Automatic Extraction of Context Information from Programs Using Abstract Reduction

Marko Schütz*, Manfred Schmidt-Schauß
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
e-mail: {marko,schauss}@cs.uni-frankfurt.de
Phone: +49 (69) 798-22890
Fax: +49 (69) 798-28213

May 28, 1997

Abstract

This paper describes context analysis, an extension to strictness analysis for lazy functional languages. In particular it extends Wadler's four point domain and permits infinitely many abstract values. A calculus is presented based on abstract reduction which given the abstract values for the result automatically finds the abstract values for the arguments.

The results of the analysis are useful for verification purposes and can also be used in compilers which require strictness information.

1 Introduction

Optimization is a very important stage in any compiler and, due to the high degree of abstraction, particularly in compilers for lazy functional languages like Haskell, Clean, etc. The optimizations need to be based on static analysis of the program at hand. One such analysis is strictness analysis which tries to find the arguments in a function application that are sure to be evaluated when evaluating the application. A generalization of strictness analysis is context analysis which tries to find the degree of evaluation that is required of the argument given a degree of evaluation required of the application.

While there has been some research on the exploitation of strictness information for optimization e. g. [PJS94, HB93, PJL91, PJP] research on context information and its use e. g. [LB96, FB94, Sew92, Far92] has not received much attention.

Many approaches to strictness analysis define a finite set of abstract values for any particular type beforehand for use in the analysis, e. g. Wadler's 4-point

*This work was supported by the Deutsche Forschungsgemeinschaft under Az Schm 986/4-1

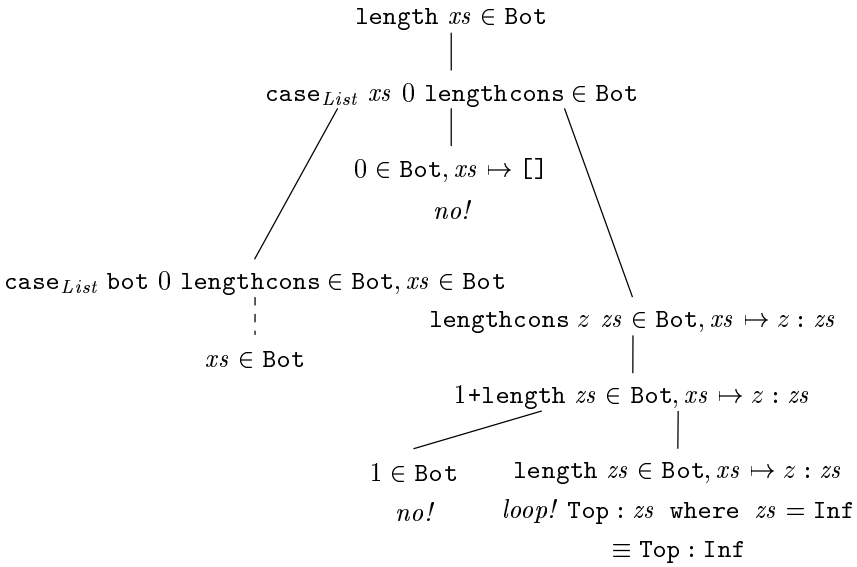
domain [Bur91, Wad87]. Often the motivating claim is that the amount of evaluation an expression can experience depends on its type. For the approach presented in this paper there is an infinity of possible evaluation contexts. It comes as no surprise that from our point of view the amount of evaluation which an expression may experience is determined by the functions present in the program that consume arguments of the expressions type.
For verification purposes our calculus will answer questions such as "For which arguments will an application of a function yield a value of a specified definedness?".
Another difference to other work on context analysis is that our approach is *constructive*, i.e. it constructs abstract values for which certain relations hold. As an example consider an application of `sum`. Typically, context analysis would give the result *true* when presented with the question whether `sum` applied to an infinite list argument had no WHNF. In contrast, our approach will result in a representation of all the arguments for which `sum` has no WHNF. Clearly, the latter is much better suited to verification than the former. Another advantage of abstract reduction over abstract interpretation can be seen when analyzing a higher order function f. If in the reduction there is no redex where f applied to variables for the higher order arguments appears, but instead all higher order arguments have either a supercombinator or a constructor as WHNF then our calculus is able to handle the analysis. Many approaches to strictness analysis using abstract interpretation are only able to handle this kind of analysis if they are able to analyze higher order functions. In this respect these analyses work similar to strict evaluation and compute all of the information about a function f before analyzing a function g which uses the information. Abstract reduction works similar to lazy evaluation and does not compute information, which is not explicitly requested or needed to obtain the requested information.
To use the results of our calculus for optimization the compiler could allow the user to specify optimization heuristics, i.e. evaluation transformers [FB94, Bur91], or tell the compiler how to use an optimized data representation as in [Hal94].
In the terminology of [CC77] our approach would be called abstract interpretation, yet subsequent work by [Myc80, BHA85, Bur87, Bur91] and others used the term abstract interpretation almost exclusively for abstract interpretation abstracting the *denotational* semantics. Therefore we use the term abstract reduction as coined by Nöcker [Nöc92, Nöc93, vEGHN93] for abstract interpretation abstracting the *operational* semantics to distinguish it from abstract interpretation abstracting the denotational semantics.
According to the four dimensions for qualifying strictness analysis in [DW90] the approach is first order (but see section 9), non-flat, high fidelity and backwards. The high-fidelity combined with the constructiveness of our approach lead to *joint evaluation contexts* for all arguments of a function.
The context inclusions we use bear a strong resemblance to set constraints [Aik94, HJ94]. The most important difference is the semantics used. Solutions of set constraints are sets of ground terms whereas the concretization of a context can be defined as a least fixed point of an appropriate function using ground expressions over Λ_c, i.e. programs with their input. It follows that the context $inf = \langle \texttt{Bot}, \texttt{Top} : inf \rangle$ represents `let c = 1:c in c` but the same expression interpreted as a set constraint does not.
Our calculus has been implemented and the implementation has been run on

inputs including all the examples in this paper.
To give an intuitive notion of the working of our calculus we will provide some simple examples.
For the purpose of the following motivating examples it will suffice to know that $\texttt{case}_A$ evaluates to WHNF and decomposes its first argument and applies to the components of this WHNF the appropriate function. Intuitively, the nodes consist of constraints and substitutions where the substitutions record the structure of variables assumed for a particular path.

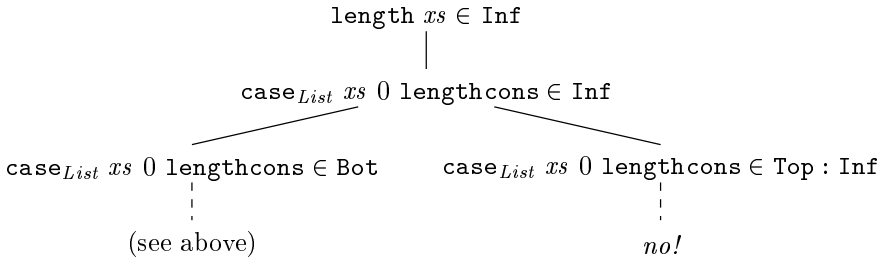
Example 1. As an example consider

$$\begin{array}{lcl}
\texttt{append}\ xs\ ys & = & \texttt{case}_{List}\ xs\ ys\ (\texttt{appendcons}\ ys) \\
\texttt{appendcons}\ ys\ z\ zs & = & z : (\texttt{append}\ zs\ ys) \\
\texttt{length}\ xs & = & \texttt{case}_{List}\ xs\ 0\ \texttt{lengthcons} \\
\texttt{lengthcons}\ z\ zs & = & 1 + \texttt{length}\ zs
\end{array}$$

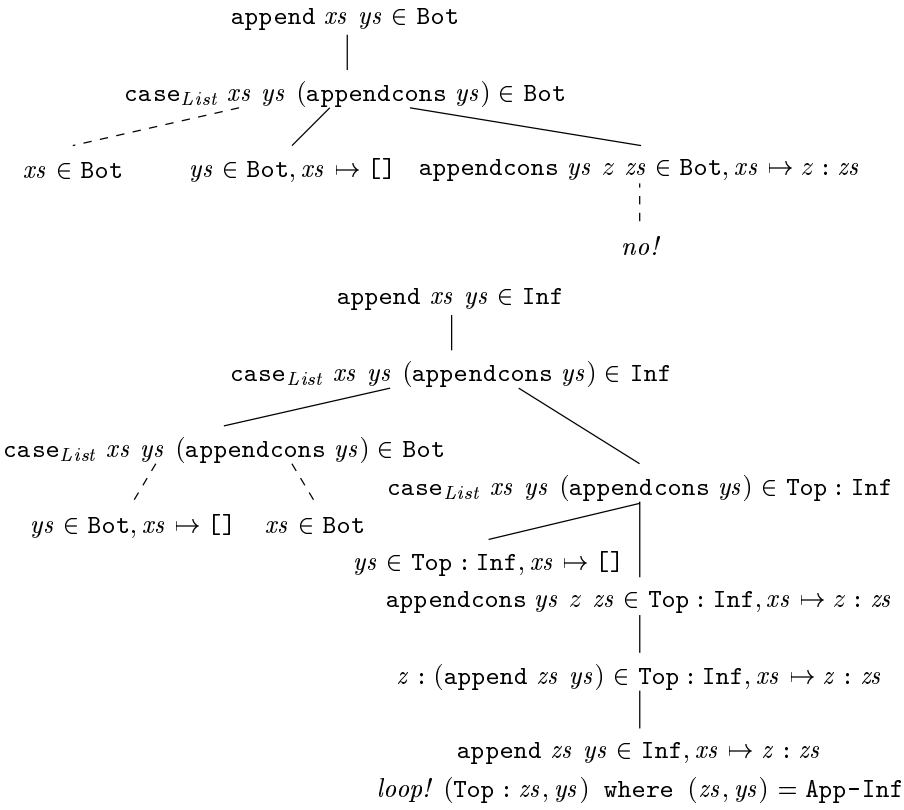
Assume no analysis has taken place, so the only relevant context is Bot.
The analysis of



yields $\texttt{Inf} = \langle \texttt{Bot}, \texttt{Top} : \texttt{Inf} \rangle$. Inf is an arbitrary name given to the representation of the solution. Intuitively, this means: evaluate the argument to WHNF, if it has a ':' as the top level constructor disregard the head and evaluate the tail in the same context as the original argument. Refer to section 4 for a treatment of contexts. Subsequently we use the abstract value Inf just as Bot is used.



