Strictness Analysis as Finite-Domain Constraint Solving*

Tihomir Gabrić and Harald Søndergaard
Dept. of Computer Science
The University of Melbourne
Parkville Vic. 3052, Australia
gabric@cs.mu.oz.au, harald@cs.mu.oz.au

Abstract

It has become popular to express dataflow analyses in "logical form". We are interested in the possibility of a uniform approach based on either inference of constrained types or on synthesis of constraint logic programs. In this paper we sketch how the language Toupie, originally designed with logic program analysis as one objective, lends itself also to sophisticated strictness analysis. Strictness analysis is straightforward in the simplest case, that of analysing a first-order functional language using just two strictness values, divergence and "don't know". Mycroft's classical translation immediately yields perfectly valid Boolean constraint logic programs, which, when run, provide the desired strictness information. However, more sophisticated analysis requires more complex domains of strictness values. We recast Wadler's classical analysis over a $2n$-point domain as finite-domain constraint solving. This allows us to leverage from established constraint-solving technology. Moreover, unlike Wadler's method, it allows us to translate a recursive function definition into a *single* constraint program clause, in a manner which extends Mycroft's translation for the 2-point case, while maintaining the precision of Wadler's method. In fact, a higher precision is gained, since a constraint logic programming approach naturally leads to *disjunctive* analysis.

1 Introduction

Strictness is an important concept in the implementation of lazy functional programming languages. To improve the generated code, compilers for such languages usually perform some sort of strictness analysis, trying to establish cases where call-by-name evaluation can be replaced by call-by-value (without changing the program's behaviour). A seminal study of strictness analysis was done by Mycroft who suggested that analysis can be viewed as abstract interpretation [5]. Mycroft considered first-order programs and gave an appealing translation method for such programs; his strictness analysis consists of translating the given program into a functional program that manipulates only Boolean values. Running the resulting program yields strictness information about the original program.

Strictness analysis was subsequently extended in two directions, to handle higher-order programs, and to deal with more complex strictness information. More complex information is usually needed in the context of lists and user-defined data structures. For example, the termination properties of a list-processing function may depend on the "shape" of the argument only,

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

as is the case with the function `length` which terminates iff the input list is finite, independently of the termination properties of the list's elements. Semantically the more complex information can be modelled by "non-flat" domains. Considerable progress was made when Wadler [8] proposed a precise strictness analysis based on case analysis over a domain of four (or in fact a family of domains with $2n, n \geq 1$) strictness values. Wadler's four-element domain is:

$\top_\in$	Any list
$\vert$	
$\perp_\in$	Any infinite list, and any finite list, some member of which is $\perp$
$\vert$	
∞	Any infinite list, and any list whose tail is $\perp$
$\vert$	
$\perp$	The undefined list

There is no simple equivalent of Mycroft's translation in the non-flat case. This is not only due to the fact that strictness properties no longer form a two-valued Boolean domain. As Wadler points out, reasoning by enumerating cases seems necessary in the non-flat case and is not captured in a Mycroft-style translation. For example, we generally have that $(\mathit{head}\ \perp_\in) = \top$, where $\top$ is the top element of the usual two-valued strictness domain, **2**, that is, $\top$ stands for any value, defined or not. We also have that $(\mathit{tail}\ \perp_\in) = \top_\in$, since $\top_\in$ may represent, say, $[\perp, 2]$. However, these simple mappings do not tell the full story. For example, if $(\mathit{head}\ \perp_\in)$ is known to be defined, that is, not to be $\perp$, then $(\mathit{tail}\ \perp_\in)$ must be $\perp_\in$. Such interrelations among dependent sub-expressions are lost in a naive translation.

We are interested in the use of constraint programming languages for program analysis, and one contribution of this paper is to show that a translation scheme for strictness analysis over a non-flat domain is not only possible, but also natural, if we work in a finite-domain constraint programming language. Dependent sub-expressions can then be handled via the introduction of "linking variables" (logic variables) and their subsequent removal can be performed via existential quantification. This translation has advantages. It is easy to understand and quite general. The need to tabulate abstract functions exhaustively disappears. Finally, sophisticated constraint solving technology promises to make analysis fast.

