On the Complexity of Set-Based Analysis

Nevin Heintze* David McAllester†

Abstract: We define a general notion of set-based analysis — any language whose operational semantics is defined by environment evaluation has a well defined set-based abstraction. This general definition covers both Aiken and Wimmers' type system and Heintze' set-based analysis. Aiken and Wimmers give a nondeterministic exponential time algorithm for their analysis. Heintze gives an $O(n^3)$ procedure. We show that this discrepancy is due to the complexity of the case statements analyzed. For polymorphic programs with deep patterns in case statements (as in the Aiken-Wimmers language) we show that the problem of determining type-safety under set-based abstraction is complete for deterministic exponential time. The problem remains complete for deterministic exponential time for programs without let (monovariant programs). However, for monovariant programs in which all patterns in case statements are shallow, as in the Heintze language, type-safety under set-based abstraction is decidable in cubic time. We give five additional theorems which (nearly exhaustively) characterize the time complexity of set-based analysis as a function of the complexity of case statements.

1 Introduction

The programming language ML provides automatic type inference for procedures but requires type declarations for data constructors. Other statically typed languages, however, do not require data type definitions. These languages infer types for both procedures and data. Automated inference of data types has been studied both in the context of logic programming [12, 8, 5, 9] and in the context of functional programming [13, 1, 2, 6]. Many different analyses have been developed. These approaches all share a common framework in that they assign sets (or types) to program expressions. Although the various approaches to inferring data types seem to share certain basic assumptions, considerable energy has been put into developing appropriate analyses for various languages. Here we give a general definition of the set-based abstraction of any language whose semantics can be defined by environment evaluation. Intuitively, this general definition of set-based analysis yields the most accurate analysis possible subject to the constraint that each expression in the program is taken to denote a set.

We also give a general definition SBA-safety — intuitively, a program is SBA-safe if it is type-safe under set-based abstraction. We address the computational complexity of determining SBA-safety in a functional language with case statements. In all cases SBA-safety is decidable in deterministic exponential time (even with polyvariance). But the complexity of determining SBA-safety is sensitive to the complexity of case statements. We show here that if we restrict the case statements to have only shallow patterns then SBA-safety (of monovariant programs) is decidable in $O(n^3)$ time under all standard operational semantics. If we allow deep patterns and use the standard "serial" semantics for case statements then determining SBA-safety is complete for deterministic exponential time. We give a systematic study of the computational complexity of determining SBA-safety as a function of the complexity of case statements.

The $O(n^3)$ procedure for determining SBA-safety for shallow patterns is based on a flow analysis and is related to the set-based analysis methods of Heintze [6]. The analysis is also similar to that of Shivers [16] or Jagannathan and Wright [10] but includes data constructors and union types. It can also be viewed as an extension of the recently established equivalence between flow analysis and recursive types in the absence of data constructors [3, 14, 7, 15]. These connections are explored more deeply in [11].

Earlier procedures for determining SBA-safety for programs with deep patterns have been based on general methods of solving set constraints [1, 2]. The general set constraint solvers run in nondeterministic exponential time in the worst case. Here we show that the full power of these set constraint solvers is not needed in determining SBA-safety. SBA-safety can be determined in deterministic exponential time using a direct abstract interpretation of the program.[1]

The basic definition of SBA-safety only applies to monovariant programs. However, the monovariant notion of SBA-safety immediately induces a polyvariant notion of SBA-safety — a polyvariant program is SBA-safe if and only if its let-expansion is SBA-safe (as a monovariant program). Given

*Bell Labs, 600 Mountain Ave, Murray Hill, NJ 07974, nch@bell-labs.com.

†AT&T Labs, 600 Mountain Ave, Murray Hill, NJ 07974, dmac@research.att.com.

[1] The abstract interpretation used here seems outside the technical framework described by Cousot and Cousot [4]. We use an abstract domain derived from the particular program being analyzed. For example, in many cases the abstract values are the program nodes themselves. In more traditional abstract interpretation the programming language as a whole is given an abstract semantics using a single abstract semantic domain.

$$\text{BETA}'\quad \frac{\mathrm{EVAL}((f\ w),\ \rho)}{\mathrm{EVAL}(f,\ \rho),\quad \mathrm{EVAL}(w,\ \rho)}$$

$$\text{IDENT}\quad \frac{\mathrm{EVAL}(\lambda x.e,\ \rho)}{\langle \lambda x.e,\ \rho\rangle \rightarrow \langle \lambda x.e,\ \rho\rangle}$$

$$\text{VAR}\quad \frac{\mathrm{EVAL}(x,\ \rho)}{\langle x,\ \rho\rangle \rightarrow \rho(x)}$$

$$\text{CONS}\quad \frac{\mathrm{EVAL}(0,\ \rho)}{\langle 0,\ \rho\rangle \rightarrow 0}$$