Thus the result for $\texttt{length}\ xs \in \texttt{Inf}$ is precisely the same as for $\texttt{length}\ xs \in \texttt{Bot}$. A compiler which has access to type information for the function being analyzed could have skipped the analysis of $\texttt{length}\ xs \in \texttt{Inf}$. Analogously, we obtain $\texttt{Botel} = \langle \texttt{Bot}, \texttt{Top} : xs\ \texttt{where}\ xs = \texttt{Botel}, \texttt{Bot} : \texttt{Top} \rangle$ from the analysis of $\texttt{sum}\ xs \in \texttt{Bot}$. This result is added to $\mathcal{C}$.
We proceed to the analysis of `append`:



The result is

$$\begin{aligned}\texttt{App-Inf} &= \langle (\texttt{Bot}, \texttt{Top}), (\texttt{[]}, \texttt{Bot}), (\texttt{[]}, \texttt{Top} : \texttt{Inf}),\\ &\quad (\texttt{Top} : zs, ys)\ \texttt{where}\ (zs, ys) = \texttt{App-Inf}\rangle\\ &\equiv \langle (\texttt{Bot}, \texttt{Top}), (\texttt{[]}, \texttt{Inf}), (\texttt{Top} : zs, ys)\ \texttt{where}\ (zs, ys) = \texttt{App-Inf}\rangle\end{aligned}$$

For $\texttt{append}\ xs\ ys \in \texttt{Botel}$ we obtain:

$$\begin{aligned}\texttt{App-Botel} &= \langle (\texttt{Bot}, \texttt{Top}), (\texttt{Bot} : \texttt{Top}, \texttt{Top}), (\texttt{[]}, \texttt{Bot}), (\texttt{[]}, \texttt{Bot} : \texttt{Top}),\\ &\quad (\texttt{[]}, \texttt{Top} : \texttt{Botel}), (\texttt{Top} : zs, ys)\ \texttt{where}\ (zs, ys) = \texttt{App-Botel}\rangle\\ &\equiv \langle (\texttt{Bot}, \texttt{Top}), (\texttt{Bot} : \texttt{Top}, \texttt{Top}), (\texttt{[]}, \texttt{Botel}),\\ &\quad (\texttt{Top} : zs, ys)\ \texttt{where}\ (zs, ys) = \texttt{App-Botel}\rangle\end{aligned}$$

These results describe sets of terms in the language analysed. Intuitively, in the results above we have values constructed as the union of pairs of values. The pairs denote a notion of *joint* membership, i.e. $(\texttt{[]}, \texttt{Bot})$ can be read as the pair of values where the first component can be represented by `[]` *and* the second by `Bot`.

The report is structured as follows: section 3 defines Λ_c a simple functional core language without any syntactic sugar to which a lazy functional language with a more comfortable syntax can easily be translated. We define reduction in Λ_c thereby giving it an operational semantics. Furthermore, we describe how to represent algebraic types in Λ_c. In section 4 we define contexts, a notion fundamental to this work. Intuitively, a context represents a set of concrete values. We derive important properties of contexts including the decidability of the question whether a context represents the empty set. Section 5.1 defines tableaux and rules with which we can determine for which arguments a given application is represented in a given context. Section 6 addresses results and limitations of our calculus. Section 7 points out the differences between contexts and evaluators and section 8.1 remarks on how to use the results of the analysis for termination analysis. Section 9 closes this report with a conclusion and an outlook to future extensions of this work.

2 Preliminaries

Vector notation, $\bar{x}$, is used for a finite sequence, $x_1, \ldots, x_n$, where the indices are obvious from the context. The notation of rules is in Gentzen-style: above the line appears the label of the leaf and below the line appear the labels of the leaves to be added separated by "|".

3 The Language

We define Λ_c, a simple functional core language, which we shall use as a target language for our analysis. It closely resembles the language used in [PJL91] and is also the target language for the strictness analysis based on abstract reduction [Sch95]. It has no local definitions and case expressions must mention an alternative for every constructor in the program.

3.1 Types

The type system of Λ_c consists of a set of (possibly user defined) type constructor names, each corresponding to a set of (data) constructors of fixed arity. For every type constructor name or type for short, A, there is a constant $\mathtt{case}_A$. The constructors of type A will be written $\mathtt{c}_{A,i}$, $1 \leq i \leq |A|$. In the proofs we will assume that expressions are well-typed w.r.t. $WT(\cdot)$, a definition of well-typedness meeting a particular set of conditions. Such definitions will be called *admissible*, a notion to be defined in definition 2. We will provide reductions in the operational semantics of Λ_c that enable a dynamic type check consistent with $WT(\cdot)$.

3.2 Expressions and Supercombinator Definitions

The following constants are in Λ_c:

- Finitely many constructors $\mathtt{c}_{A,1}, \ldots, \mathtt{c}_{A,c_A}$, where c_A is the number of all constructors of type A, $|A|$, each taking $\mathit{arity}(\mathtt{c}_{A,i})$ arguments.

- a case constant $\texttt{case}_A$, which takes $|A|+1$ arguments. The first argument is the expression to be cased. The other arguments of $\texttt{case}_A$ are functions taking $arity(\texttt{c}_{A,i})$ arguments for $1 \leq i \leq |A|$.
- supercombinators that may be defined in a recursive way (see below). There is at least the special supercombinator `bot`.

An *expression* in Λ_c can be a constant `c`, a variable x, or an application $(s\ t)$. Application associates to the left, i.e. $e_1\ e_2\ e_3$ is equivalent to $(e_1\ e_2)\ e_3$. We define variables that occur in some expression as usual. The set of free variables occurring in an expression s is denoted by $\mathcal{FV}(s)$. We assume $\mathcal{FV}$ to be defined for other kinds of syntactic objects also. A *ground expression* in Λ_c is one without free variables.
A *program* consists of a set of supercombinator definitions of the form

$$sc\ x_1 \ldots x_n := e$$

where e is an expression that may contain the variables $x_1, \ldots, x_n$, but no other variables.
A supercombinator $\texttt{bot} := \texttt{bot}$ is defined in any program.
As a notational convention $C[t]$ will denote an expression which has at some position the subexpression t, i.e. $C[.]$ denotes an expression context with a hole.

3.3 Operational Semantics

The operational semantics consists of $\delta-$reductions.

$$sc\ t_1 \ldots t_n \ \rightarrow_\delta e[t_1/x_1, \ldots, t_n/x_n]$$

if $sc\ x_1 \ldots x_n := e$ is a supercombinator definition. For the $\texttt{case}_A$-constant there is a $\delta-$rule of the following form:

$$\texttt{case}_A\ (\texttt{c}\ t_1 \ldots t_m)\ e_1 \ldots e_n \rightarrow_\delta (e_j\ t_1 \ldots t_m)$$

if `c` is a constructor of type A and $m = arity(\texttt{c})$.
An expression is in *weak head normal form* (WHNF), if it is in one of the following forms:

- $\texttt{c}\ e_1 \ldots e_n$, where `c` is a constructor of arity n
- $\texttt{c}\ e_1 \ldots e_n$, where `c` is a supercombinator name, a constructor or the $\texttt{case}_A$ constant and n is less than the arity of `c`.

To differentiate we will call the first kind *SCWHNF* and the second *function WHNF*.
The relation $\stackrel{*}{\rightarrow}_\delta$ denotes the transitive closure of $\rightarrow_\delta$. We say that t has a WHNF if $t \stackrel{*}{\rightarrow}_\delta t_0$ and t_0 is in WHNF.

Lemma 1. *The reduction relation $\rightarrow_\delta$ is confluent on ground expressions.*

Every term in Λ_c has a normal-order reduction which either terminates with a WHNF or does not terminate.

