.e)$	=	$(C, \alpha \rightarrow \beta)$
	where	
		α is a new type variable
		$(C, \beta)\ =\ type\ (,\ \cup (x : \alpha))\ e$

Figure 2: Type Inference Algorithm

The resulting set of constraints is consistent. When solved over the classical domain the standard type for twice is derived.

We would like to encourage readers to actually read through figure 4 closely or, better still, derive it themselves. We have found that the only way to understand type inference is to actually run through an example or two.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \alpha \to \alpha \quad \dfrac{\Gamma \vdash f : \alpha \to \alpha \quad \Gamma \vdash x : \alpha}{\Gamma \vdash f\ x : \alpha}}{\Gamma \vdash f\ (f\ x) : \alpha}}{f : \alpha \to \alpha \vdash \lambda x.f\ (f\ x) : \alpha \to \alpha}}{\vdash \lambda f.\lambda x.f\ (f\ x) : (\alpha \to \alpha) \to \alpha \to \alpha}$$

Where $\Gamma = f : \alpha \to \alpha\ ,\ x : \alpha$

Figure 3: The Type Inference Rules In Action

$$
\begin{array}{lcl}
\text{type } \{\}\ (\lambda f.\lambda x.f\ (f\ x)) & = & (C, \alpha_0 \to \beta_0) \\
& \text{where} & \\
& & (C, \beta_0)\ =\ \text{type } \{f : \alpha_0\}\ (\lambda x.f\ (f\ x)) \\
 & & \\
\text{type } \{f : \alpha_0\}\ (\lambda x.f\ (f\ x)) & = & (C, \alpha_1 \to \beta_1) \\
& \text{where} & \\
& & (C, \beta_1)\ =\ \text{type },\ (f\ (f\ x)) \\
 & & \\
\text{type },\ (f\ (f\ x)) & = & (\{\beta_2 \le (\beta_2' \to \alpha_2\} \cup C_2 \cup C_2', \alpha_2) \\
& \text{where} & \\
& & (C_2, \beta_2)\ =\ \text{type },\ f\ =\ (\{\}, \alpha_0) \\
& & (C_2', \beta_2')\ =\ \text{type },\ (f\ x) \\
& = & (\{\alpha_0 \le (\beta_2' \to \alpha_2)\} \cup C_2', \alpha_2) \\
& \text{where} & \\
& & (C_2', \beta_2')\ =\ \text{type },\ (f\ x) \\
 & & \\
\text{type },\ (f\ x) & = & (\{\beta_3 \le (\beta_3' \to \alpha_3\} \cup C_3 \cup C_3', \alpha_3) \\
& \text{where} & \\
& & (C_3, \beta_3)\ =\ \text{type },\ f\ =\ (\{\}, \alpha_0) \\
& & (C_3', \beta_3')\ =\ \text{type },\ x\ =\ (\{\}, \alpha_1) \\
& = & (\{\alpha_0 \le (\alpha_1 \to \alpha_3)\}, \alpha_3) \\
 & & \\
\text{type },\ (f\ (f\ x)) & = & (\{\alpha_0 \le (\beta_2' \to \alpha_2\} \cup C_2', \alpha_2) \\
& \text{where} & \\
& & (C_2', \beta_2')\ =\ \text{type },\ (f\ x)\ =\ (\{\alpha_0 \le (\alpha_1 \to \alpha_3)\}, \alpha_3) \\
& = & (\{\alpha_0 \le (\alpha_3 \to \alpha_2)\ ,\ \alpha_0 \le (\alpha_1 \to \alpha_3)\}, \alpha_2) \\
 & & \\
\text{type } \{\}\ (\lambda f.\lambda x.(f\ (f\ x))) & = & (\{\alpha_0 \le (\alpha_3 \to \alpha_2)\ ,\ \alpha_0 \le (\alpha_1 \to \alpha_3)\}, \alpha_0 \to \alpha_1 \to \alpha_2) \\
\text{Where }, \ = \{f : \alpha_0,\ x : \alpha_1\} & &
\end{array}
$$

Figure 4: The Type Inference Algorithm In Action

3.3.2 A Classical Type Inferencer

We read $x \leq y$ as "x is an *instance* of y". The constraints can be checked for consistency using a unification based algorithm. The $\leq$ can be seen as dictating a lattice where the top element is X[11] and where the rest of the domain looks like ...

$$\begin{array}{lcl} int & \leq & X \\ X \to X & \leq & X \\ X \to int & \leq & X \to X \\ X \to int & \leq & X \to X \\ int \to X & \leq & X \to X \\ int \to int & \leq & X \to int \\ int \to int & \leq & int \to X \\ (X \to X) \to X & \leq & X \\ X \to (X \to X) & \leq & X \\ & \vdots & \end{array}$$

This domain ordering is the reverse of the usual. This is due to a desire to be compatible with the strictness analysis algorithm given in [LM91]. Recall from section 3 that the interpretation of $\top$ and $\bot$ in abstract interpretation is the reverse of the usual. This property is inherited in the formulation of strictness analysis in [LM91].

3.3.3 A Strictness Analyser Based on Type Inference

To get a type inference based strictness analyser we use the same algorithm but with a different domain of types. There are actually a number of domains we could use. In addition to designing a language of type expressions we need to provide an ordering on them (together these form the domain) and an algorithm for testing for consistency of a set of constraints.

Figure 5 shows an example of a type inference strictness analysis. The language of types used is

$$\alpha \ ::= \ \bot \mid \top \mid \Omega \mid \alpha_1 \to \alpha_2 \mid \alpha_1 \times \ldots \times \alpha_n$$

The intuition behind the types is that

- $\top$ represents any element
- $\bot$ represents an element with no *head normal form*
- Ω represents an element with no *normal* form

Thus $\bot \to \bot$ is the type of a function that maps an element with no head normal form to another such element – that is given an undefined argument returns an undefined result. This is a round about way of saying "strict".