$$\text{BETA}\quad \frac{\begin{array}{l}\mathrm{EVAL}((f\ w),\ \rho)\\ \langle f,\ \rho\rangle \rightarrow \langle \lambda x.u,\ \sigma\rangle\\ \langle w,\ \rho\rangle \rightarrow v\end{array}}{\langle (f\ w),\ \rho\rangle \rightarrow \langle u,\ \sigma[x := v]\rangle}$$

$$\text{CALL}\quad \frac{\langle u,\ \rho\rangle \rightarrow \langle w,\ \sigma\rangle}{\mathrm{EVAL}(w,\ \sigma)}$$

$$\text{RETURN}\quad \frac{\langle u,\ \rho\rangle \rightarrow \langle w,\ \sigma\rangle,\ \langle w,\ \sigma\rangle \rightarrow v}{\langle u,\ \rho\rangle \rightarrow v}$$

Figure 1: Environment evaluation for the pure λ-calculus. A value is either the constant 0 or a pair of the form $\langle \lambda x.e, \sigma\rangle$ where σ is an environment mapping the free variable of $\lambda x.e$ to values. In the rules BETA and RETURN we require that v be a value.

a definition of polyvariant SBA-safety one can consider the question of the computational complexity of determining polyvariant SBA-safety for a variety of programming languages. McAllester has shown [11] that even for first order programs restricted to procedures of at most one argument and shallow case statements determining polyvariant SBA-safety is PSPACE-hard. This is in contrast to Hindley-Milner typability which can be done in nearly linear time for programs of bounded order and arity [11]. This result indicates that polyvariant SBA-safety for shallow case statements is harder to determine than Hindley-Milner typability. On the other hand, we show here that polyvariant SBA-safety with deep case statements is decidable in deterministic singly exponential time, so in a theoretical sense it is no harder than Hindley-Milner typability with unbounded order and arity. Our results also completely characterize the complexity of monomorphic analysis. One gap remains for polyvariant programs — for shallow case statements the polyvariant problem has a lower bound of PSPACE and an upper bound of exponential time.

An important open problem is the efficiency of determining polyvariant SBA-safety in practice. Constrained types seem to provide a promising practical approach [1, 2, 15]. Constrained types are beyond the scope of this paper.

2 SBA-safety

In this section we define the general notion of set-based abstraction and SBA-safety. Ultimately we are interested in performing set-based analysis on programs involving data constructors and case statements. However, our notion of set-based abstract is well defined for any language with an operational semantics defined by environment evaluation. As a preliminary example we consider the pure λ-calculus extended with a constant 0. The terms of this language are defined by the following grammar.

$$e \ ::= \ x \mid 0 \mid \lambda x.e \mid (e_1\ e_2)$$

As usual for the λ-calculus, we let $\lambda xy.e$ abbreviate $\lambda x.\lambda y.e$ and we let $(f\ e_1\ e_2)$ abbreviate $((f\ e_1)\ e_2)$. Similar abbreviations hold for larger numbers of arguments.

Figure 1 gives environment evaluation rules for the pure λ-calculus (with the constant 0). The rules can be viewed as a bottom-up logic program. The assertions describe "events" occurring in a more traditional environment evaluator. The assertion EVAL(e, σ) means that the evaluator is to evaluate e in the environment σ. An evaluation is initiated by making an assertion of the form EVAL(e, []) where e is a closed term and [] is the empty environment. The rule BETA' states that if one is to evaluate an application $(f\ w)$ then one must recursively evaluate both f and w. This rule, as all rules in figure 1, is intended to be used in a forward chaining manner — the evaluation of an application causes the evaluation of the operator and operand. The rule BETA uses values returned from the evaluation of f and w. A *value* is defined to be either the constant 0 or a pair of the form $\langle \lambda x.e,\ \sigma\rangle$ where σ is an environment mapping the free variables of $\lambda x.e$ to values. In the rule BETA and RETURN we require that v be a value.

Suppose we seed the inference rules with the initial assertion EVAL($(\lambda x.x\ 0)$, []). Then from BETA$'$ we can derive EVAL($\lambda x.x$, []) and EVAL(0, []). From IDENT we can infer

$$\langle \lambda x.x,\ []\rangle \rightarrow \langle \lambda x.x,\ []\rangle.$$

From CONS we can infer

$$\langle 0,\ []\rangle \rightarrow 0.$$

So, from BETA we can derive

$$\langle (\lambda x.x\ 0),\ []\rangle \rightarrow \langle x,\ [x := 0]\rangle.$$