Definition 2 Admissible. A predicate $WT(\cdot)$, for well-typed terms, is admissible if it meets the following conditions:

- $WT(e)$ implies $WT(e')$ if $e \to_\delta e'$
- $WT(\mathtt{case}_A\ (\mathtt{c}_{A'}\ t_1 \ldots t_n) \ldots)$ is false if $n > \mathit{arity}(\mathtt{c}_{A'})$ or $A \neq A'$
- $WT(\mathtt{case}_A\ t \ldots)$ is false if t is a function WHNF
- $WT(\mathtt{c}\ t_1 \ldots t_n)$ is false if $n > \mathit{arity}(\mathtt{c})$
- $WT(e)$ implies $WT(e')$ for every subexpression e' of e
- $WT(e)$ implies that an expression e' exists with $WT(e')$ and $WT(e\ e')$ if e has a function WHNF.

A term t for which $WT(t)$ holds for an admissible WT is *dynamically well-typed.*

For ground expressions t we can define redexes as subexpressions where a reduction rule can be applied. The normal order redex can be found by shifting an evaluation label, ^{E}, into an expression. Initially the label is at the root.

$$\begin{array}{lcl} C[(\mathtt{case}_A\ e\ t_1 \ldots t_m)^E] & \to & C[(\mathtt{case}_A\ e^E\ t_1 \ldots t_m)] \\ & & \text{if the } \mathtt{case}_A\text{-expression is not a redex} \\ C[\mathtt{c}^E] & \to & C[\mathtt{c}] \\ C[(s\ t)^E] & \to & C[(s^E\ t)] \text{ if } (s\ t) \text{ is not a redex.} \end{array}$$

If in some ground expression t there is a final subexpression s with the label ^{E} then s is the normal order redex. An expression has no subexpression labeled ^{E} iff it is in WHNF.
In the case where t may have variables, the situation is slightly different. We call expressions of the form $(\mathtt{case}_A\ x\ t_1 \ldots t_{|A|})$ where x is a variable a *potential redex.* The normal order potential redex can be found by shifting an evaluation label, ^{E}, into an expression, almost as above. Again, the label is at the root initially.

$$\begin{array}{lcl} C[(\mathtt{case}_A\ e\ t_1 \ldots t_m)^E] & \to & C[(\mathtt{case}_A\ e^E\ t_1 \ldots t_m)] \text{ if the } \mathtt{case}_A\text{-expression} \\ & & \text{is neither a redex nor a potential redex} \\ C[\mathtt{c}^E] & \to & C[\mathtt{c}] \\ C[(s\ t)^E] & \to & C[(s^E\ t)] \text{ if } (s\ t) \text{ is not a redex.} \end{array}$$

Now the label ^{E} may be at a subexpression that is the normal order redex, at a normal order potential redex, or it may be at a variable.

4 Contexts

Contexts are a finite representation of some possibly infinite sets of expressions. They are used to stand for the values that are expected of a function application and are able to represent values that have no normal form as well as values that have no normal form, but can be approximated by an infinite sequence of normal forms having an increasing number of constructors.

Definition 3. Syntactically, *context terms* can be formed as follows:

- the constants `Bot` and `Fun`
- a context name
- a constructor expression $\mathtt{c}\ C_1 \ldots C_n$ for every constructor `c` of arity n in Λ_c, where the C_i are context terms
- a `where`-expression S `where` $T = N$ where S is a context term allowed to contain free variables, called *context variables*, T is a *context pattern* allowed to contain variables, constructors and `bot` and in which every variable occurs only once, $\mathcal{FV}(S) = \mathcal{FV}(T)$, and N is a context name
- a union expression $\langle C_1, \ldots, C_n \rangle$ for context terms $C_1, \ldots, C_n$
- an intersection expression $C_1 \cap \ldots \cap C_n$ for context terms $C_1, \ldots, C_n$

A *ground context term* is one that does not contain context names. A *context definition* has the form $A = C$ where A is a context name and C is a context term. Each context name must appear exactly once in a left hand side of a definition. Substitutions applied to context terms substitute ground context terms for context variables.
Usually we assume that there is a fixed set of global context definitions and denote this set by $\mathcal{C}$.

Note that the $\cap$s in the definition above are syntactic symbols only. Intersection and union are assumed to be associative, commutative and idempotent operations, i.e. we regard $t_1 \cap t_2 \cap t_3$, $t_3 \cap (t_1 \cap t_2)$, $t_2 \cap t_1 \cap t_3 \cap t_3$ and the like as *syntactically* equivalent (the same holds for unions).
We call the language of context terms Λ_c^C. Contrary to some other work on strictness analysis [Nöc93, Bur91], our contexts are able to represent sets without $\perp$.
Next we define the depth of context terms.

Definition 4.

$$\begin{array}{lcl} \mathit{depth}(A) & = & 1, \text{ if } A \text{ is a context name, } \mathtt{Bot} \text{ or } \mathtt{Fun} \\ \mathit{depth}(\mathtt{c}\ C_1 \ldots C_n) & = & 1 + max_i\, \mathit{depth}(C_i) \\ \mathit{depth}(\langle C_1, \ldots, C_n \rangle) & = & 1 + max_i\, \mathit{depth}(C_i) \\ \mathit{depth}(C_1 \cap \ldots \cap C_n) & = & 1 + max_i\, \mathit{depth}(C_i) \end{array}$$

A context represents a set of expressions from Λ_c: it *abstracts* expressions from Λ_c.

Example 2. We will give examples:

`Bot` represents the set of terms without WHNF

`Top` represents the set of all terms

`Fun` represents the set of terms with function WHNF

`Inf` $= \langle$`Bot, Top : Inf`$\rangle$ represents the set of terms approximated by finite lists whose tail has no WHNF

$\texttt{Fin} = \langle \texttt{Nil}, \texttt{Top} : \texttt{Fin} \rangle$ represents the set of all finite lists

The following definition prepares the definition of concretization. The primitive terms represented by a context have a particularly simple structure: they are terms without a WHNF, with a function WHNF or built from a constructor with arguments of the same simple structure.

Definition 5 Primitive terms represented. The set of primitive terms represented by the context term M, $\Lambda_c^{pr}(M)$ is the smallest set defined by:

$$\begin{aligned}
\Lambda_c^{pr}(C) &= \bigcup_{i,m} \Lambda_{c,i}^{pr,m}(C) \\
\Lambda_{c,n}^{pr,m}(\texttt{Bot}) &= \{t | t \text{ has no WHNF}\} \\
\Lambda_{c,n}^{pr,m}(\texttt{Fun}) &= \{t | t \text{ has a function WHNF}\} \\
\Lambda_{c,n}^{pr,m}(\langle C_1, \ldots, C_m \rangle) &= \bigcup_{i} \Lambda_{c,n}^{pr,m}(C_i) \\
\Lambda_{c,n}^{pr,m}(C_1 \cap \ldots \cap C_m) &= \bigcap_{i} \Lambda_{c,n}^{pr,m}(C_i) \\
\Lambda_{c,n}^{pr,m}(S \texttt{ where } T = N) &= \bigcup_{\sigma : \Lambda_{c,n-1}^{pr,m}(\sigma T) \subseteq \Lambda_{c,n-1}^{pr,m}(N)} \Lambda_{c,n-1}^{pr,m}(\sigma S), n > 0 \\
\Lambda_{c,n}^{pr,m}(N) &= \begin{cases} \Lambda_{c,n}^{pr,m-1}(C), & \text{if } (N = C) \in \mathcal{C}, m > 1 \\ \emptyset, & \text{if } (N = C) \in \mathcal{C}, m = 0 \end{cases} \\
\Lambda_{c,n}^{pr,m}(\texttt{c}\ C_1 \ldots C_r) &= \{\texttt{c}\ t_1 \ldots t_r | t_i \in \Lambda_{c,n}^{pr,m}(C_i)\}
\end{aligned}$$

Definition 6 less defined. For $s, t \in \Lambda_c$ we define s to be less defined than t, $s \leq_c t$, iff in all contexts $C[\cdot]$ if $C[s]$ has SCWHNF $\texttt{c}\ s_1 \ldots s_n$ then $C[t]$ has SCWHNF $\texttt{c}\ t_1 \ldots t_n$.

Definition 7 strict least upper bound. Using the known notion of least upper bound, $\bigsqcup$, w.r.t. $\leq_c$ we define a stricter notion, $\bigsqcup^s$. s is the strict least upper bound of a chain $s_1 \leq_c s_2 \leq_c \ldots$, $s = \bigsqcup_i^s s_i$ iff $s = \bigsqcup_i s_i$ and if for any context $C[\cdot]$, $C[s]$ has a SCWHNF $\texttt{c}\ t_1 \ldots t_n$ then so does $C[s_i]$ for at least one s_i.

Evidently, a strict least upper bound need not exist.
Obviously, all terms are monotonous wrt. $\leq_c$. Furthermore, all terms g are continuous wrt. $\bigsqcup^s$, i.e. $a_1 \leq_c a_2 \leq_c \ldots, a = \bigsqcup_i^s a_i$ implies $\bigsqcup_i^s g\ a_i = g\ (\bigsqcup_i^s a_i) = g\ a$ which is not hard to prove.

Definition 8 Concretization. The concretization of a context term is the set of well-typed terms which are strict least upper bounds of chains of primitive terms represented by the context term.

$$\gamma(C) = \{t | t = \bigsqcup_i^s s_i, s_1 \leq_c s_2 \ldots, s_i \in \Lambda_c^{pr}(C), t \text{ is well-typed}\}$$

For the loop detection rules 2a and 2b of the calculus we will need to decide for a context term D if $\gamma(\texttt{Bot}) \subseteq \gamma(D)$, written $\texttt{Bot} \leq D$. These context terms will be restricted in the sense that they do not contain `where`-expressions, neither directly nor by reference to other context definitions.