It should be noted that a function has a number of types, just as in conventional type inference a polymorphic function has many instances. A function with type, say, $(\beta \to \alpha) \to \beta \to \alpha)$ where we have the constraint $\alpha \leq \beta$ could be instantiated to ...

- $(\top \to \bot) \to (\top \to \bot)$
  "the function is strict in it's first argument (which is a function)"

[11] Which should be thought of as a concrete domain element — *not* a variable

- $(\bot \to \bot) \to (\bot \to \bot)$
  "if the function's first argument is a strict function then the result of the (partial) application is a strict function"

Space does not permit us to reproduce the details — in particular the constraint checking algorithm. The interested reader should see [LM91, KM87, Jen91, Wri91] for the details. The example type language we use is taken from [LM91].

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \bot \to \bot \quad \dfrac{\Gamma \vdash f : \bot \to \bot \quad \Gamma \vdash x : \bot}{\Gamma \vdash f\ x : \bot}}{\Gamma \vdash f\ (f\ x) : \bot}}{f : \bot \to \bot \vdash \lambda x.f\ (f\ x) : \bot \to \bot}}{\vdash \lambda f.\lambda x.f\ (f\ x) : (\bot \to \bot) \to \bot \to \bot}}{\text{Where } \Gamma = f : \bot \to \bot\ ,\ x : \bot}$$

Figure 5: An Example of a Strictness Type Inference

3.3.4 Advantages and Drawbacks of Type Inference Based Methods

+ No need for fixpoint iteration
+ Handling higher-orderness comes "for free"
– Type inference is not as simple as abstract interpretation
– Constraint solving can be both complex and expensive
– Correctness proofs are harder than for abstract interpretation based methods

3.4 Abstract Reduction Based Strictness Analysis

A recent newcomer to the area is *abstract reduction* based strictness analysis [Nöc92, Nöc93a, Nöc93b]. Briefly, the idea is that the functions are executed over a domain containing graphs and union ($\cup$) operators. Expansion only takes place to a predetermined depth, and at that depth undetermined expressions are approximated by $\top$.

Unfortunately we only obtained access to published work on this method rather late, thus we have not had the time to fully understand this method. We feel that it is worth mentioning abstract reduction even if only to provide pointers to published work and to note its existence as a promising method for strictness analysis.

3.4.1 Advantages and Drawbacks of Abstract Reduction

It should be stressed that these are derived from a cursory overview of [Nöc92, Nöc93a, Nöc93b].

+ Fast
+ Handles non-flat data well
– Correctness proofs are hard

3.5 Comparison of the Different Methods

It is hard to compare abstract interpretation and type inference based methods. Type inference based methods have seen less work and are less mature. Specifically we have not been able to find performance measurements for type inference based methods. While abstract interpretation is simpler and has pleasant theoretical properties it is hard to deny that is it slow. An open question is whether type inference based methods are significantly faster for real programs.

Abstract reduction is an algorithm developed from scratch. This makes explanation harder in comparison to abstract interpretation and makes correctness proofs difficult. It is however (claimed to be) a fast practical method. By comparison, our new proposed method is based on abstract interpretation and is thus easy to both explain and prove correct. It is fast — faster than abstract reduction. Its accuracy is however currently low.

4 Yet Another Analysis Method

4.1 Motivation

The bottom line is that the methods discussed in the previous section are too slow. With the exception of methods based on type checking they require the use of fixpoint iteration over the abstract domain. This is workable for small domains. Unfortunately handling higher order functions and non flat domains requires extending the abstract domain.

The methods based on type checking don't require fixpoint iteration. We have not however been able to find any performance figures for this class of methods in the literature. Hindley-Milner type checking is known to be potentially exponential and is not a fast operation. Strictness analysis using type checking techniques involves a more complex unification procedure and thus are likely to be slower than conventional type checking.

The method we outline in this section is based on an insight that allows us to avoid the need for fixpoint iteration. The algorithm we derive is efficient.

4.2 The Insight

The reason abstract interpretation based methods are slow is that they require fixpoint iteration. The function being iterated over is an *arbitrary function over the abstract domain*

The key insight is that **given that the domain is higher order we can constrain the function being iterated over to be an element of the domain.**

This means that the fixpoint computation takes place entirely *within* the abstract domain. This allows us to simply precompute the fixpoint operator over the abstract domain.

The central equivalence to bear in mind is

$$f \;=\; \lambda x. \ldots f \ldots \quad\equiv\quad f \;=\; \mathcal{Y}\,(\lambda g.\, \lambda x. \ldots g \ldots)$$

Having transformed our functions to non-recursive ones involving the fixpoint operator $\mathcal{Y}$ we need to be able to take an arbitrary lambda expression and determine an abstract domain element corresponding to it. That is we need a function `abs` such that

$$\texttt{abs}\;(\lambda g.\, \lambda x. \ldots g \ldots) \;=\; \Box \in \mathrm{ADom}$$

This is nontrivial. The solution we have adopted makes use of *SK Combinators*.

A *combinator* is a λ-calculus expression which does not contain any occurrences of free variables. Remarkably, it turns out that any closed λ-calculus expression can be expressed[12] using only the two combinators S and K. *SK combinators* have been used as an implementation technique for lazy functional languages [Tur79].

In practice translating a λ-expression to SK combinators produces rather blown up code. Adding combinators can reduce the size of the resulting expression. Other commonly used combinators are I, B, S', $B*$ and C. In our presentation we shall only make use of S, K and I.

$S\ f\ g\ x$	$=$	$f\ x\ (g\ x)$
$K\ x\ y$	$=$	x
$I\ x$	$=$	x

[12] Pun intended

4.3 An Overview of the Method

What are the steps involved?