We have implemented a method for a simple functional programming language with lists and pattern matching. The constraint language we use is Toupie [1]. Toupie can be considered a finite-domain μ-calculus interpreter which allows fixpoint problems to be stated succinctly and solved efficiently. It was originally designed for the abstract interpretation of logic programs. However, it would seem that finite-domain constraint solvers would be of great help for a variety of static analyses used in the compilation and transformation of functional programs, and a uniform approach to several such analyses, for example usage analysis and binding-time analysis, seems perfectly possible.

In Section 2 we outline classical strictness analysis via abstract interpretation and sketch the approach that uses CLP. Section 3 gives details of how we translate a functional program to a Toupie program that performs the strictness analysis. Section 4 discusses the handling of other data structures, and Section 5 concludes.

2 Strictness Analysis

Consider the functional program

```
power x n  = if n=0 then 1 else x * (power x (n-1));
```

Following Mycroft [5] we can analyse this program for strictness by translating it to a recursively defined Boolean function:

$$power^{\#}(x,n) = n \wedge true \wedge (true \vee (x \wedge power^{\#}(x, n \wedge true))).$$

The idea is to translate termination behaviour into propositional logic, associating *truth* with lack of information and *falsehood* with inevitable non-termination. Thus a constant such as `0` translates to *true*, since it would be wrong to claim that evaluation of `0` fails to terminate, while the comparison `n=0` translates to $n \wedge true$ because '=' is strict in both positions. So evaluation of the expression `n=0` will terminate if and only if evaluation of `n` terminates.

It is easy to see that we can simplify the right-hand side of the equation for *power* to get

$$power^{\#}(x,n) = n.$$

Clearly then the evaluation of `power` fails to terminate if evaluation of its second argument does. So `power` is strict in position 2.

In general, when solving recursive equations, we cannot rely on algebraic simplifications to make recursion "go away", as happened above. We then resort to Kleene iteration. Consider the program

```
gcd x y = if x=y then x
          else if x>y then gcd (x-y) y
          else gcd x (y-x);
```

The corresponding strictness equation is

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

To find a closed form, we iterate from the initial guess $gcd^{\#}(x,y) = \mathit{false}$, eventually reaching the fixpoint $gcd^{\#}(x,y) = x \wedge y$.

In the case of list-manipulating programs, we need more than two strictness values to get useful information. A list may be (partially) divergent, or some of its elements may diverge. Wadler's domain allows us to identify cases where list-processing functions can safely evaluate their arguments using special evaluation mechanisms, such as element reduction, spine reduction, or reduction to head normal form.

Consider the program

```
sum []     =  0;
sum (x:xs) =  x + sum xs;
```

(In programs we use ':' for list construction, but outside programs we refer to the constructor as '*cons*'.) To translate the program into strictness equations, Wadler first determines that the best approximation to *cons* is as follows. For the possible values of x and xs, $cons(x, xs)$ is:

$x \backslash xs$	$\bot$	∞	$\bot_{\in}$	$\top_{\in}$
$\top$	∞	∞	$\bot_{\in}$	$\top_{\in}$
$\bot$	∞	∞	$\bot_{\in}$	$\bot_{\in}$

This makes it possible to set up (and subsequently solve) four equations describing the best approximation to *sum*, by a tedious but mechanical case analysis:

$$\begin{array}{lcl} sum^{\#}\ \top_{\in} & = & \top \sqcup (\top \sqcap (sum^{\#}\ \top_{\in})) \\ sum^{\#}\ \bot_{\in} & = & (\bot \sqcap (sum^{\#}\ \top_{\in})) \sqcup (\top \sqcap (sum^{\#}\ \bot_{\in})) \\ sum^{\#}\ \infty & = & \top \sqcap (sum^{\#}\ \infty) \\ sum^{\#}\ \bot & = & \bot \end{array}$$

For example, the equation for $sum^{\#}\ \top_{\in}$ is generated by observing that the (list) argument $\top_{\in}$ would either have been `[]` or it would have been a composite list. In the former case, the right-hand side translates to '$\top$'. In the latter case, the table for *cons* shows that the list would have to be composed from prepending $\top$ to $\top_{\in}$, so the translation of '`x + sum xs`' should be $\top \sqcap (sum^{\#}\ \top_{\in})$. The result is finally obtained by taking the join of the two possible results.