Definition 9. We define

$$\begin{array}{ll}
\texttt{Bot} \leq \texttt{Bot} & = \text{ True} \\
\texttt{Bot} \leq \texttt{Fun} & = \text{ False} \\
\texttt{Bot} \leq \texttt{c}_A \ \ldots & = \text{ False} \\
\texttt{Bot} \leq \langle C_1, \ldots, C_n \rangle & = \text{ True, if } \texttt{Bot} \leq C_i \text{ for } \textit{some } i \\
\texttt{Bot} \leq C_1 \cap \ldots \cap C_n & = \text{ True, if } \texttt{Bot} \leq C_i \text{ for } \textit{all } i \\
\texttt{Bot} \leq N & = \texttt{Bot} \leq C \text{ if } (N = C) \in \mathcal{C}
\end{array}$$

The above definition of $\texttt{Bot} \leq D$ is easily seen to be decidable, since there are only finitely many context definitions in $\mathcal{C}$. An algorithm could keep track of the ones already used and abandon paths in which a context name is used more than once.
One of the loop detection rules (2c) uses the notion of `Bot`-closedness. It is defined as follows:

Definition 10. Let D be a context term not using `where`-expressions, neither directly nor via context definitions. D is `Bot`-*closed* iff $C[t] \in \Lambda_c^{pr}(D)$ implies that for all $t' \in \gamma(\texttt{Bot}) : C[t'] \in \Lambda_c^{pr}(D)$.

It is not obvious that this property is decidable. The proof of decidability will be the subject of future work.

5 The Calculus

5.1 Tableaux

Given a supercombinator f and an abstract value A we want to infer an abstract value C such that for elements e in the concretization of C the application $(f\ e)$ is in the concretization of A. We permit additional constraints on the variables appearing as arguments in the application. These constraints are of the form $x \in A$, where x is a variable and A is a context. It is no restriction to assume that variables without such an additional constraint are constrained by `Top`.
A substitution σ is a mapping $\sigma : \Lambda_c \to \Lambda_c$ that respects the structure of expressions. It is usually represented as a set of variable-expression pairs, denoted as $\{x_i \mapsto t_i, \ldots\}$. The application of a substitution σ to an expression t is written $\sigma(t)$, the composition of two substitutions σ, τ is written $\sigma \circ \tau$ such that $\sigma \circ \tau(t) = \sigma(\tau(t))$.

Definition 11 Tableau. A *tableau* $\mathcal{T}$ is a finite tree whose nodes are labeled with expressions of the form:

$$rs, \sigma$$

where rs is a (possibly empty) set of constraints and σ is an idempotent substitution where the substituted expressions are constructed from variables, constructors, and `bot`. A constraint is of the form

$$s \in A, \text{ with } s \in \Lambda_c, A \in \Lambda_c^C$$

The set of constraints in a label is also written $s \in A, R$, if we want to speak about a particular constraint, $s \in A$, and possibly others, R. We assume that

for every constraint $s \in A$, in a label $s \in A, R, \sigma$, we have $\sigma(s) = s$. We will refer to rs as the set of *constraints*. If the free variables of two constraints form disjoint sets, the constraints are said to be *independent*. The elements of the set of constraints where s is a variable form the *context constraint for variables*. Within a constraint r we call the left-hand side variables the variables *constrained* by r. The root of a tableau is labeled with a set of constraints $t \in A$ and the identity substitution. In general the label of the root is one non-trivial constraint with a well-typed left-hand side and a context term not using `where`-expressions, neither directly nor via context definitions, together with a CCV. We have to compute instances for the set of initial variables $\mathcal{FV}(t)$. Hence a fixed order of these initial variables is assumed and the tuple of these variables is denoted $W_{\mathcal{T}}$, or simply W if no confusion arises.
Leaves may have an additional label *no!* or *loop!* followed by a `where`-expression.

Definition 12. A ground substitution θ is a *solution* for (rs, σ) where $rs = \{s_1 \in C_1, \ldots, s_n \in C_n\}$ iff for all i we have $\theta(s_i) \in \gamma(C_i)$ and for all variables x_j occurring in rs we have $\theta(x_j) = \theta(\sigma(x_j))$. The set of solutions for rs, σ is written $U(rs, \sigma)$.
A leaf labeled *no!* has no solution.
The *size* of a solution is the sum of the sizes of the terms it substitutes for the variables x_i.

Let R, S be constraints. Then we say R implies S, iff for all ground substitutions θ, if θ is a solution of R, then θ is also a solution of S. $U(R, \{\}) \subseteq U(S, \{\})$.
A tableau is a representation of the information it contains suitable for application of the rules we give in the subsequent sections. The final result, however, is better represented as a context, such that it can easily be used in constraints.

Definition 13. A tableau is *closed* if each leaf

- is labeled *no!* or
- is labeled *loop!* together with a `where`-expression and a CCV or
- has a label in which all the constraints are CCVs. We also call such a leaf *solved*.

In order to obtain a context definition from a closed tableau we proceed as follows. Since the tableau represents a joint evaluation context for the free variables of the root, we give the context a unique name. In the right hand side of the context definition we collect the information from the leaves not labeled *no!* in a union. This is straightforward for variables appearing either in a substitution or in a CCV. If a variable appears in both, we use the intersection of the value substituted for it and the constraint from the CCV.

Definition 14. A tableau is *sound* iff for all interior nodes N with children $N_1, \ldots, N_m$:

$$U(N_i) \subseteq U(N)$$

Definition 15. A tableau is *complete* iff for all interior nodes N with children $N_1, \ldots, N_m$:

$$U(N) \subseteq \bigcup_{i=1}^{m} U(N_i)$$

5.2 Rules

To develop a calculus which computes closed tableaux by stepwise derivation of new sound tableaux with the same root node from a sound tableau, we describe the expansion rules for tableaux in the following subsections. Expansion rules may only extend a path in the original tableau by attaching a new leaf. An expansion rule is called *sound* if it transforms sound tableaux into sound tableaux and *complete* if it transforms complete tableaux into complete tableaux. The left-hand sides are from Λ_c. The rules that correspond to reduction rules on Λ_c are assumed to obey normal order. We use the term-context notation $C[\cdot]$ for Λ_c-terms.
The rules assume the calculus is started with a tableau consisting of a root node only labeled with a single constraint that has a well-typed left hand side. Some of the rules given below are not obviously decidable, e.g. the rule for equivalent contexts on page 14 uses *equivalence*. We will not address these decidability issues in this report, but assume effective and sound (perhaps incomplete) algorithms to compute certain relationships on contexts, e.g. $\gamma(C) = \emptyset$, $\gamma(C) \cap \gamma(D) = \emptyset$ etc. These issues will be dealt with in a separate work.

Rules for `Bot`

$$\frac{\texttt{bot} \in \texttt{Bot}, R, \sigma}{R, \sigma}$$

A leaf labeled $\texttt{bot} \in A, R, \sigma$ receives the additional label *no!* if $\texttt{Bot} \leq A$ is false.

Unions

A context that is a union can be decomposed.

$$\frac{s \in \langle C_1, \ldots, C_m \rangle, R, \sigma}{s \in C_1, R, \sigma | \ldots | s \in C_m, R, \sigma}$$

Intersections

We can use intersections to replace multiple constraints on the same expression.

$$\frac{t \in C, t \in D, R, \sigma}{t \in C \cap D, R}$$

Also, an intersection can be split into multiple constraints.

$$\frac{t \in C \cap D, R}{t \in C, t \in D, R, \sigma}$$

Since the second intersection rule reverses the effect of the first an implementation of the calculus will have to use appropriate heuristics not to get into an infinite loop.

δ and $\texttt{case}_A$-rules

The simplest extension of tableaux is by reduction on a leaf. The term side in a constraint is a Λ_c expression. We can use δ-reduction as defined for Λ_c.

$$\frac{t \in a, R, \sigma}{t' \in a, R, \sigma}, \text{ if } t \to t' \text{ is a normal order reduction.}$$

Type Error

A leaf for which $WT(\cdot)$ does not hold is labeled with the additional label *no!*.

Constructors

$$\frac{\texttt{c}\ t_1 \dots t_n \in \texttt{c}\ C_1 \dots C_n, R, \sigma}{t_1 \in C_1, \dots, t_n \in C_n, R, \sigma}$$

The following rule identifies leaves in the tableau which cannot represent any solutions.

> A leaf labeled $\texttt{c}\ s_1 \dots s_m \in A, R, \sigma$, where $A \cap \texttt{c Top} \dots \texttt{Top}$ is inconsistent will receive the additional label *no!*.

Case

We give the rule for a $\texttt{case}_A$-expression that is a potential redex.