Given a function, say,

$$f\ x\ y\ =\ \text{if } x = 1 \text{ then } y \text{ else } f\ (x-1)\ (x * y)$$

we go through the following steps

1. Convert to a value declaration using explicit λs

$$f\ =\ \lambda\, x.\, \lambda\, y.\, \text{if } x = 1 \text{ then } y \text{ else } f\ (x-1)\ (x * y)$$

2. Remove recursion by using the $\mathcal{Y}$ combinator

$$f\ =\ \mathcal{Y}\ (\lambda\, g.\, \lambda\, x.\, \lambda\, y.\, \text{if } x = 1 \text{ then } y \text{ else } g\ (x-1)\ (x * y))$$

3. Abstract the function definition

$$f^{\#}\ =\ \mathcal{Y}\ (\lambda\, g.\, \lambda\, x.\, \lambda\, y.(x \wedge \top)\ \wedge\ (y \vee (g\ (x \wedge \top)\ (x \wedge y))))$$

4. Simplify the definition

$$f^{\#}\ =\ \mathcal{Y}\ (\lambda\, g.\, \lambda\, x.\, \lambda\, y.(x\ \wedge\ (y \vee (g\ x\ (x \wedge y)))))$$

5. Transform the expression to SK combinators. This leaves us with an expression involving only applications of a fixed set of constants

$$f^{\#}\ =\ \mathcal{Y}\ (((\mathcal{B}*\ ((\mathcal{S}'\ \mathcal{B})\ \wedge))\ (\mathcal{B}\ (\mathcal{S}\ \vee)))\ ((\mathcal{C}\ (\mathcal{S}'\ \mathcal{B}))\ \wedge))$$

6. Abstract each constant to an element of the abstract domain.

7. Use the rules associated with the abstract domain to reduce the expression on the right hand side to a single value in the abstract domain.

8. Interpret this value.

We have omitted the workings of the last few steps here since they depend on the choice of the abstract domain.

The domain we shall use as an illustration is a little simplistic. In particular it does not handle the full Turner set of combinators.

4.4 Designing the Domain

Requirements

For this method to work the domain must be higher order. More precisely we must be able to abstract the combinators used. For the analysis to yield good results we would like the domain to contain a variety of functions.

Note that the algorithm is quite fast and that quite large abstract domains are tractable.

Method

We begin by populating the domain with elements corresponding to constants and combinators. We then consider the results of applications of domain elements to each other.

In some cases the result will be an existing domain element.

In other cases the result of the application will not be an existing element of the domain. We have two choices we can make in this situation

- Define the result to be $\top$
- Add a new domain element to represent the result

When we invent a new domain element we need to consider the results of applications involving it.

4.4.1 The Domain Elements

Our basic domain elements are

$$\top \bot \wedge \vee \quad Y \; S$$

We use π_i^j to represent the function that takes j arguments and selects the ith argument ($j \geq i$). We also assume that the domain contains these selector functions π_i^j. This lets us abstract K and I.

$$K = \pi_1^2 \quad I = \pi_1^1$$

We then consider all possible applications.

It should be noted that the domain is infinite. This does not pose a problem since we don't need to iterate to a fixpoint. The selector functions behave in a fairly regular fashion and their behavior can be captured by a simple set of rules.

Note that $\bot$ is polymorphic in a sense in that it captures the set $\{\bot, \lambda\, x.\bot, \lambda\, x.\, \lambda\, y.\bot \ldots\}$ An analogous property holds for $\top$.

Name	*Rule(s)*						
S							
π_i^n	$\pi_{i+1}^{n+1}\; x$	$=$	π_i^n		$\pi_1^1\; x$	$=$	x
$\vee$	$\vee\; \bot$	$=$	π_1^1		$\vee\; \top$	$=$	$\top$
$\wedge$	$\wedge\; \bot$	$=$	$\bot$		$\wedge\; \top$	$=$	π_1^1
$\top$	$\top\; x$	$=$	$\top$				
$\bot$	$\bot\; x$	$=$	$\bot$				
Y	$(Y\; x$	$=$	$x\; (Y\; x))$				

Figure 6: The primitive domain elements

The observant reader will notice the symbol $\bullet$ being used as the result of some applications. This is equivalent to $\top$. It is used simply to allow us to distinguish between a result that is $\top$ because it should be $\top$ and a result that we have defined to be $\top$ since the real result isn't in the domain.

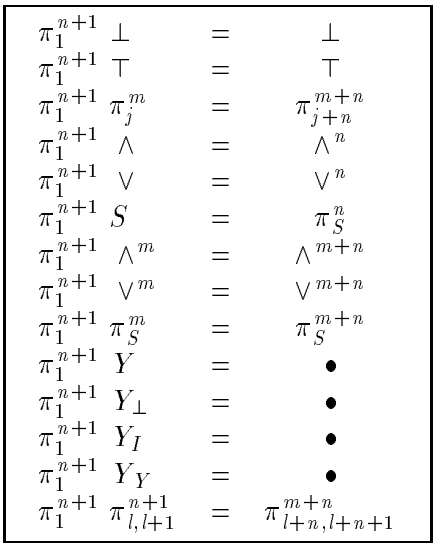
The rationale behind the naming of the domain elements Y_x is that $(Y\; Y_x) = x$.

Name		*Defined as*	*Rules*						
$\vee^n$	$\equiv$	$\pi_1^{n+1}\ \vee$	$\vee^{m+1}\ x$	$=$	$\vee^m$	,	$\vee^0$	$\equiv$	$\vee$
$\wedge^n$	$\equiv$	$\pi_1^{n+1}\ \wedge$	$\wedge^{m+1}\ x$	$=$	$\wedge^m$	,	$\wedge^0$	$\equiv$	$\wedge$
$Y_\perp$	$\equiv$	$S\ \wedge$							
Y_I	$\equiv$	$S\ \vee$							
Y_Y	$\equiv$	$S\ \pi_1^1$							
π_S^n	$\equiv$	$\pi_1^{n+1}\ S$	$\pi_S^{n+1} x$	$=$	π_S^n	,	π_S^0	$\equiv$	S
$\pi_{1,2}^n$	$\equiv$	$S\ \pi_2^n$	$(\mathrm{n} > 2)$						