By CALL we get EVAL(x, $[x := 0]$). By VAR we then get $\langle x,\ [x := 0]\rangle \rightarrow 0$. And finally, by RETURN we get

$$\langle (\lambda x.x\ 0),\ []\rangle \rightarrow 0.$$

The presence of the constant 0 provides a notion of run time error. A run time error is generated if there is an attempt to apply the constant 0 as a procedure. This notion of run time error is captured in the following inference rule.

$$\text{ERROR}\quad \frac{\begin{array}{l}\mathrm{EVAL}((f\ w),\ \rho)\\ \langle f,\ \rho\rangle \rightarrow 0\end{array}}{\mathrm{ERROR}}$$

$$\text{BETA}'\quad \frac{\mathrm{EVAL}((f\ w))}{\mathrm{EVAL}(f),\quad \mathrm{EVAL}(w)}$$

$$\text{IDENT}\quad \frac{\mathrm{EVAL}(\lambda x.e)}{\lambda x.e \to \lambda x.e}$$

$$\text{CONS}\quad \frac{\mathrm{EVAL}(0)}{0 \to 0}$$

$$\text{ERROR}\quad \frac{\mathrm{EVAL}((f\ w)),\quad f \to 0}{\mathrm{ERROR}}$$

$$\text{BETA}\quad \frac{\begin{array}{l}\mathrm{EVAL}((f\ w))\\ f \to \lambda x.u\\ w \to v\end{array}}{x \to v,\quad (f\ w) \to u}$$

$$\text{CALL}\quad \frac{u \to w}{\mathrm{EVAL}(w)}$$

$$\text{RETURN}\quad \frac{\begin{array}{l}u \to w\\ w \to v\end{array}}{u \to v}$$

Figure 2: Determining SBA-safety. In BETA and RETURN we require that v be either 0 or a λ-expression. e is SBA-safe under the semantics $\vdash_\lambda$ if and only $\mathrm{EVAL}(e) \not\vdash_R$ ERROR where R is the above rule set.

We let $\vdash_\lambda$ be the inference relation defined by the inference rules in figure 1 plus the above error rule. For example, we have the following facts about the inference relation $\vdash_\lambda$.

$$\mathtt{EVAL}(((\lambda x.x\ 0)\ 0),\ [])\ \vdash_\lambda\ \langle(\lambda x.x\ 0),\ []\rangle \to 0$$

$$\mathtt{EVAL}(((\lambda x.x\ 0)\ 0),\ [])\ \vdash_\lambda\ \mathtt{ERROR}$$

We say that a closed term e generates a run time error if $\mathtt{EVAL}(e,\ [])\ \vdash_\lambda$ ERROR. The term $((\lambda x.x\ 0)\ 0)$ generates a run time error. The problem of determining whether a given term generates a run time error is undecidable.

We now formally define set-based abstraction and SBA-safety. These definitions apply to any environment evaluation semantics given by inference rules for deriving assertions of the form $\mathtt{EVAL}(e,\ \sigma)$ and $\langle e,\ \rho\rangle \to v$.

> **Definition:** Let R be any set of inference rules. We define $\vdash_R$ to be the inference relation generated by those rules. We define A to be the following abstraction rule.
>
> $$\mathtt{SBA}\quad \frac{\begin{array}{l}\mathtt{EVAL}(u,\ \rho)\\ \langle u,\ \rho\rangle \to v\\ \mathtt{EVAL}(u,\ \gamma)\end{array}}{\langle u,\ \gamma\rangle \to v}$$
>
> The set-based abstraction of R is defined to be the rule set $R \cup \{A\}$.

For any set R of environment evaluation rules, the rule set $R \cup \{A\}$ defines the set-based abstraction of the semantics R. For example, let λ be the set of inference rules in figure 1 plus the above error rule. The inference relation $\vdash_{\lambda\cup\{A\}}$ defines the set-based abstraction of the pure λ-calculus under the semantics given in figure 1. As an example consider the following term where I abbreviates $\lambda x.x$ and K abbreviates $\lambda xy.x$.

$$((\lambda f.\ (K\ ((f\ I)\ 0)\ (f\ 0)))\ \lambda y.y)$$

This term evaluates to 0 and does not generate a run time error. But the variable y is evaluated in two different environments. The variable f becomes bound to $\lambda y.y$ and the evaluation of $(f\ I)$ results in an environment in which y is bound to I while the evaluation of $(f\ 0)$ results in an environment in which y is bound to 0. So we have the following facts where e is the above term.

$$\mathtt{EVAL}(e,\ [])\ \vdash_\lambda\ \mathtt{EVAL}(y,\ [y := I])$$

$$\mathtt{EVAL}(e,\ [])\ \vdash_\lambda\ \mathtt{EVAL}(y,\ [y := 0])$$

Now the above set-based abstraction rule can be used to obtain the following.

$$\mathtt{EVAL}(e,\ [])\ \vdash_{\lambda\cup\{A\}}\ \langle y,\ [y := I]\rangle \to 0$$