> To extend a tableau with a leaf L labeled
>
> $$D[\texttt{case}_A\ x\ e_1 \dots e_{|A|}] \in a, R, \sigma$$
>
> a leaf for each of the labels
>
> $$\begin{array}{l} \{x \mapsto \texttt{bot}\}(D[\texttt{case}_A\ \texttt{bot}\ e_1 \dots e_{|A|}] \in a, R), x \in \texttt{Bot}, \sigma, \\ \rho_1(D[\texttt{case}_A\ x\ e_1 \dots e_n] \in a, R), \rho_1 \circ \sigma, \\ \qquad\qquad \vdots \\ \rho_{|A|}(D[\texttt{case}_A\ x\ e_1 \dots e_n] \in a, R), \rho_{|A|} \circ \sigma \end{array}$$
>
> can be attached to d where $\rho_i = \{x \mapsto (\texttt{c}_{A,i}\ x_{i,1} \dots x_{i,k_i})\}$ and the $x_{i,j}$ are new variables.
>
> Subsequently, these labels will be δ-reduced according to paragraph 5.2.

It is not necessary to attach a label for the case that the first argument of the $\texttt{case}_A$ may have a function as its WHNF since then the $\texttt{case}_A$ would not be well-typed.
For a $\texttt{case}_A$-expression that is not a potential redex, we can apply the following rule.

> To extend a tableau with a leaf L labeled
>
> $$D[\texttt{case}_A\ t\ e_1 \dots e_{|A|}] \in a, R, \sigma$$

a leaf for each of the labels

$$D[\mathtt{case}_A\ \mathtt{bot}\ e_1 \dots e_{|A|}] \in a, t \in \mathtt{Bot}, R, \sigma,$$
$$D[\mathtt{case}_A\ (\mathtt{c}_{A,1}\ x_{1,1} \dots x_{1,k_1})\ e_1 \dots e_n] \in a, t \in \mathtt{c}_1\ \mathtt{Top} \dots \mathtt{Top}, R, \sigma,$$
$$\vdots$$
$$D[\mathtt{case}_A\ (\mathtt{c}_{A,|A|}\ x_{|A|,1} \dots x_{|A|,k_{|A|}})\ e_1 \dots e_n] \in a, t \in \mathtt{c}_{A,|A|}\ \mathtt{Top} \dots \mathtt{Top}, R, \sigma$$

can be attached to d in which the $x_{i,j}$ are new variables, $k_i = arity(\mathtt{c}_{A,i})$.

Rules for equivalent contexts

The following rule summarizes all semantically correct transformations on the contexts in a tableau, for example shifting the union to the top, or expanding a context name using the global definition.

$$\frac{s \in C, R, \sigma}{s \in C', R, \sigma} \text{ if } \gamma(C) = \gamma(C')$$

Rules for independent constraints

If a constraint is independent of the other constraints in a leaf its solution may be computed independently.

> Let $t \in A, rs, \sigma$ be the label at a leaf where $t \in A$ is independent from the constraints in rs. Furthermore, assume we have a solution $S = T$ where S is the name and T the context term for the solution for the tableau with root $t \in A$ and that $T \neq \langle\rangle$. Then we can add a leaf labeled $rs, (x_1, \dots, x_n) \in S, \sigma$ where $(x_1, \dots, x_n)$ are the free variables in t. If $T = \langle\rangle$ we can add to the leaf the label *no!*.

Rules for loop-detection

The rules for detecting loops make the calculus a powerful tool. For loop detection we define *size* on expressions that are constructed from variables, constructors and `bot` as follows:

$$\begin{aligned} size(\mathtt{bot}) &= 1 \\ size(\mathtt{c}\ t_1 \dots t_n) &= 1 + \sum_i size(t_i) \\ size(x) &= 0, \text{ if } x \text{ is a variable} \end{aligned}$$

The *size* of a substitution is the sum of the sizes of the substituted terms.
In the following rules let $B[\cdot]$ and $C[\cdot]$ be n-ary and r-ary term contexts, respectively.
Let $B[x_1, \dots, x_n] \in E, S, \emptyset$ be the label at the root, $C[x_1, \dots, x_r] \in D, S, \emptyset$ be a label above the first branch and let the leaf be labeled $C[t_1^1, \dots, t_r^1] \in D, \dots, C[t_1^m, \dots, t_r^m] \in D, R, \sigma$. Let R and S be CCVs.
We consider four different rules for detecting loops:

1. *loop-size*

   Assume the following conditions are satisfied:

- The constraint R implies the constraints $\{x_i \mapsto t_i^j \mid i = 1, \ldots, r\}(S)$ for all j.
- the expressions t_i^j consist only of variables, constructors and `bot`.
- for all j: $\sum_i size(\sigma x_i) > \sum_i size(t_i^j)$
- For all j and every variable $y \in \mathcal{FV}(t_1^j, \ldots, t_r^j)$, the number of occurrences in $\sigma x_1, \ldots, \sigma x_r$ is not less than the number of occurrences in $t_1^j, \ldots, t_r^j$.
- $C[t_1^i, \ldots, t_r^i] \in D$ and $C[t_1^j, \ldots, t_r^j] \in D$ are independent if $i \neq j$.

Then we can add a leaf labeled *loop!* together with the `where`-expression $(\sigma x_1, \ldots, \sigma x_r, \underbrace{\texttt{Top}, \ldots, \texttt{Top}}_{n-r})$ `where` $((t_1^1, \ldots, t_1^m), \ldots, (t_r^1, \ldots, t_r^m)) = \underbrace{(N, \ldots, N)}_{m}$ where $t_i^j = y_i^j$ for new variables y_i^j, $r+1 \leq i \leq n$, N is the name of the context definition for the solution at the root and R as the CCV.

2. *loop-decomp*

   Assume that for all i, j $\sigma x_i = t_i^j$ and that R implies the constraint $\sigma(S)$. Then there are three cases:

   (a) If `Bot` $\leq D$ and there is no decomposition of a constructor on top level, but at least one abstract reduction on the path from the root to the leaf. Then we can remove the constraints $C[t_1^1, \ldots, t_r^1] \in D, \ldots, C[t_1^m, \ldots, t_r^m] \in D$ from the leaf and keep only R, σ.

   (b) If `Bot` $\not\leq D$ and there is no decomposition of a constructor on top level, but at least one abstract reduction on the path from the root to the leaf. Then then we can add the label *no!* to the leaf.

   (c) If the context D is `Bot`-closed and if there is a constructor decomposition on the path from the root to the leaf, then we can remove the constraints $C[t_1^1, \ldots, t_r^1] \in D, \ldots, C[t_1^m, \ldots, t_r^m] \in D$ and keep only R, σ.

Example 3. The condition that the context is `Bot`-closed is necessary in rule 2c. Consider `rep` $1 \in$ `Fin`. The rules show that if we do not require the context to be `Bot`-closed in rule 2c, then the tableau can be closed, and would represent a solution to `rep` $1 \in$ `Fin`. However, such a solution can not exist.

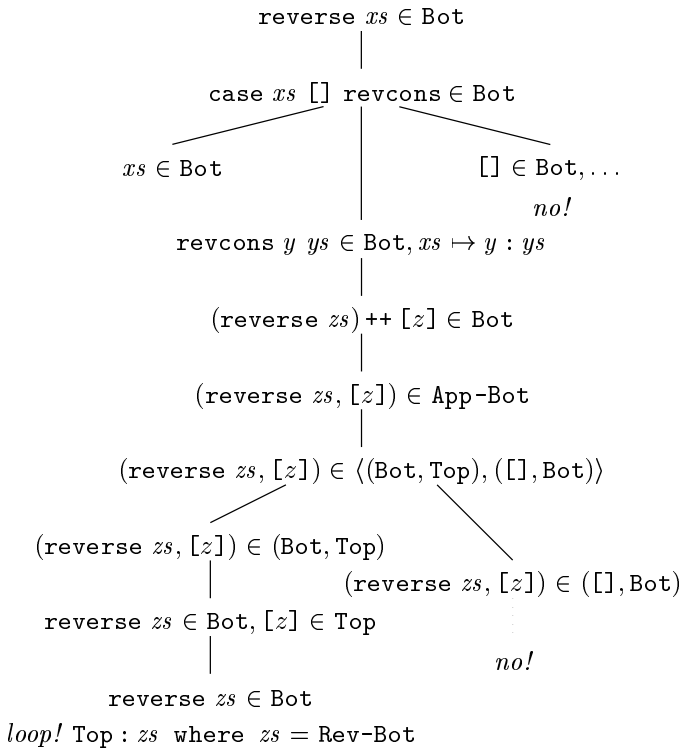
6 Results and Limitations

We implemented our calculus as a prototype to get an idea of its applicability in practice. The timings we did are more than encouraging. On an HP 715/50 analyzing applications of `append` and `zip` in the contexts `Inf`, `Bot`, and `Botelem` and `concat` in the context `Bot` takes less than 0.3 seconds. The implementation was compiled using hbc 0.9999.3 *without* optimization. We refrain from testing the prototype on functions using arithmetic and numbers because on the one hand we did not implement any primitive functions or data types on the other

hand simulating the behavior of the operations and comparisons on integers using Peano numbers gives analysis results that are very hard to grasp.
An input on which our calculus fails to produce the desired result at first is $\texttt{reverse}\ xs \in \texttt{Bot}$ where `reverse` is defined as follows:

$$\begin{array}{lcl} \texttt{reverse}\ xs & = & \texttt{case}_{List}\ xs\ \texttt{[]}\ \texttt{revcons} \\ \texttt{revcons}\ z\ zs & = & \texttt{append}\ (\texttt{reverse}\ zs)\ \texttt{[}z\texttt{]} \end{array}$$