Figure 7: The compound domain elements

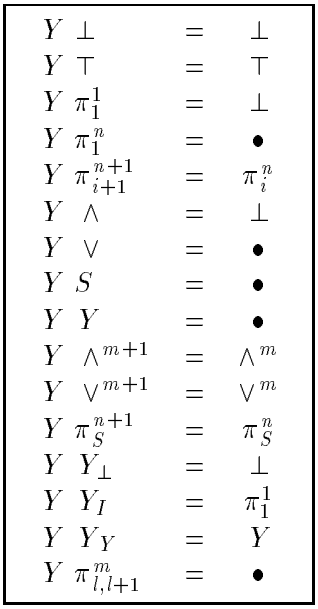
$S\ \perp$	$=$	$\perp$	
$S\ \top$	$=$	$\top$	
$S\ \pi_1^1$	$=$	Y_Y	
$S\ \pi_2^2$	$=$	π_1^1	
$S\ \pi_1^n$	$=$	π_2^n	$(\mathrm{n} > 2)$
$S\ \pi_j^n$	$=$	π_j^n	$(\mathrm{j} \geq 3)$
$S\ \pi_2^i$	$=$	$\pi_{1,2}^i$	$(\mathrm{j} \geq 3)$
$S\ \wedge$	$=$	$Y_\perp$	
$S\ \vee$	$=$	Y_I	
$S\ S$	$=$	$\bullet$	
$S\ Y$	$=$	$\bullet$	
$S\ \wedge^m$	$=$	$\wedge^m$	$(\mathrm{m} \geq 2)$
$S\ \vee^m$	$=$	$\vee^m$	$(\mathrm{m} \geq 2)$
$S\ \pi_S^n$	$=$	π_S^n	$(\mathrm{n} \geq 2)$
$S\ Y_\perp$	$=$	$Y_\perp$	
$S\ Y_I$	$=$	$\bullet$	
$S\ Y_Y$	$=$	$\bullet$	
$S\ \pi_{l,l+1}^m$	$=$	$\bullet$	

Figure 8: The reduction rules for S



π_1^{n+1}	$\bot$	$=$	$\bot$
π_1^{n+1}	$\top$	$=$	$\top$
π_1^{n+1}	π_j^m	$=$	π_{j+n}^{m+n}
π_1^{n+1}	$\wedge$	$=$	$\wedge^n$
π_1^{n+1}	$\vee$	$=$	$\vee^n$
π_1^{n+1}	S	$=$	π_S^n
π_1^{n+1}	$\wedge^m$	$=$	$\wedge^{m+n}$
π_1^{n+1}	$\vee^m$	$=$	$\vee^{m+n}$
π_1^{n+1}	π_S^m	$=$	π_S^{m+n}
π_1^{n+1}	Y	$=$	$\bullet$
π_1^{n+1}	$Y_\bot$	$=$	$\bullet$
π_1^{n+1}	Y_I	$=$	$\bullet$
π_1^{n+1}	Y_Y	$=$	$\bullet$
π_1^{n+1}	$\pi_{l,l+1}^{n+1}$	$=$	$\pi_{l+n,l+n+1}^{m+n}$

Figure 9: The reduction rules for π_1^{n+1}



Y	$\bot$	$=$	$\bot$
Y	$\top$	$=$	$\top$
Y	π_1^1	$=$	$\bot$
Y	π_1^n	$=$	$\bullet$
Y	π_{i+1}^{n+1}	$=$	π_i^n
Y	$\wedge$	$=$	$\bot$
Y	$\vee$	$=$	$\bullet$
Y	S	$=$	$\bullet$
Y	Y	$=$	$\bullet$
Y	$\wedge^{m+1}$	$=$	$\wedge^m$
Y	$\vee^{m+1}$	$=$	$\vee^m$
Y	π_S^{n+1}	$=$	π_S^n
Y	$Y_\bot$	$=$	$\bot$
Y	Y_I	$=$	π_1^1
Y	Y_Y	$=$	Y
Y	$\pi_{l,l+1}^m$	$=$	$\bullet$

Figure 10: The reduction rules for Y

4.5 How Were These Rules Derived?

Most of the rules were derived in a fairly mechanical way. We show the derivation of a few rules as examples to give the reader a feeling for the process.

Notation: We use x^m and y^m to represent m arguments.

1.

$$\begin{aligned}(S\ \pi_2^2)\ x\ y &= \pi_2^2\ y\ (x\ y)\\ &= \pi_1^1\ (x\ y)\\ &= x\ y\end{aligned}$$

Hence $(S\ \pi_2^2) = \pi_1^1$.

2.

$$\begin{aligned}(\pi_1^{n+1}\ \vee^m)\ x^n\ y^m\ a\ b &= \vee^m\ y^m\ a\ b\\ &= \vee\ a\ b\end{aligned}$$

Hence $(\pi_1^{n+1}\ \vee^m) = \vee^{m+n}$.

3.

$$\begin{aligned}Y\ \pi_1^1 &= \pi_1^1\ (Y\ \pi_1^1)\\ &= Y\ \pi_1^1\\ &\vdots\\ &= \bot\end{aligned}$$

4.

Let $f = (Y\ \wedge)$ then $f = (\wedge\ f)$.

$$\begin{aligned}f_0 &= \bot\\ f_1 &= \wedge\ f_0\\ &= \wedge\ \bot\\ &= \bot\end{aligned}$$

Hence $(Y\ \wedge) = \bot$.

5.

$$\begin{aligned}Y\ \pi_S^{n+1} &= \pi_S^{n+1}\ (Y\ \pi_S^{n+1})\\ &= \pi_S^n\end{aligned}$$

4.6 Examples

4.6.1 Factorial

Our first example is the factorial function.