This implies the following.

$$\mathtt{EVAL}(e,\ [])\ \vdash_{\lambda\cup\{A\}}\ \langle(f\ I),\ [f := \lambda y.y]\rangle \to \langle y,\ [y := I]\rangle$$

$$\mathtt{EVAL}(e,\ [])\ \vdash_{\lambda\cup\{A\}}\ \langle(f\ I),\ [f := \lambda y.y]\rangle \to 0$$

$$\mathtt{EVAL}(e,\ [])\ \vdash_{\lambda\cup\{A\}}\ \mathtt{ERROR}$$

So even though e does not generate a run-time error, under the set-based abstraction an error is generated. Intuitively, the set-based abstraction rule states that each expression is associated with a set of values independent of environment. The set of values associated with y must include both I and 0 and hence any application of $\lambda y.y$ can return either I or 0.

> **Definition:** Let R be any set of inference rules. A term e is called SBA-safe under semantics R if $\mathtt{EVAL}(e,\ [])\ \not\vdash_{R\cup\{A\}}$ ERROR.

Intuitively, a term is SBA-safe under semantics R if it is type-safe where the types are the sets of values generated by the rules $R \cup \{A\}$. SBA-safety is decidable for most languages. The decision procedure for SBA-safety for the pure λ-calculus with the constant 0 is shown in figure 2. Under the SBA inference rule A, each expression has a set of values independent of the environment in which it is evaluated. Since the value set is independent of the environment, the environments can be removed from the evaluation rules. The rules of figure 2 generate statements of the form EVAL(u) and $u \to w$ where u and w are subterms of the top level term being evaluated. There are $O(n^2)$ such assertions of the form $u \to w$ where n is the number of subterms of the top level input. The rules can be run to completion in $O(n^3)$ time.

BETA′	$\dfrac{\text{EVAL}((f\ w),\ \rho)}{\text{EVAL}(f,\ \rho),\quad \text{EVAL}(w,\ \rho)}$	BETA	$\dfrac{\text{EVAL}((f\ w),\ \rho) \quad \langle f,\ \rho\rangle \to \langle \lambda x.u,\ \sigma\rangle \quad \langle w,\ \rho\rangle \to v}{\langle (f\ w),\ \rho\rangle \to \langle u,\ \sigma[x := v]\rangle}$
IDENT	$\dfrac{\text{EVAL}(w,\ \rho)}{\langle w,\ \rho\rangle \to \langle w,\ \rho\rangle}$	CALL	$\dfrac{\langle u,\ \rho\rangle \to \langle w,\ \sigma\rangle}{\text{EVAL}(w,\ \sigma)}$
VAR	$\dfrac{\text{EVAL}(x,\ \rho)}{\langle x,\ \rho\rangle \to \rho(x)}$	RETURN	$\dfrac{\langle u,\ \rho\rangle \to \langle w,\ \sigma\rangle,\ \langle w,\ \sigma\rangle \to v}{\langle u,\ \rho\rangle \to v}$
CONS	$\dfrac{\text{EVAL}(0,\ \rho)}{\langle 0,\ \rho\rangle \to 0}$	SYM	$\dfrac{\text{EVAL}(\lambda x.e,\ \rho)}{\text{EVAL}(e,\ \rho[x := ?])}$

Figure 3: An arbitrary reduction environment evaluator. A suspended computation is either the constant 0 or a pair of the form $\langle \lambda x.e, \sigma\rangle$ where σ is an environment mapping the free variable of $\lambda x.e$ to suspended computations. In the rules BETA and RETURN we let v be an arbitrary suspended computation. The notation $\rho[x := ?]$ used in the rule SYM denotes an environment in which x is bound to a fresh variable.

The rules in figure 2 can be viewed as defining a kind of abstract interpretation where the abstract values are subexpressions of the input program. All of the decision procedures for SBA-safety given in this paper are presented as abstract evaluation rules. In many cases, however, there can be exponentially many abstract values.

3 Evaluation Strategies

The environment evaluator shown in figure 1 implements call by value semantics. We also discuss two other evaluation strategies — lazy evaluation and arbitrary reduction. Both of these other evaluation strategies can be implemented in an environment evaluator analogous to the one shown in figure 1. Set-based abstraction and the notion of SBA-safety is defined relative to an arbitrary environment evaluator and hence is also defined for lazy and arbitrary reduction semantics. The notion of SBA-safety is different for these different evaluation strategies. Here we describe the changes needed to the inference rules to implement the other evaluation strategies and the consequences for the notion of of SBA-safety.