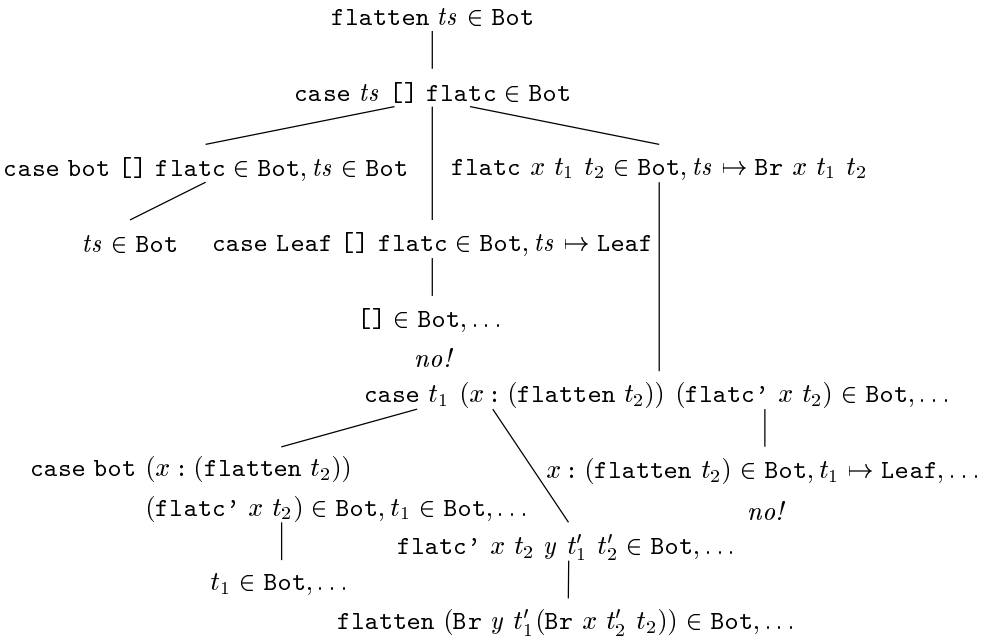
In the tableau for the above input the redex $\texttt{reverse}\ xs$ will repeatedly appear, but each time one level deeper inside the nested context introduced by reduction of the `append`. Thus the loop-detection rules above cannot be applied. If we use the rule for independent contexts on the other hand and use $\texttt{App-Bot} = \langle(\texttt{Bot}, \texttt{Top}), (\texttt{[]}, \texttt{Bot})\rangle$, the result computed for $xs \texttt{++} ys \in \texttt{Bot}$, we get the following tableau:



Let a data-type for binary trees be given that stores the data as the first argument of 3-ary branch nodes constructed by `Br` and has nullary leaves to terminate paths constructed by `Leaf`. The function `flatten` maps such trees to the list of the inorder collected data they contain. It can be defined in several ways. Below, we will first present a definition for which our calculus fails, followed by a definition for which it succeeds and computes the desired result.

$$\begin{array}{lcl} \texttt{flatten}\ ts & = & \texttt{case}_{Tree}\ ts\ \texttt{[]}\ \texttt{flatc} \\ \texttt{flatc}\ x\ t_1\ t_2 & = & \texttt{case}_{Tree}\ t_1\ (x : (\texttt{flatten}\ t_2))\ (\texttt{flatc'}\ x\ t_2) \\ \texttt{flatc'}\ x\ t_2\ y\ t_1'\ t_2' & = & \texttt{flatten}\ (\texttt{Br}\ y\ t_1'\ (\texttt{Br}\ x\ t_2'\ t_2)) \end{array}$$

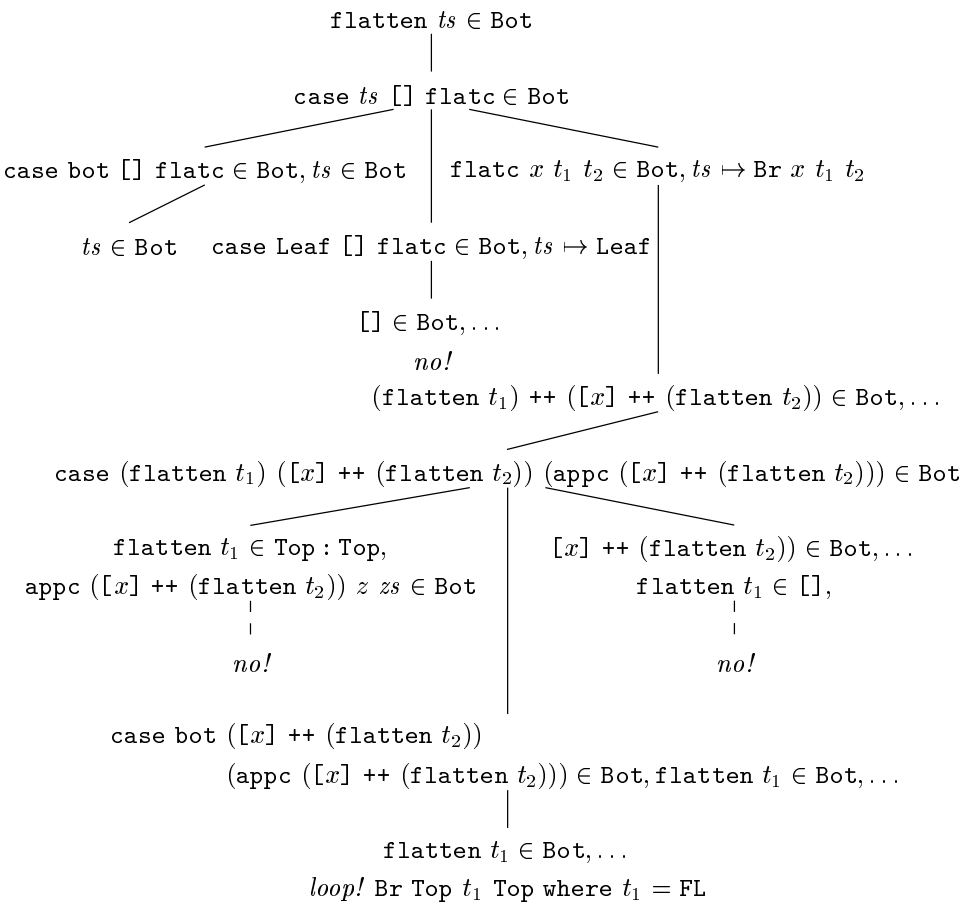
In the following tableau we have dropped the index of the $\texttt{case}_A$-constants only to make the tableaux fit on the page.



None of our loop detection rules can be applied: for *loop-size* the condition $size(\sigma x_i) > size(t_i)$ is violated and for *loop-decomp* $\sigma x_i = t_i$ is violated.
On the other hand, if `flatten` is defined differently then it can be analysed.

$$\begin{array}{lcl} \texttt{flatten}\ ts & = & \texttt{case}_{Tree}\ ts\ \texttt{[]}\ \texttt{flatc} \\ \texttt{flatc}\ x\ t_1\ t_2 & = & (\texttt{flatten}\ t_1)\ \texttt{++}\ (\texttt{[}x\texttt{]}\ \texttt{++}\ (\texttt{flatten}\ t_2)) \end{array}$$

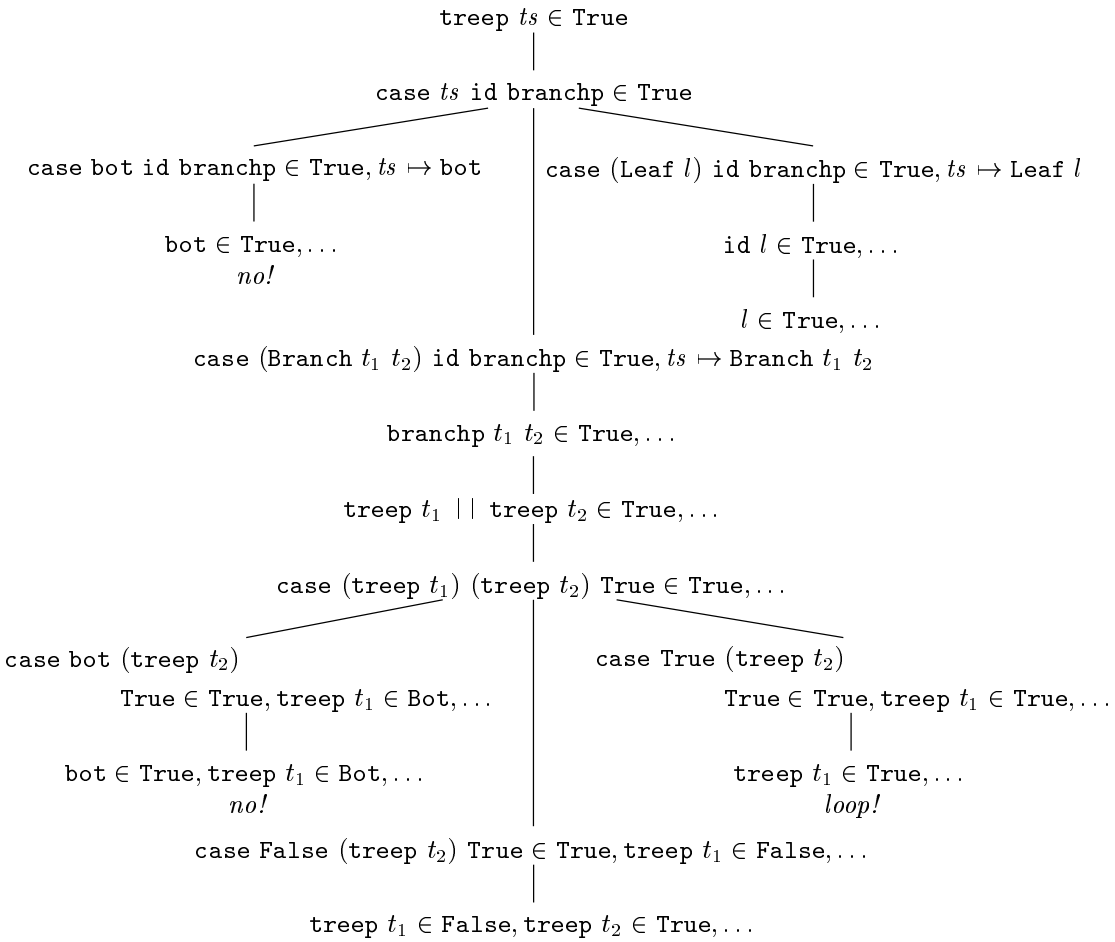
The analysis of `flatten` $xs \in$ `Bot` then yields the following tableau and yields a result which we name `FL`.