This function is prototypical of a whole class of functions using "naive" recursion.

$$f\ x = \text{if } x = 1 \text{ then } 1 \text{ else } x * (f\ (x-1))$$

1. Convert to a value declaration

$$f \;=\; \lambda x.\,\text{if } x = 1 \text{ then } 1 \text{ else } x * (f\ (x-1))$$

2. Remove recursion using the $\mathcal{Y}$ combinator

$$f \;=\; \mathcal{Y}\,(\lambda g.\,\lambda x.\,\text{if } x = 1 \text{ then } 1 \text{ else } x * (g\ (x-1)))$$

3. Abstract the function definition

$$f^{\#} \;=\; \mathcal{Y}\,(\lambda g.\,\lambda x.(x \wedge \top) \wedge (\top \vee (x \wedge (g\ (x \wedge \top)))))$$

4. Simplify the definition

$$f^{\#} \;=\; \mathcal{Y}\,(\lambda g.\,\lambda x.x)$$

5. Transform the expression to SK combinator form

$$(\mathcal{Y}\ (K\ I))$$

6. Abstract each constant to an abstract domain element

$$(Y\ (\pi_1^2\ \pi_1^1))$$

7. Reduce down to an abstract domain element

   (a) $(Y\ (\pi_1^2\ \pi_1^1))$
   (b) $(Y\ \pi_2^2)$
   (c) π_1^1

8. Interpret this value:
   Since $\pi_1^1\ \bot\ =\ \bot$ we can conclude that f is strict.

4.6.2 An Undefined Function

$$f\ x \;=\; \text{if } x = 0 \text{ then } f\ x \text{ else } f\ (x-1)$$

1. Convert to a value declaration

$$f \;=\; \lambda x.\,\text{if } x = 0 \text{ then } f\ x \text{ else } f\ (x-1)$$

2. Remove recursion using the $\mathcal{Y}$ combinator

$$f \;=\; \mathcal{Y}\,(\lambda g.\,\lambda x.\,\text{if } x = 0 \text{ then } g\ x \text{ else } g\ (x-1))$$

3. Abstract the function definition

$$f^{\#} \;=\; \mathcal{Y}\,(\lambda g.\,\lambda x.(x \wedge \top) \wedge ((g\ x) \vee (g\ (x \wedge \top))))$$

4. Simplify the definition

$$f^{\#} \;=\; \mathcal{Y}\,(\lambda g.\,\lambda x.x \wedge (g\ x))$$

5. Transform the expression to SK combinator form

$$(\mathcal{Y}\ (S\ \wedge))$$

6. Abstract each constant to an abstract domain element

$$(Y\ (S\ \wedge))$$

7. Reduce down to an abstract domain element

   (a) $(Y\ (S\ \wedge))$

   (b) $(Y\ Y_\perp)$

   (c) $\perp$

8. Interpret this value:
   The function does not terminate even if the argument is defined.

4.7 Correctness Proof Outline

Most of the steps in our algorithm are known to preserve meaning.

The only steps which we need to argue the correctness of are

- Abstraction (6)
- Reduction (7)
- Interpretation (8)

Interpretation is trivial — one analyses the resulting functions using a proven analysis method. The functions are mostly trivial.

Abstraction follows since each element is abstracted to a domain element that has been specifically created to correspond to it.

Reduction then, is the only step in our method the correctness of which is neither known nor obvious.

We need to show that the result of using an abstract reduction rule will always be "faithful"[13] to the result derived by doing a real reduction and then abstracting the result of the real reduction. That is

$$abs\ (eval\ e)\ \leq\ eval^{\#}\ (abs\ e)$$

We will not be giving a full proof since the proof is dependent on the design of the reduction rules associated with the domain and we intend the existing domain only as an illustration.

It is useful to know that any rule that maps to $\top$ (or $\bullet$) is trivially safe.

The method of proof consists of calculating the result in the "real" domain of each application and then comparing the result's abstraction to the abstract reduction result.

The work involved is similar to that presented in section 4.5. We leave the details for the pedantically inclined reader.

[13] that is greater than or equal

5 Implementation

The source can be found in `/home/hons/winikoff/Thesis/src` on munta.

5.1 SKA

The interesting parts of the algorithm above have been implemented in Gofer [Jon].

The program handles simplified lambda expressions involving the $\mathcal{Y}$ combinator. It converts them to SKI combinators and analyses them. In terms of the steps discussed in section 4 it does steps 5 and onwards.

Usage

`ska foo` Analyse file `foo` with embedded expressions producing LaTeX.

Produces the file `foo.tex`

All lines not beginning with `#` are simply copied through to the output file. Lines beginning with a `#` are treated as expressions using the syntax in figure 11. The nonterminal l represents a single letter. The underscore is used to indicate a variable. Thus A is the constant $\wedge$ and $\_x$ is the variable x.

Figure 12 summarizes the constants allowed in expressions.

$$\begin{array}{lcccccl} e & \rightarrow & \backslash x.e & | & (e\ e) & | & x \\ x & \rightarrow & \_l & | & l & & \end{array}$$

Figure 11: Syntax for ska

$$\begin{array}{lcl} S & & \\ I & & \\ K & & \\ Y & & \\ & & \\ A & = & \wedge \\ O & = & \vee \\ T & = & \top \\ B & = & \bot \\ U & = & \top \\ D & = & \bot \end{array}$$

Figure 12: Constants handled by ska

Given an input file `foo`:

```
\subsubsection*{A sample input file to ska}

Consider the example $\ldots$
```

```
#(Y \g.\x.((A _x) (_g _x)))
```

The command `ska foo` will create a file `foo.tex`:

```
%
% This file was produced by ska
% Author: Michael Winikoff
% Date: Mid October 1993
%

\subsubsection*{A sample input file to ska}

Consider the example $\ldots$

Input: $$({\cal Y} \; \lambda g . \lambda x . ((\land \; x) \; (g \; x)))$$

Abstracted: $$({\cal Y} \; ({\cal S} \; \land))$$

Analysis Result: $(\bot)$
$\ldots$  doesn't terminate and hence is strict in all arguments.

\begin{enumerate}
\item $Y \; (S \; \land)$
\item $Y \; Y_\bot$
\item $\bot$

\end{enumerate}
```

Which, when LaTeX'ed produces the output

A sample input file to ska

Consider the example ...

Input:

$$(\mathcal{Y} \; \lambda g . \lambda x . ((\wedge \; x) \; (g \; x)))$$

Abstracted:

$$(\mathcal{Y} \; (\mathcal{S} \; \wedge))$$

Analysis Result: $(\bot)$ ... doesn't terminate and hence is strict in all arguments.