Solving the four recursive equations, we arrive at the solution

$$sum^{\#}\ xs = \begin{cases} \top & \text{if } xs = \top_{\in} \\ \bot & \text{otherwise.} \end{cases}$$

So the sum of a list is defined only if the list is finite and all of its elements are non-$\bot$, that is, they do not diverge.

This was the traditional method. Note that with this method, an n-place function calls for up to 4^n (mutually recursive) equations to be solved. We now sketch a solution which uses constraints and generates only one (albeit more complex) recursive equation. First, it is convenient to use domains of integers to represent the strictness values (such as $\mathbf{2} = \{1, 2\}$ and $\mathbf{4} = \{1, 2, 3, 4\}$), since then the constraint solver knows about the intended ordering of these values. Thus, instead of using $\bot_{\in}$ we use 3 ($\in \mathbf{4}$), and so on. Note that it is not necessary to treat strictness properties as integers in Toupie—indeed it is easy to capture partially ordered sets of properties—but in our case integers and their linear ordering are convenient.

Here is how we translate `sum` into a recursive constraint equation (exact rules appear in Section 3):

$$\begin{array}{lcll} sum^{\#}(xs, res) & += & & (xs = 1 \wedge res = 1) \\ & & \vee & (xs = 4 \wedge res = 2) \\ & & \vee & \exists\ hd, tl, res' : \\ & & & res = min(hd, res') \wedge sum^{\#}(tl, res') \wedge cons^{\#}(hd, tl, xs) \\ & & & \\ cons^{\#}(hd, tl, xs) & = & & xs = max(2, min(2 + hd, tl)) \end{array}$$

The $+=$ indicates that we want the least solution to the recursive equation. The $cons^{\#}$ rule captures the effect of list construction. This clause is used every time a program involves list construction or list patterns—but the domains of the variables may change, as we shall see later. In the example, *hd* belongs to **2** and *tl* and *xs* both belong to **4**. Since Toupie needs to know the domains of variables, a simple type analysis is necessary to establish these.

Notice how we have built in reasoning about dependencies of sub-expressions, but hidden the mechanics with the use of existential quantifiers. In this sense we have generalised Mycroft's method to non-flat domains.

The solution to the recursive equation for $sum^{\#}$ is left to the constraint programming language. Notice that it is perfectly possible to obtain a closed form for $sum^{\#}$, by Kleene iteration and constraint solving. The Toupie solution comes out as

```
append(X0,X1,Result) +=
    exist V0,V1,V2,V3
    (  (  (  (V0 = Result)
          &  nil(X0)
          &  (V0 = X1)
          )
       |  pat_bot(X0,Result)
       )
    |  (  (  exist E0
             (  cons(V1,E0,Result)
             &  append'(V2,V3,E0)
             )
          &  cons(V1,V2,X0)
          &  (V3 = X1)
          )
       |  pat_bot(X0,Result)
       )
    )
```

```
pat_bot(X0,X1) +=
    (  (X0 = 1)
    &  (X1 = 1)
    )
```

Figure 1: The Toupie code generated for analysis of `append`

```
{Xs=1, Res=1}
{Xs=2, Res=1}
{Xs=3, Res=1}
{Xs=4, Res=2}
```

which is in agreement with Wadler's analysis. In Section 5 we discuss so-called disjunctive strictness analysis and argue that the constraint approach has the potential to offer greater precision than the classical approach.

The programs that we generate are slightly more verbose than indicated by the example. Consider the following program:

```
append []     ys = ys;
append (x:xs) ys = x : append xs ys;
```

The Toupie translation of `append` is given in Figure 1. (Again notice that we produce *one* recursive equation to solve, rather than the 16 equations suggested by the traditional case analysis.) Most resulting Toupie programs are easily simplified, for example by eliminating existentially quantified variables. However, we present Toupie programs as they are produced by the translator.

3 The Translation

In this section we give the details of the simple functional language used, and the rules for translating programs into Toupie programs. The syntax of our language is given in Figure 2. It is a simple first order functional language with pattern matching on lists and integer constants.