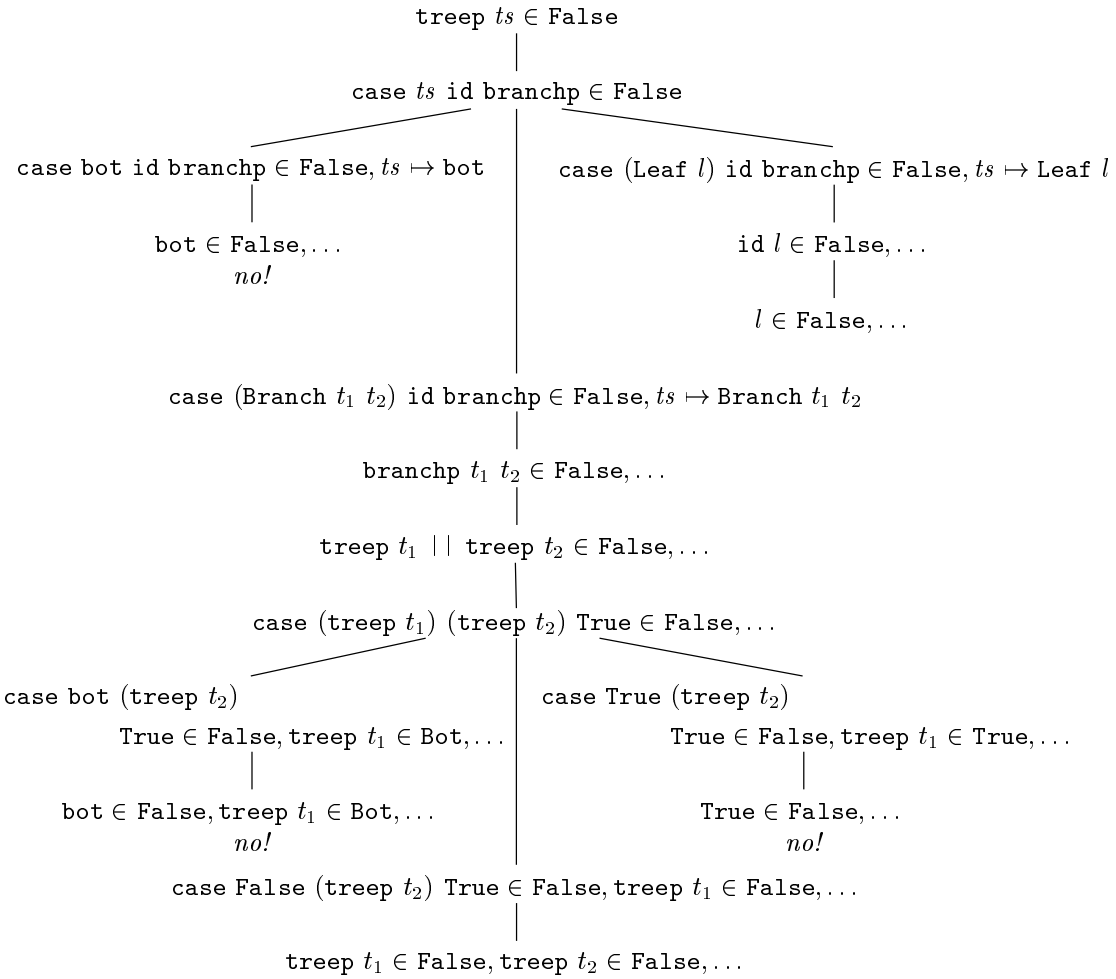
The two labels *no!* that are attached with dashed lines will be produced independently of the choice of the constraint to apply rules to in the nodes above them. The result of this analysis is FL = ⟨Bot, BR Top t_1 Top where t_1 = FL. The calculus is able to analyse both flatten $ts \in$ [] and flatten $ts \in$ Top : Top. The results are FL-NIL = Leaf and FL-CONS = ⟨Br Top FL-CONS Top, Br Top FL-NIL Top⟩ respectively.
As another example consider a predicate treep on trees which maps to True the trees containing at least one Leaf True. In this example trees store data in the leaves.

$$\begin{array}{lcl} \texttt{treep}\ ts & = & \texttt{case}_{Tree}\ ts\ \texttt{id branchp} \\ \texttt{branchp}\ t_1\ t_2 & = & \texttt{treep}\ t_1\ ||\ \texttt{treep}\ t_2 \\ a\ ||\ b & = & \texttt{case}_{Bool}\ a\ b\ \texttt{True} \end{array}$$

Now the calculus can be used to obtain a representation of all the trees which treep maps to True.

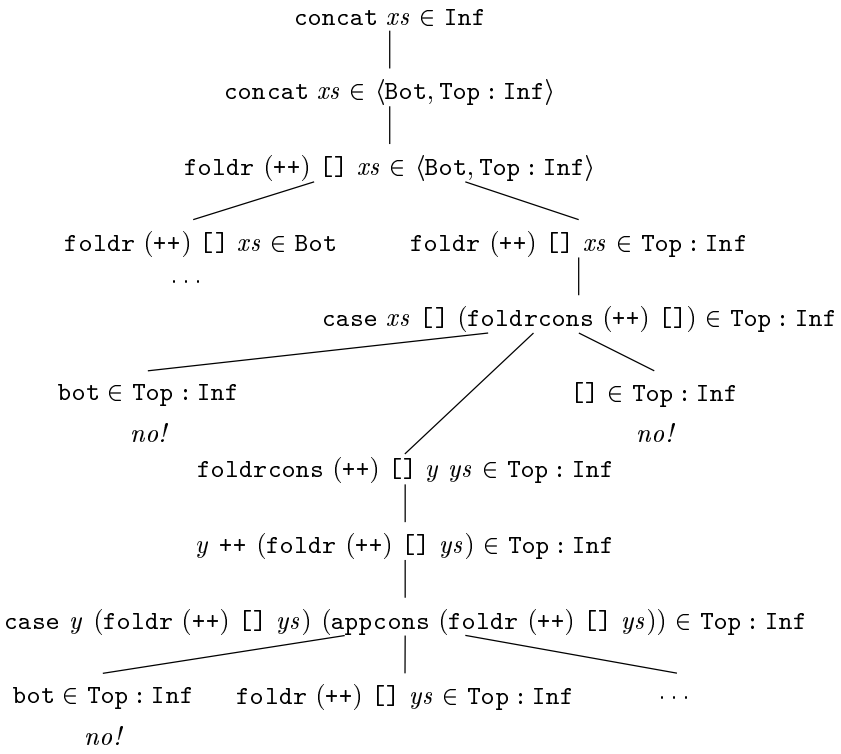


This analysis can be completed using the result of another analysis treep $ts \in$ False.



The results are `TR-F` $=$ $\langle$`Leaf False, Br TR-F TR-F`$\rangle$ and `TR-T` $=$ $\langle$`Leaf True, Br TR-F TR-T`$\rangle$.
For `concat` defined in the following way the analysis fails.

```
concat              =  foldr (++) []
foldr f n xs        =  case xs n (foldrcons f n)
foldrcons f n y ys  =  f y (foldr f n ys)
```



At the leaf labeled `foldr (++) []` $ys \in$ `Top : Inf` we cannot apply a loop detection rule, although there is a node in the tableau labeled exactly like the leaf. One possibility here is to not apply the union deconstruction rule as long as other rules may be applied. While this has an advantage in the case above, it could have a disadvantage in other cases. This issue is currently being investigated.

7 Contexts and Evaluators

Burn [Bur91] defines evaluation transformers to be mappings from the evaluator for an application to the evaluator for its arguments where the evaluators are reduction strategies. The intuitive view is that every unary supercombinator S can be an evaluator, where the demand is that $S\ x$ should be evaluated to WHNF. Relating this to contexts, this means that the tableau starting with $S\ x \in$ `Bot` may compute a solution, which is a description of expressions that, instantiated for x, do not result in a WHNF, if S is applied to them. We have that application of S to expressions in the context yields `Bot`, whereas application of S to expressions not in the context yields a WHNF. This follows from our result above.