1. $Y \; (S \; \wedge)$
2. $Y \; Y_\bot$
3. $\bot$

5.2 SKO and SKN

These two interactive programs translate lambda expressions in the same syntax as `ska` to SK combinators. They were written earlier on to aid with the design of the domain.

Usage

Note that the programs expect each expression to be on its own line.

`skn` Translate lambda expressions into SKI combinators

`sko` Translate lambda expressions into the full Turner set of combinators[14]

Example

```
Munta> sko
SK program (Optimising)

Author: Michael Winikoff

Eg Usage:  \x.\y.(_x _y)
Note
        1) Use of ``_'' for vars
        2) Use of ``()'' and space for application

\f.\x.(_f (_f _x))
   1) (({\cal S} \; {\cal B}) \; {\cal I})
\x.(_x _x)
   2) (({\cal S} \; {\cal I}) \; {\cal I})
^D
Munta>
```

[14] SKIBCS'B*

6 Optimisations

There are a number of variations of the basic algorithm that improve its accuracy.

The optimisations listed here involve little or no further research. Optimisations whose realisation would involve nontrivial research work are discussed in section 7.

- Taking maximal applications
  The algorithm as it stands will treat a subexpression such as

  $$((S\ Y_Y)\ \pi_1^3)$$

  as two applications to be computed separately. For our sample domain this will give $\top$. We could however, treat this as a single *multiple application*.

  $$\begin{array}{lcl} S\ Y_Y\ \pi_1^3\ x\ y & = & S\ (S\ \pi_1^1)\ \pi_1^3\ x\ y \\ & = & S\ \pi_1^1\ x\ (\pi_1^3\ x)\ y \\ & = & \pi_1^1\ (\pi_1^3\ x)\ (x\ (\pi_1^3\ x))\ y \\ & = & \pi_1^3\ x\ (x\ (\pi_1^3\ x))\ y \\ & = & x \end{array}$$

  Hence $S\ Y_Y\ \pi_1^3\ =\ \pi_1^2$.

  This optimisation is only relevant for combinators that take more than two arguments.

- Distinguish between *primitive* and *compound* domain elements.
  - Definition: A domain element is *primitive* if a constant that can appear in an expression is abstracted to it.
  - Definition: A domain element is *compound* if it is not primitive. (Ie., it can only be generated by an application or can never be generated).

  This distinction doesn't buy us anything directly but it enables us to observe that ...

- If we exploit maximal applications then compound domain elements are never applied — this significantly reduces the number of rules that we have to work with.
  For the sample domain used this means that we only have to work with rules for π, S and $\mathcal{Y}$ (since $\bot$, $\top$, $\wedge$ and $\vee$ behave in the same way regardless of their argument).

- Low fidelity [DW90]
  Rather then analysing functions of multiple variables we can select a particular variable as the one we are interested in analysing and then replace all other variables by $\top$. This reduces the size of the SK expression produced.

  The actual procedure involves defining an auxiliary function

  $$f'\ x\ =\ f^{\#}\ x\ \top$$

  and then analysing f'. For the optimisation to actually buy us something we need to be able to replace the recursive call to $f^{\#}$ with one to f'.

- Simplification
  We can reduce subexpressions involving $\top$, $\bot$, $\vee$ and $\wedge$.

6.1 An Example with Optimisation

The example is the addition function as it would be written using only the predecessor and successor functions. This function is prototypical of a whole class of functions using accumulators.

$$f\ x\ y\ =\ \text{if } x = 0 \text{ then } y \text{ else } f\ (x-1)\ (y+1)$$

We will make use of the low fidelity optimisation.
The function is obviously strict in x.
What about y?
We define

$$f'\ y\ =\ f^{\#}\ \top\ y$$

The actual order of operations is somewhat different from the original algorithm — we abstract and simplify *before* converting to a value declaration.

1. Abstract the function

$$f^{\#}\ x\ y = (x \wedge \top) \wedge (y \vee (f^{\#}\ (x \wedge \top)\ (y \wedge \top))))$$

2. Consider the low fidelity version for y

$$f'\ y = (\top \wedge \top) \wedge (y \vee (f^{\#}\ (\top \wedge \top)\ (y \wedge \top))))$$

3. Simplify

$$f'\ y = (y \vee (f^{\#}\ \top\ y))$$

4. Replace calls to $f^{\#}$ with calls to f' where possible

$$f'\ y = (y \vee (f'\ y))$$

5. Convert to a value declaration

$$f'\ =\ \lambda\, y.(y \vee (f'\ y))$$

6. Remove recursion using the $\mathcal{Y}$ combinator

$$f'\ =\ \mathcal{Y}\ (\lambda\, g.\lambda\, y.(y \vee (g\ y)))$$

7. Transform the expression to SK combinators

$$(\mathcal{Y}\ (S\ \ \vee))$$

8. Abstract each constant to an element of the abstract domain

$$(Y\ (S\ \ \vee))$$

9. Reduce down to a single element of the abstract domain

   (a) $(Y\ (S\ \ \vee))$
   (b) $(Y\ \ Y_I)$
   (c) π_1^1

10. Interpret the result
    Since $\pi_1^1\ \bot\ =\ \bot$ it follows that f' is indeed strict and hence f is strict in y.

7 Further Work

Since the basic method proposed is new there is considerable scope for further work involving its enhancement and application.

- Optimisations
  The optimisations discussed in the previous section could be implemented.

- More optimisations
  There are a number of other optimisations to the method which involve a nontrivial amount of research. These include

  - Argument ordering
    SK abstraction is sensitive to the ordering of arguments. A possible research area is the investigation of heuristics for reordering the arguments to functions.
  - Other combinators
    While SK combinators are perhaps the best known they are by no means the only set of combinators. An interesting area for future investigation involves experimentation with other choices of combinators [Pip87, Sta91, Hir91].

- Other Domains
  The domain we have used is small and simple. It would be desirable to design a larger domain which could handle the full Turner set of combinators.