Figure 3 gives the algorithm used to translate functional programs into Toupie, and Figure 4 shows how expressions and patterns are translated. Note that patterns are translated using the same techniques as expressions. Also note that results returned by functions may be lists, and

```
Prog ::= {Def}*                        PatList ::= Pat
                                                 | Pat ":" PatList
 Def ::= Fname {Pat}* "=" Exp ";"
                                           Exp ::= Exp Op Exp
 Pat ::= Integer                                 | Integer
       | Var                                     | Fname {Exp}*
       | "[]"                                    | "[]"
       | "(" PatList ")"                         | "(" Exp ")"
                                                 | "if" Exp "then" Exp "else" Exp
  Op ::= "+" | "-" | etc...                      | Var
```

Figure 2: A simple functional language

For each function f defined in P do:
  Let k = the arity of f;
  Let $D_1, \ldots, D_n$ be the n equations defining f, renamed apart so that
    $vars(D_i) \cap vars(D_j) = \emptyset$ whenever $i \neq j$;

  For each $i \in \{1, \ldots, n\}$ do:
    Let $[\![f\ p_{i1}\ \ldots p_{ik} = e_i]\!] = D_i$
    Let $V = vars\{p_{qr} : q \in \{1 \ldots n\}, r \in \{1 \ldots k\}\}$

  Return $[\![\, f^{\#}(X_1, \ldots, X_k, R) \mathrel{+}= \exists\, V : e_1' \vee \ldots \vee e_n']\!]$
    where $e_i' \;=\; (\mathcal{E}[\![e_i]\!]\ R\ \wedge \mathcal{E}[\![p_{i1}]\!]\ X_1 \wedge \ldots \wedge \mathcal{E}[\![p_{ik}]\!]\ X_k) \vee pMatch(p_{i1}, \ldots, p_{ik})$

Figure 3: Translating function definitions

so they cannot adequately be represented as Boolean values—hence the use of a `Result` variable in a logic programming style.

One complication is that the translation, outlined in Figure 3, must account for the use of (eager) pattern matching. During execution, if a function needs to compare some input value against a pattern, it must first reduce the value to head normal form. Only then is it able to make a meaningful comparison. However, if the input argument is $\perp$, this comparison does not terminate. The function *pMatch* takes all the input patterns for a given definition D_i, and for each non-variable pattern p_{iq} it produces a constraint $(R = \perp) \wedge (X_q = \perp)$. It then forms the disjunction of these.

The rules for translating expressions are given in Figure 4. The function $\mathcal{E}$ takes an expression and a result variable name, and constructs a Toupie expression to constrain the result. For simplicity the rules assume that analysis of lists uses the 4-point domain. In the next section we look at the handling of nested lists.

4 Other Data Types

The translation extends easily to handle cases with nested lists, using ideas from Wadler [8]. For example, lists of lists are approximated by the six-element domain **6**:

$$
\begin{array}{rcl}
\mathcal{E}[\![n]\!]\ R & = & (R = \top) \\[2ex]
\mathcal{E}[\![[]]\!]\ R & = & (R = \top_{\in}) \\[2ex]
\mathcal{E}[\![v]\!]\ R & = & (R = v) \\[2ex]
\mathcal{E}[\![e_0\ \langle op \rangle\ e_1]\!]\ R & = & \exists\ E_0, E_1 : \mathcal{E}[\![e_0]\!]\ E_0 \wedge \mathcal{E}[\![e_1]\!]\ E_1 \wedge (R = E_0 \sqcap E_1) \\[2ex]
\mathcal{E}[\![e_0 : e_1]\!]\ R & = & \exists\ E_0, E_1 : \mathcal{E}[\![e_0]\!]\ E_0 \wedge \mathcal{E}[\![e_1]\!]\ E_1 \wedge cons(E_0, E_1, R) \\[2ex]
\mathcal{E}[\![f\ e_1 \ldots e_k]\!]\ R & = & \exists\ E_1, \ldots, E_k : \\
 & & f^{\#}(E_1, \ldots, E_k, R)\ \ \wedge \mathcal{E}[\![e_1]\!]\ E_1 \wedge \ldots \wedge \mathcal{E}[\![e_k]\!]\ E_k \\[2ex]
\mathcal{E}[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!]\ R & = & \exists\ E_1, E_2, E_3 : \\
 & & (\mathcal{E}[\![e_1]\!]\ E_1) \wedge (\mathcal{E}[\![e_2]\!]\ E_2) \wedge (\mathcal{E}[\![e_3]\!]\ E_3) \wedge \\
 & & (((R = E_2 \vee R = E_3) \wedge (E_1 = \top)) \\
 & & \vee((R = \bot) \wedge (E_1 = \bot)))
\end{array}
$$