First we consider lazy evaluation. To implement lazy evaluation we first define an appropriate notion of a lazy value. A lazy value is either 0 or a pair of the form $\langle \lambda x.e,\ \sigma\rangle$ where σ maps the free variables of $\lambda x.e$ to suspended computations — a suspended computation is either 0 or a pair of the form $\langle u,\ \sigma\rangle$ where u is any term and σ maps the free variables of u to other suspended computations. Note that values are a special case of suspended computations in which the expression must be a λ-expression. We can modify the rules in figure 1 to implement lazy evaluation with three changes. First, we change the restriction on v in the rule RETURN so that it ranges over lazy values rather than strict values. Second, we modify the rule BETA' so that $\text{EVAL}((f\ w)\ \rho)$ generates $\text{EVAL}(f,\ \rho)$ but does not generate $\text{EVAL}(w,\ \rho)$. Finally, we replace the rule BETA with the following.

LAZY $\quad \text{EVAL}((f\ w),\ \rho)$

$$\frac{\langle f,\ \rho\rangle \to \langle \lambda x.u,\ \sigma\rangle}{\langle (f\ w),\ \rho\rangle \to \langle u,\ \sigma[x := \langle w,\ \rho\rangle]\rangle}$$

Let WRONG be the expression (0 0). Under lazy evaluation the term (λx.0 WRONG) does not generate a run time error because the term WRONG is never evaluated. Now consider adding the set-based abstraction rule which forces expressions to denote sets independent of environments. The set-based abstraction does not change the fact that this term does not generate an error — the term is SBA-safe under lazy semantics.

As another example, let $\perp$ be $((\lambda h.(h\ h))\ (\lambda h.\ (h\ h)))$. Under the call by value semantics given in figure 1 the term (λx.WRONG $\perp$) does not generate a run time error. This is also true under the set-based abstraction — this term is SBA-safe under call by value semantics. Under lazy semantics this term generates a run time error and fails to be SBA-safe.

Arbitrary reduction semantics is different from both strict and lazy evaluation. Arbitrary reduction semantics is most easily formulated using reduction rules. In particular we have the following reduction rules.

$$(\lambda x.e\ w) \to e[w/x]$$
$$(0\ w) \to \text{ERROR}$$
$$e \to \text{ERROR} \quad \text{if } e \text{ contains ERROR}$$

These rules can be applied anywhere inside a term. For example, under both lazy and strict semantics the term λx.WRONG does not generate a run time error. Under arbitrary reduction, however, we have WRONG $\to$ ERROR and hence

$$\lambda x.\text{WRONG} \to \text{ERROR}.$$

To formulate a notion of SBA-safety based on arbitrary reduction semantics we must first implement the arbitrary reduction semantics as an environment evaluator. An environment evaluator implementing arbitrary reduction semantics is shown in figure 3. Under the arbitrary reduction

$$\text{BETA}' \quad \frac{\text{EVAL}(e)}{\text{EVAL}(u) \quad u \text{ a subterm of } e}$$

$$\text{IDENT} \quad \frac{\text{EVAL}(u)}{u \to u}$$

$$\text{ERROR} \quad \frac{\text{EVAL}((f\ w)), \quad f \to 0}{\text{ERROR}}$$

$$\text{BETA} \quad \frac{\begin{array}{l}\text{EVAL}((f\ w)) \\ f \to \lambda x.u\end{array}}{x \to w, \quad (f\ w) \to u}$$

$$\text{TRANS} \quad \frac{u \to w, w \to s}{u \to s}$$

Figure 4: Determining SBA-safety under arbitrary reduction.

semantics a given term can be rewritten in many different ways and terms generally do not have unique values. If the rules generate $\langle u,\ \rho\rangle \to \langle w,\ \sigma\rangle$ then the term represented by $\langle u,\ \rho\rangle$ reduces under the above reduction rules to the term represented by $\langle w,\ \sigma\rangle$. Also, it is possible to show that for any closed term e, we have that $e \to$ ERROR if and only if EVAL(e, []) $\vdash$ ERROR where $\vdash$ is the inference relation defined by the inference rules in figure 3. In other words, the notion of run time error defined by figure 3 is the same as the notion of run time error defined by the reduction rules. For example, note that the term λx.WRONG generates a run time error under the semantics in figure 3 — the rule SYM allows evaluation to proceed into the body of a λ-expression even if that λ-expression is never applied. The notation $\rho[x :=?]$ denotes an environment which maps x to a fresh variable. The proof that the notion of run time error defined by figure 3 is the same as that defined by the reduction rules involves establishing an invariant relating the two systems. Unfortunately, this invariant is somewhat difficult to state and we omit the proof here.