- Extending the analysis
  The current domain does not handle non-flat data types.

  It is not obvious what extensions would be required to the domain in order to handle compound data types effectively. While capturing elements to a certain depth is straight forward (for example "the list with both head and tail defined") [Bur87, Bur91] we would like to be able to capture *patterns*, such as "the list whose spine is defined".

  In the context of abstract interpretation based analysis we needed to modify the strictness question we were asking — what is the analogue for our analysis method?

- Automatic derivation of the domain
  Designing the domain was a large, tedious and rather mechanical task. What scope is there for the (partial or full) automation of this process?

- Other analyses
  We have been looking at strictness analyses. The insight behind our method is equally applicable to other analyses. An area for further work would be developing other analyses based on this insight.

- Non functional languages
  The class of declarative languages has two main branches, logic and functional. Can our method be carried across to analysis on logic programming languages? How about imperative[15] languages?

[15] Or should that be Imperial? :-)

8 Conclusion

We have explained lazy functional languages and strictness analysis. In our survey of methods we have highlighted inefficiency due to fixpoint iteration as the single most significant obstacle to the realisation of *practical* strictness analysers.

We then proposed a new method based on an insight — if the domain is higher order then we can do the fixpoint iteration *within* the domain. This enables us to avoid fixpoint iteration making our method significantly faster than abstract interpretation based methods. Like both abstract reduction and type inference based methods (and unlike abstract interpretation) our method makes use of an infinite domain.

Our method is also generic in that it is a framework to which we can attach any domain. By comparison abstract reduction is committed to a particular domain. Our method appears to be more efficient than abstract reduction due to the need for path reduction analysis. It is also simpler than abstract reduction and remains closer to abstract interpretation making it easier to demonstrate its correctness.

Like our method, type inference based methods are a generic framework to which one attaches a domain. They are less efficient than our method — in addition to a constraint collection pass over the expression they have to check the consistency of the constraint set collected. This consistency check is a fairly expensive operation and is also quite complex — its definition in [LM91] is in the form of a forty clause function! By comparison our reduction rules for the domain are simple and easily derived.

We continued our presentation with a discussion of possible optimisations and further work. We feel that the framework we have presented lays out a promising alternative to existing strictness analysis methods and is worthy of further work to realise its potential as a fast and practical strictness analysis method.

References

[BHA85] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher-order functions. In *Programs as Data Objects*, pages 42–62. Springer-Verlag, 1985. LNCS 217.

[BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[Bur87] Geoffrey Burn. Evaluation transformers — a model for the parallel evaluation of functional languages (extended abstract). In *Functional Programming and Computer Architecture*, pages 446–470. Springer-Verlag, 1987. LNCS 274.

[Bur91] G.L. Burn. The evaluation transformer model of reduction and its correctness. In *Theory and Practice of Software Development*, pages 458–482. Springer-Verlag, 1991. LNCS 494.

[CP85] Chris Clack and Simon L. Peyton Jones. Strictness analysis — a practical approach. In *Functional Programming and Computer Architecture*, pages 35–49. Springer-Verlag, 1985. LNCS 201.

[Dij79] Edsger W. Dijkstra. An exercise attributed to R.W. hamming. In *A discipline of programming*, pages 129–134. Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[DW90] Kei Davis and Philip Wadler. Strictness analysis in 4D. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming Glasgow*, pages 23–43. Springer-Verlag, 1990.

[Hir91] Sachio Hirokawa. Principal type-schemes of BCI-lambda-terms. In *Theoretical Aspects of Computer Science*, pages 633–650. Springer-Verlag, 1991. LNCS 526.

[HL90] John Hughes and John Launchbury. Towards relating forwards and backwards analyses. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming Glasgow*, pages 101–113. Springer-Verlag, 1990.

[Hol91] Ian Holyer. *Functional Programming With Miranda.* Pitman, 1991.

[Hud89] Paul Hudak. Conception, evolution and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Hug85] John Hughes. Strictness detection in non-flat domains. In *Programs as Data Objects*, pages 112–135. Springer-Verlag, 1985. LNCS 217.

[Hug87] John Hughes. Analysing strictness by abstract interpretation of continuations. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 4, pages 63–102. Ellis Horwood Ltd., 1987. ISBN 0-7458-0109-9.

[Hug89] John Hughes. Projections for polymorphic strictness analysis. In *Category Theory and Computer Science*, pages 82–100. Springer-Verlag, 1989. LNCS 389.

[Hug90a] John Hughes. Compile time analysis of functional programs. In David A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, chapter 5, pages 117–154. Addison Wesley, 1990.

[Hug90b] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, chapter 2, pages 17–42. Addison Wesley, 1990. Also Appeared in Computer Journal April 1989.

[Hun89] Sebastian Hunt. Frontiers and open sets in abstract interpretation. In *Functional Programming and Computer Architecture*, pages 1–13, 1989.

[HY86] Paul Hudak and Jonathan Young. Higher-order strictness analysis in untyped lambda calculus. In *Principles of Programming Languages*, pages 97–109, 1986.

[Jen90] Thomas Jensen. Abstract interpretation vs. type inference: A topological perspective. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming Glasgow*, pages 141–145. Springer-Verlag, 1990.

[Jen91] Thomas P. Jensen. Strictness analysis in logical form. In *Functional Programming and Computer Architecture*, pages 352–366. Springer-Verlag, 1991. LNCS 523.

[Jon] Mark P. Jones. *Gofer Manual.*

[KM87] Tsung-Min Kuo and Prateek Mishra. On strictness and its analysis. In *Principles of Programming Languages*, pages 144–155, 1987.

[KPRS92] O. Kaser, S. Pawagi, C.R. Ramakrishnan, and R.C. Sekar. Fast parallel implementation of lazy languages — the EQUALS experience. In *Lisp and Functional Programming*, pages 335–344. ACM, 1992.

[LGY87] Gary Lindstrom, Lal George, and Dowming Yeh. Generating efficient code from strictness annotations. In *Theory and Practice of Software Development*, pages 140–154. Springer-Verlag, 1987. LNCS 250.