Figure 4: Translating expressions into constraints

$$
\begin{array}{c}
\top_{\in\in} \\ | \\ \bot_{\in\in} \\ | \\ \infty_{\in} \\ | \\ \bot_{\in} \\ | \\ \infty \\ | \\ \bot
\end{array}
$$

Here $\bot$ and ∞ have the same meaning as before, while an element of the form $d_{\in} \in \mathbf{6}$ corresponds to a list, some member of which is described by $d \in \mathbf{4}$. For example, $[[1, \bot, 3]]$ is approximated by $\bot_{\in\in}$ but not by $\infty_{\in}$.

An initial type analysis determines the domains for the various variables. Consider the following program which uses `append` from before:

```
revall xss = rev (maprev xss);

maprev [] = [];
maprev (x:xs) = rev x : maprev xs;

rev [] = [];
rev (x:xs) = append (rev xs) (x:[]);
```

```
revall(X0,Result) +=
    exist V0
    (  exist E0
       (  rev(E0,Result)
       &  maprev(V0,E0)
       )
    &  (V0 = X0)
    )

maprev(X0,Result) +=
    exist V0,V1
    (  (  (  (Result = 6)
          &  (X0 = 6)
          )
       |  pat_bot(X0,Result)
       )
    |  (  (  exist E0,E1
             (  cons(E0,E1,Result)
             &  rev(V0,E0)
             &  maprev(V1,E1)
             )
          &  cons(V0,V1,X0)
          )
       |  pat_bot(X0,Result)
       )
    )
```

```
rev(X0,Result) +=
    exist V0,V1
    (  (  (  (Result = 6)
          &  (X0 = 6)
          )
       |  pat_bot(X0,Result)
       )
    |  (  (  exist E0,E1,E2
             (  append(E0,E1,Result)
             &  rev(V1,E0)
             &  (  cons(V0,E2,E1)
                &  (E2 = 1)
                )
             )
          &  cons(V0,V1,X0)
          )
       |  pat_bot(X0,Result)
       )
    )

pat_bot(X0,X1) +=
    (  (X0 = 1)
    &  (X1 = 1)
    )
```

Figure 5: Toupie clauses generated for the `revall` program

The generated Toupie clauses are shown in Figure 5. In this case the list variables are in **6**. Notice that *cons* need not be redefined—it was defined in such a way that it extends gracefully. The output from asking the most general query is shown below, both as output from Toupie and in tabular form.

```
p |= revall(A,R)?
lambda (A@1:wad6, R@2:wad6) revall(A,R)
{A=1, R=1}
{A=2, R=1}
{A=3, R=3}
{A=4, R=3}
{A=5, R=5}
{A=6, R=6}
```

input	output
$\top_{\in\in}$	$\top_{\in\in}$
$\perp_{\in\in}$	$\perp_{\in\in}$
$\infty_{\in}$	$\perp_{\in}$
$\perp_{\in}$	$\perp_{\in}$
∞	$\perp$
$\perp$	$\perp$

Handling other free data types is not hard. In general we will not obtain a linearly ordered domain of abstractions, but the ordering relation can be programmed explicitly in the CLP language.