In the remainder of this paper we consider only arbitrary reduction semantics. Arbitrary reduction semantics has the drawback of being less accurate in cases where a more restricted reduction strategy is actually used. But we focus on arbitrary reduction semantics here for three reasons. First, it can be argued that set-based analysis derived from arbitrary reduction semantics catches more programming errors because it rejects intuitively ill-typed terms, such as λx.WRONG, that can be accepted under other semantics. Second, arbitrary reduction semantics is not fundamentally different from the semantics of other evaluation strategies — the complexity of determining SBA-safety seems insensitive the choice of reduction strategy. Finally, arbitrary reduction semantics simplifies the algorithms for determining SBA-safety. Figure 4 gives the algorithm for determining SBA-safety for the pure λ-calculus with 0 under arbitrary reduction semantics. It useful to compare figures 2 and 4 and note the simplicity gained from arbitrary reduction semantics. It is also interesting to note that a term of this calculus is SBA-safe under arbitrary reduction if and only if it is typable in the Amadio-Cardelli type system [3, 14].

4 Case Statements and their Semantics

The syntax and arbitrary reduction semantics of a simple operational language with case statements and nondeterminism is given in figure 5. The nondeterminism primitive AMB does not effect the simplicity or run time of set-based analysis and is convenient for giving examples. Case is a pattern matching construct. In the expression

$$\text{CASE}(e \text{ of } p_1 : b_1,\ \ldots,\ p_n : b_n)$$

each p_i is a pattern to be matched against the value of e. A successful match binds the variables in p_i and continues by executing b_i under those bindings. We consider four different semantics for case statements. Case statements can be serial or parallel. The serial case reduction rule generates the reduction CASE($c(a)$ of $c(x) : x,\ y : y) \to_s a$ but not the reduction CASE($c(a)$ of $c(x) : x,\ y : y) \to_p c(a)$; the latter is however generated by the parallel rule. Case statements can also be strict or nonstrict. Strict case statements generate an error when no pattern matches the given value. Nonstrict case statements do not generate such an error — they simply "diverge" if no pattern matches. The choice of serial or parallel and strict or nonstrict gives four possibilities each of which corresponds to a particular subsets of the reduction rules in figure 5. In all cases the order of reduction is unconstrained.

As an example let e be the following term.

$$(\lambda h.\ c((h\ h))\ \lambda h.\ c((h\ h)))$$

We have that $e \to c(e)$. Intuitively, e can be viewed as the infinite term $c(c(c(\ldots)))$. However, the reduction rules only manipulate finite terms such as $c(c(e))$. Now consider the following case statement under the serial semantics.

$$\text{CASE}(e \text{ of } c(c(x)) : x,\ c(y) : y)$$

Under the serial semantics the branch corresponding to $c(y)$ is never taken. The term e neither matches nor clashes with the pattern $c(c(x))$. The term $c(e)$ again neither matches nor clashes with $c(c(x))$. But $c(c(e))$ matches $c(c(x))$ with x bound to e.

As in the case of the pure λ-calculus, the reduction rule semantics can be translated into an environment evaluator which implements the same notion of run time error — for any closed term e we have $e \to w$ where w contains ERROR if and only if EVAL(e, []) $\vdash$ ERROR where $\vdash$ is the inference relation defined by the environment evaluator. The environment evaluator for the language with case is given in figure 6. The case statements can be either serial or parallel and either strict or nonstrict. The evaluator in figure 6 is serial and strict. To get a semantics for parallel case the clash antecedent is removed from the rule CASE. To get a nonstrict semantics we delete the rule CERR.

$$
\begin{array}{rl}
e ::= & x \mid (e_1\ e_2) \mid \lambda x.e \mid c(e_1,\ \ldots,\ e_n) \mid \mathrm{CASE}(e_0 \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n) \mid \mathrm{AMB}(e_1,\ e_2) \\
p ::= & x \mid c(p_1,\ \ldots,\ p_n) \mid x \textit{ as } p
\end{array}
$$

β	$((\lambda x.e)\ \ v) \to e[v/x]$
β-error	$(c(v_1,\ \ldots,\ v_n)\ v) \to$ ERROR
AMB1 AMB2	$\mathrm{AMB}(e_1,\ e_2) \to e_1$ $\mathrm{AMB}(e_1,\ e_2) \to e_2$
parallel case	$\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n) \to \sigma(e_i)$ where $u = \sigma(p_i)$ and $\sigma(x) = x$ for $x \notin p_i$
serial case	$\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n) \to \sigma(e_i)$ where $u = \sigma(p_i)$ and $\sigma(x) = x$ for $x \notin p_i$ and p_j clashes with u for $j < i$
error case	$\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n) \to$ ERROR p_j clashes with u for $1 \leq j \leq n$

Figure 5: Reduction semantics for case statements. A term is any expression generated by e in the above grammar subject to the constraint that the patterns in case expressions must be linear, i.e., a given variable occurs at most once in any given pattern. The operational semantics of terms is defined by the above set of term reduction rules. We will only consider certain subsets of the reduction rules. We adopt either the parallel or the serial case rule, but not both, and the case error rule is optional. A term e clashes with a pattern $c(p_1,\ \ldots,\ p_n)$ if e is a λ-expression, an application of a different constructor, or a term of the form $c(u_1,\ \ldots,\ u_n)$ where some u_i clashes with p_i.