This representation of evaluators is also able to express head-strictness, a property that not all approaches to strictness and context analysis are able to express [BHA85, Bur89, HW87].

The context `head` defined as $\{$`head` $= \langle$`Bot, Bot : Top`$\rangle\}$ represents the head strict evaluator.

Paterson [Pat96] uses the notion of *latent demand* to stand for contexts in which an evaluation to WHNF is not necessary, but if such an evaluation will become

necessary then it is known that an even stronger demand on the application is given. In our framework this can be modeled by contexts in which there is no `Bot` on the top-level, e.g. `h` = `Bot : Top` stands for a context in which evaluation of the first list element is required as soon as it is known that the expression evaluates to a WHNF having `:` as its top most constructor. Latent and immediate demand can freely be mixed as in `g` = $\langle$`Bot`, `Top` : (`Bot` : `Top`)$\rangle$.

Example 4. Examples of contexts representing evaluators:

- `Bot` represents the evaluators that attempt to reduce the term to WHNF.
- `Top` represents the evaluators that do not attempt to evaluate the term at all.
- `Inf` defined as {`Inf` = $\langle$`Bot`, `cons Top Inf`$\rangle$} represents the evaluators that attempt to iteratively reduce the spine of the list-expression to WHNF.

It is not hard to see that such contexts corresponding to evaluators must be `Bot`-closed. Interestingly, not all `Bot`-closed contexts are evaluators, for example the context `Listpar` defined as `Listpar` = $\langle$`Bot`, `Bot : Bot`$\rangle$. The problem is that the context `Listpar` corresponds to a parallel evaluation of head and tail of a list and there is no sequentialisation of this evaluation that results in the context `Listpar`.
This shows that the analysis method is also able to work for parallel evaluation if the rules are adapted.

8 Applications

8.1 Termination Analysis

If we are interested in a description of instances for x for which the application $(S\ x)$ terminates if we assume lazy evaluation to WHNF, then we can use the following approach. Compute a closed tableau with root $(S\ x) \in$ `nonBot`, where we define `nonBot` = $\langle$`Fun`, $\mathtt{c}_1$ `Top` ... `Top`, ... , $\mathtt{c}_n$ `Top` ... `Top`$\rangle$ which represents all expressions having a WHNF and a context `nonBotelem` = $\langle$$\mathtt{c}_1$ `nonBotelem` ... `nonBotelem`, ... , $\mathtt{c}_n$ `nonBotelem` ... `nonBotelem`$\rangle$ which represents all expressions having a normal form. As another example consider `append` xs ys $\in$ `nonBotelem` for which we obtain the solution (xs, ys) = $\langle$(`[]`, `nonBotelem`), (`nonBotelem` : xs, ys)$\rangle$

8.2 n-Packs

The motivation for n-packs is that often evaluation of lists (or other structured data) could advantageously be performed for more than one element at a time. While this is obvious for spine-strict functions, it is equally true for many others. The n-pack approach [KPSSS] extends lists (and possibly other structured datatypes in the future) with an additional constructor that stores n elements per node. A source-to-source transformation using partial evaluation then tries to

propagate the use of this new constructor in the program. It is this transformation that can benefit from the results of context analysis to introduce the new constructor in cases where partial evaluation alone would not have been sufficient to do so. As an example consider $C[e_1 : \texttt{case}\ e_2 \ldots]$. For n-packing to proceed the : needs to be moved to the alternatives of the case. Can this be achieved without changing the semantics of the program? It can be done provided that if e_1 has no WHNF $C[e_1 : \texttt{case}\ e_2 \ldots]$ does not have a WHNF either.
Thus context analysis could be used to determine contexts for the variables in e_1 such that $e_1 \in \texttt{Bot}$ followed by an analysis (strictness or context) of $C[e_1 : \texttt{case}\ e_2 \ldots]$ with the additional constraints on the variables from the first analysis.

9 Conclusion and Future Work

In this paper we have presented a calculus for context analysis. We have shown that in contrast to other methods for context analysis our method tries to construct a representation of all values for which an application yields a particular value. It is this feature that makes the calculus valuable for verification in addition to being valuable for context and termination analysis.
Future work will extend the calculus to handle higher order functions. The rule *loop-size* above could be applied if its conditions are met for a definition of *size* satisfying $size(e) > 0$ for all e and $size(\texttt{c}\ t_1 \ldots t_n) = n_0 + \sum n_i size(t_i)$. Measures of this kind can be computed automatically using ordering tableaux [Pan96]. As an extension our calculus could be made to use such a suitable *size* function computed for the terms between which there is a potential loop.

Acknowledgments

Very special thanks go to John Launchbury for posing the initial question that triggered this work over lunch at the First International Spring-school on Advanced Functional Programming. Also, this work owes to Sven Eric Panitz. He lend us an ear or two whenever we felt a need to talk about this work and had helpful comments on this work at all its stages.

References

[Aik94] A. Aiken. Set constraints: Results, applications, and future directions. In *Second Workshop on the Principles and Practice of Constraint Programming*, pages 171–179, Orcas Island, Washington, May 1994.

[BHA85] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory for strictness analysis for higher order functions. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Structures*, number 217 in Lecture Notes in Computer Science, pages 42–62. Springer, 1985.

[Bur87] Geoffrey Burn. Evaluation transformers - a model for the parallel evaluation of functional languages (extended abstract). In Gilles

Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 446–470. Springer, 1987.

[Bur89] Geoffrey L. Burn. *Deriving a parallel evaluation model for lazy functional languages using abstract interpretation*, pages 111–160. J. Wiley and Sons, 1989.

[Bur91] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Pitman, London, 1991.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 252–252. ACM Press, 1977.

[DW90] Kei Davis and Philip Wadler. Strictness analysis in 4d. In *Glasgow functional programming workshop*, Aug 90.

[Far92] John William Farrell. *Context Approximation of Functional Languages*. PhD thesis, University of Queensland, 1992.

[FB94] Sigbjørn Finne and Geoffrey Burn. Assessing the evaluation transformer model of reduction on the Spineless G-machine. Technical report, Imperial College of Science, Technology and Medicine, Department of Computing, 1994.

[Hal94] Cordelia V. Hall. Using hindley milner type inference to optimise list representation. In *Conference on Lisp and Functional programming*, 1994.

[HB93] Denis B Howe and Geoffrey L Burn. Using strictness in the STG machine. In John T O'Donnel and Kevin Hammond, editors, *Functional Programming*, Workshops in Computingt, pages 127–137. Springer, 1993.

[HJ94] Nevin Heintze and Joxan Jaffar. Set constraints and set-based analysis. In *Second Workshop on the Principles and Practice of Constraint Programming*, LNCS, pages 281–298, Orcas Island, Washington, May 1994. Springer.

[HW87] John Hughes and Philip Wadler. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 385–407. Springer, 1987.

[KPSSS] Norbert Klose, Sven Eric Panitz, Manfred Schmidt-Schauß, and Marko Schütz. Personal comminication.

[LB96] John Launchbury and Gebreselassie Baraki. Representing demand by partial projections. *Journal of functional programming*, 6:563–585, 1996.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *4th International Symposium on Programming*, number 83 in Lecture Notes in Computer Science, pages 269–281. Springer, 1980.

[Nöc92] Eric Nöcker. Strictness analysis by abstract reduction in orthogonal term rewriting systems. Technical Report 92-31, University of Nijmegen, Department of Computer Science, 1992.

[Nöc93] Eric Nöcker. Strictness analysis using abstract reduction. In *Functional Programming Languages and Computer Architecture*, pages 255–265. ACM Press, 1993.

[Pan96] Sven Eric Panitz. Termination poofs for a lazy functional language by abstract reduction. Technical Report Fank 06, J. W. Goethe-Universität, FB 20, Professur KIST, `www.ki.informatik.uni-frankfurt.de/papers/articles.html`, 5 1996.

[Pat96] Ross Paterson. Compiling laziness using projections. In *Static Analysis Symposium*, volume 1145 of *LNCS*, pages 255–269, Aachen, Germany, September 1996. Springer.

[PJL91] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a Tutorial*. Prentice-Hall International, London, 1991.

[PJP] Simon L. Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. Draft.

[PJS94] Simon L. Peyton Jones and André Santos. Compilation by transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 184–204. Springer, 1994.

[Sch95] Marko Schütz. The $G^{\#}$-machine: Efficient strictness analysis in Haskell. Technical Report 1/95, Johann Wolfgang Goethe-Universität, Fachbereich Informatik, January 1995.

[Sew92] Julian R. Seward. Towards a strictness analyser for haskell: Putting theory into practice. Master's thesis, University of Manchester, Computer Science Department, 1992.

[vEGHN93] M. van Eekelen, E. Goubault, C.L. Hankin, and E. Nöcker. Abstract reduction: Towards a theory via abstract interpretation. In M.R. Sleep, M.J. Plasmeijer, and M.C.J.D. van Eekelen, editors, *Term Graph Rewriting - Theory and Practice*, chapter 9. Wiley, Chichester, 1993.

[Wad87] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declerative Languages*, chapter 12. Ellis Horwood Limited, Chichester, 1987.