Alternatively, the components of a user-defined data structure can be represented by separate logic variables. For functions of several variables we have seen how detailed information can be maintained about their interrelation. This principle can be extended to include the components of results that are composite. As we also have the ability, for functions defined by cases, to return constraints that are *disjunctions*, rather than taking least upper bounds of result values, we obtain program analyses with a very fine granularity. As an example, consider the program

```
f x = if B then (x,17) else (17,x);
g x = add (f x);
```

Jensen [4] uses this example to show how disjunctive information readily translates into greater precision. Assume the value of the Boolean expression `B` cannot be determined at analysis time. If we were to take the join of the two contributions from the branches of the conditional, we would end up with $(x, true) \sqcup (true, x) = (true, true)$ as the result of analysing `f`. Based on this, the result for `g` would be *true*, that is, we fail to recognise that `g` is strict. A translation into a CLP language naturally generates separate disjunctive branches. In this case,

```
f(X0,(R0,R1)) +=
    (   <B>
    &   (   (R0 = X0) & (R1 = 1)
        |   (R0 = 1) & (R1 = X0)
        )
    )

g(X0,R) +=
    (   f(X0,(R0,R1))
    &   (R = min(R0,R1))
    )
```

A query `g(1,R)` will then return the answer `R=1` as desired.

5 Conclusion

It has become popular to cast analyses for functional programs analysis in logical form. Examples are Jensen's "strictness logic" (for example [4]), and "usage logic" as developed by Wadler and by Wright and Baker-Finch—for an attempt to provide a unifying framework for these, see Wright [9]. Another trend is the formulation of analyses as "constraint" problems, as in the case of binding-time analysis [3] which appears to use a combination of Boolean and Herbrand constraints. More recently, Sekar and Ramakrishnan [6] have embraced constraint solving methods for strictness analysis. In all cases, the invention of some constraint system is involved, so it is natural to ask what constraint programming can offer, and whether some uniform approach can be developed.

There are two natural avenues for this research. One can view analysis as inference of "constrained types". A good candidate for a basis for this is Sulzmann, Odersky and Wehr's $HM(X)$ [7], which is a Hindley-Milner type system extended with additional constraint reasoning capabilities. The 'X' refers to the constraint domain applied, so $HM(X)$ is parametric with respect to constraint domains.

This paper has explored the alternative avenue, that we can readily exploit available constraint programming technology for non-trivial program analysis. To this end we have developed a method for strictness analysis over non-flat domains, using a constraint programming language to express strictness properties of the program under analysis. This method reconciles Mycroft's classical two-valued approach with Wadler's technique for the non-flat case, translating a function definition into a single constraint program clause. We find this preferable to the classical method of generating up to 4^n recursive equations for an n-place function.

There have been other approaches to non-flat strictness analysis, such as projection analysis and PER analysis, see for example Cousot and Cousot [2]. We have chosen to avoid the intricacies of these because their advantages are not obvious from a practical point of view. For a compiler, 'strictness' translates directly to the knowledge that an argument can safely be evaluated at call time, and this would seem more useful information than simple 'absence', the knowledge that an argument is not used. Wadler's domain appears to offer the right kind of information for specialising the evaluation mechanism at compile time.

The constraint programming approach allows us to improve the granularity of information, partly because it naturally leads to disjunctive analysis, and partly because the components of composite data structures can be represented by separate logic variables. Furthermore, complex interactions amongst both input and output components are naturally captured by the introduction of local "linking" variables that are subsequently eliminated through existential quantification.

References

[1] M.-M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient bottom-up abstract interpretation of Prolog by means of constraint solving over symbolic finite domains. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, LNCS 714, pages 75–91. Springer-Verlag, 1993.

[2] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages. In *Proc. 1994 Int. Conf. Computer Languages*, pages 95–112. IEEE Computer Society, 1994.

[3] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, LNCS 523, pages 448–472. Springer-Verlag, 1991.

[4] T. P. Jensen. Disjunctive strictness analysis. In *Proc. Seventh Ann. IEEE Symp. Logic in Computer Science*, pages 174–185, 1992.

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

[6] R. Sekar and I. V. Ramakrishnan. Fast strictness analysis based on demand propagation. *ACM Trans. Programming Languages and Systems*, 17(6):896–937, 1995.

[7] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types (extended abstract). In B. Pierce, editor, *Proc. Fourth Int. Workshop Foundations of Object-Oriented Languages*, 1997. http://www.cs.indiana.edu/hyplan/pierce/fool.

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

[9] D. A. Wright. Linear, strictness and usage logics. In M. Houle and P. Eades, editors, *Proc. CATS'96*, Australian Computer Science Communications **18** (3), pages 73–80, 1996.