[LM91] Allen Leung and Prateek Mishra. Reasoning about simple and exhaustive demand in higher-order lazy languages. In *Functional Programming and Computer Architecture*, pages 328–351. Springer-Verlag, 1991. LNCS 523.

[MH87] Chris Martin and Chris Hankin. Finding fixed points in finite lattices. In *Functional Programming and Computer Architecture*, pages 426–445. Springer-Verlag, 1987.

[Myc81] A. Mycroft. *Abstract Interpretations and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[NM92] Marc Neuberger and Prateek Mishra. A precise relationship between the deductive power of forward and backward strictness analysis. In *Lisp and Functional Programming*, pages 127–310. ACM, 1992.

[Nöc92] Eric Nöcker. Strictness analysis by abstract reduction in orthogonal term rewriting systems, December 1992.

[Nöc93a] Eric Nöcker. Strictness analysis by abstract reduction in orthogonal constructor systems (extended abstract), April 1993.

[Nöc93b] Eric Nöcker. Strictness analysis using abstract reduction. In *Functional Programming and Computer Architecture*, pages 255–265, 1993.

[PC87] Simon Peyton Jones and Chris Clack. Finding fixpoints in abstract interpretation. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 11, pages 246–265. Ellis Horwood Ltd., 1987. ISBN 0-7458-0109-9.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1987.

[Pip87] Adolfo Piperno. A compositive abstraction algorithm for combinatory logic. In *Theory and Practice of Software Development*, pages 39–51. Springer-Verlag, 1987. LNCS 250.

[Sew93] Julian Seward. Polymorphic strictness analysis using frontiers. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 186–193, 1993. LNCS 274.

[SNvGP91] Sjaak Smesters, Eric Nöcker, John van Groningen, and Rinus Plasmeijer. Generating efficient code for lazy functional languages. In *Functional Programming and Computer Architecture*, pages 592–617. Springer-Verlag, 1991. LNCS 523.

[Sta91] Rick Statman. Freyd's hierarchy of combinator monoids. In *Logic in Computer Science*, pages 186–190, 1991.

[Tur79] D.A. Turner. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9:31–49, 1979.

[Wad87] Philip Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis Horwood Ltd., 1987. ISBN 0-7458-0109-9.

[Wad92] Philip Wadler. The essence of functional programming. Invited Talk at 19th POPL, 1992.

[WH81] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison-Wesley, 1981.

[WH87] Philip Wadler and R.J.M. Hughes. Projections for strictness analysis. In *Functional Programming and Computer Architecture*, pages 385–407. Springer-Verlag, 1987. LNCS 274.

[Wri91] David A. Wright. A new technique for strictness analysis. In S. Abramsky and T. Maibaum, editors, *Theory and Practice of Software Development*, pages 235–258. Springer-Verlag, 1991.

A A Brief Overview of Lattice Theory

A binary relation $\mathcal{R}$ is

- *Reflexive* iff $\forall\, x\ x\mathcal{R}x$
- *Transitive* iff $\forall\, x \forall\, y \forall\, z\ ((x\mathcal{R}y) \wedge (y\mathcal{R}z)) \Rightarrow (x\mathcal{R}z)$
- *Anti-Symmetric* iff $\forall\, x \forall\, y\ ((x\mathcal{R}y) \wedge (y\mathcal{R}x)) \Rightarrow (x = y)$

A *preorder* is a binary relation which is reflexive and transitive.
A *partial order* is a preorder which is also antisymmetric.
Given a domain X and a subset of the domain $Y \subseteq X$.

- $x \in X$ is an *upper bound* for Y iff $\forall\, y \in Y\ (y\mathcal{R}x)$
- $x \in X$ is a *lower bound* for Y iff $\forall\, y \in Y\ (x\mathcal{R}y)$

$x \in X$ is a *least* upper bound for Y iff x is an upper bound and all upper bounds are greater or equal to x.
Notation:

$$x = lub Y \text{ or } x = \bigsqcup Y$$

We also use a binary version of this operator written

$$x \vee y \text{ or } x \textit{ join } y$$

Dually x is a *greatest* lower bound iff it is a lower bound for Y and all lower bounds are less than or equal to it.
Notation:

$$x = glb Y \text{ or } x = \bigsqcap Y$$

We also use a binary version of this operator written

$$x \wedge y \text{ or } x \textit{ meet } y$$

Y is a *chain* iff $\forall\, y, y' \in Y\ (y\mathcal{R}y') \vee (y'\mathcal{R}y)$.
A set X with a partial order defined on it (a poset) is a *lattice* if $\bigsqcup Y$ and $\bigsqcap Y$ exist for all *finite* subsets $Y \subseteq X$.
A set X with a partial order defined on it (a poset) is a *complete* lattice if $\bigsqcup Y$ and $\bigsqcap Y$ exist for *all* subsets $Y \subseteq X$.
We define

$$\top = \bigsqcup X = \bigsqcap \emptyset \text{ and } \bot = \bigsqcap X = \bigsqcup \emptyset$$

Let X and Y be complete lattices. Let $f :: X \to Y$ and $g :: X \to X$ be functions.
Then f is *monotonic* iff

$$\forall\, x, x'\ (x\mathcal{R}x') \Rightarrow (f(x)\ \mathcal{R}\ f(x'))$$

f is *continuous* iff

$$f(\bigsqcup Z) = \bigsqcup \{f(z) \mid z \in Z\}$$

for all non-empty chains Z.
If f is continuous then it is also monotonic.
An element $x \in X$ is a *fixpoint* of g iff $g(x) = x$.
An element $x \in X$ is the *least* fixpoint of g if it is a fixpoint of g and all fixpoints are greater than or equal to it.

If g is a function operating over complete lattices and g is monotonic then the *least fixpoint* of g is well defined.
If $g :: X \to X$ is a *continuous* function over a complete lattice then its least fixpoint is well defined and is the limit of the sequence

$$\bot,\ (g\bot),\ (g^2\bot),\ (g^3\bot)\ldots$$