BETA
$$\frac{\begin{array}{l}\mathrm{EVAL}((f\ w),\ \rho) \\ \langle f,\ \rho\rangle \to \langle \lambda x.u,\ \sigma\rangle \\ \langle w,\ \rho\rangle \to v\end{array}}{\langle (f\ w),\ \rho\rangle \to \langle u,\ \sigma[x := v]\rangle}$$

CASE
$$\frac{\begin{array}{l}\mathrm{EVAL}(\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n),\ \rho) \\ \langle u,\ \rho\rangle \to v \\ v \text{ clashes with } p_j \text{ for all } 1 \leq j < i \\ v \text{ matches } p_i\end{array}}{\begin{array}{l}\langle \mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n),\ \rho\rangle \\ \to \langle e_i,\ \rho[p_i := v]\rangle\end{array}}$$

CONS
$$\frac{\begin{array}{l}\mathrm{EVAL}(c(e_1,\ \ldots,\ e_n),\ \rho) \\ \langle e_1,\ \rho\rangle \to v_1,\ \ldots,\ \langle e_n,\ \rho\rangle \to v_n\end{array}}{\langle c(e_1,\ \ldots,\ e_n),\ \rho\rangle \to c(v_1,\ \ldots,\ v_n)}$$

AMB
$$\frac{\mathrm{EVAL}(\mathrm{AMB}(e_1,\ e_2),\ \rho)}{\begin{array}{l}\langle \mathrm{AMB}(e_1,\ e_2),\ \rho\rangle \to \langle e_1,\ \rho\rangle \\ \langle \mathrm{AMB}(e_1,\ e_2),\ \rho\rangle \to \langle e_2,\ \rho\rangle\end{array}}$$

BERR
$$\frac{\begin{array}{l}\mathrm{EVAL}((f\ w),\ \rho) \\ \langle f,\ \rho\rangle \to c(v_1,\ \ldots,\ v_n)\end{array}}{\mathrm{ERROR}}$$

CERR
$$\frac{\begin{array}{l}\mathrm{EVAL}(\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n),\ \rho) \\ \langle u,\ \rho\rangle \to v \\ v \text{ clashes with } p_i \text{ for all } 1 \leq i \leq n\end{array}}{\mathrm{ERROR}}$$

BETA′
$$\frac{\mathrm{EVAL}((f\ w),\ \rho)}{\mathrm{EVAL}(f,\ \rho),\quad \mathrm{EVAL}(w,\ \rho)}$$

CASE′
$$\frac{\mathrm{EVAL}(\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n),\ \rho)}{\mathrm{EVAL}(u,\ \rho)}$$

CONS′
$$\frac{\mathrm{EVAL}(c(e_1,\ \ldots,\ e_n),\ \rho)}{\mathrm{EVAL}(e_1,\ \rho),\ \ldots,\ \mathrm{EVAL}(e_n,\ \rho)}$$

VAR
$$\frac{\mathrm{EVAL}(x,\ \rho)}{\langle x,\ \rho\rangle \to \rho(x)}$$

CALL
$$\frac{\langle u,\ \rho\rangle \to \langle w,\ \sigma\rangle}{\mathrm{EVAL}(w,\ \sigma)}$$

RETURN
$$\frac{\langle u,\ \rho\rangle \to \langle w,\ \sigma\rangle,\ \langle w,\ \sigma\rangle \to v}{\langle u,\ \rho\rangle \to v}$$

IDENT
$$\frac{\mathrm{EVAL}(u,\ \rho)}{\langle u,\ \rho\rangle \to \langle u,\ \rho\rangle}$$

SYM1
$$\frac{\mathrm{EVAL}(\lambda x.u,\ \rho)}{\mathrm{EVAL}(u,\ \rho[x := ?])}$$

SYM2
$$\frac{\mathrm{EVAL}(\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n),\ \rho)}{\begin{array}{l}\mathrm{EVAL}(e_1,\ \rho[p_1 := ?]) \\ \vdots \\ \mathrm{EVAL}(e_n,\ \rho[p_n := ?])\end{array}}$$

Figure 6: Environment evaluation for case statements. A *suspended computation* is a pair $\langle u,\ \sigma\rangle$ where u is a term and σ is a substitution, i.e., a finite mapping from variables to values. A *value* is either a variable, a suspended computation or an expression of the form $c(v_1,\ \ldots,\ v_n)$ where each v_i is a value. A *procedure* is a value of the form $\langle \lambda x.u,\ \sigma\rangle$. A value v clashes with a pattern $c(p_1,\ \ldots,\ p_n)$ if v is either a procedure, an application of a different constructor, or of the form $c(v_1,\ \ldots,\ v_n)$ where some v_i clashes with p_i. Matching is defined in the usual way. The substitution $\rho[p := v]$ is the substitution identical to ρ except that it maps variables in p to the corresponding values derived from matching p to v. The substitution $\rho[p := ?]$ is identical to ρ except that variables occurring in p are bound to "fresh" variables, i.e., ones not occurring free in values in ρ.

	Polyvariance	Deep	as	Overlapping	Serial	Strict	Lower Bound	Upper Bound
1.	+	+	+	+	+	+	EXPTIME	EXPTIME
2.	-	-	+	+	+	+	—	$O(n^3)$
3.	-	+	+	-	-	-	EXPTIME	EXPTIME
4.	-	+	-	+	+	-	EXPTIME	EXPTIME
5.	-	+	-	+	-	-	—	$O(n^3)$
6.	-	+	-	+	-	+	coNP	coNP
7.	-	+	-	-	+	-	—	$O(n^3)$
8.	-	+	-	-	+	+	—	$O(n^6 \log n)$

Figure 7: Complexity results. The above table shows the computational complexity of determining SBA-safety for a variety of operational semantics and syntactic restrictions on the case statement. A polyvariant term is one in which the grammar is extended to let-expressions. The let-expansion of a polyvariant term is the result of replacing each let expression let $x = e$ in u by the expansion $u[e/x]$. By definition, a polyvariant term is SBA-safe if its let-expansion is SBA-safe. A plus in the first column indicates that let-expressions are allowed. A shallow pattern is either a variable or a pattern of the form $c(x_1, \ldots, x_n)$ or y *as* $p(x_1, \ldots, x_n)$. A minus in the second column means we allow only shallow patterns in case statements. A minus in the third column means that we do not allow "as" in patterns. A minus in the fourth column means that the patterns in a given case statement must be disjoint, i.e., no value can match two patterns simultaneously. A minus in the fifth column means that we drop the clash antecedent from the rule CASE in figure 6. A minus in the last column means that we drop the rule CERR form figure 6. Changing an entry from a minus to a plus always makes determining SBA-safety more difficult. This monotonicity property implies that the results in the table completely characterize the complexity of SBA-safety for monovariant terms.

Any environment evaluator determines a well defined notion of set-based abstraction and SBA-safety. Here we are interested in the complexity of determining SBA-safety as a function of the complexity of case statements. In addition to the variations discussed above we consider various restrictions on the patterns allowed in case statements. In particular, we consider restricting case statements so that all patterns are shallow, i.e., are of the form x or $c(x_1, \ldots, x_n)$ or x *as* p where p is shallow. Shallow patterns are sufficient to express arbitrary case statements but the conversion of an arbitrary case statement to shallow patterns alters the set-based abstraction. SBA-safety is easier to determine with shallow patterns. We also consider whether the programs are allowed to have let constructs, i.e., be polyvariant, whether patterns are allowed to contain as, and whether the patterns in a single case statement are allowed to overlap, i.e., have common instances. Our results on the complexity of determining SBA-safety are shown in figure 7.

5 Determining SBA-Safety

In this section we prove all of the upper bounds in figure 7, except that in line 8, plus the lower bound in line 6. The upper bounds are proved by giving decision procedures for determining SBA-safety under the various conditions corresponding to the lines in the table. All these decision procedures can be viewed as a form of abstract evaluation with abstract values derived from program text. Figure 8 defines the abstract evaluation process for shallow patterns. Intuitively, the abstract values are the subterms of the input expression that are of the form $\lambda x.u$ or $c(e_1, \ldots, e_n)$. To simplify the rules, however, other subterms of the input are also treated as abstract values, although these other subterms do not match or clash with nontrivial patterns.

The correctness of the decision procedure encoded in figure 8 relies on the fact that match and clash are well defined operations on the abstract values and *shallow* patterns, i.e., if v is a subterm of the input of the form $\lambda x.u$ or $c(e_1, \ldots, e_n)$, and p is a shallow pattern, then v matches p if and only if every concrete value represented by v matches p and v clashes with p if and only if only if every concrete value represented by v clashes with p. The set of concrete values represented by an abstract value v is simply the set of values of v under the set based abstraction of environment evaluation.

A more detailed discussion of $O(n^3)$ decision procedures for shallow case statements, and the relationship to the inference of recursive types, can be found in [11].

When deep patterns are allowed the match and clash operations cease to be well defined for abstract values of the form $c(e_1, \ldots, e_n)$. For example, the some concrete values of $c((f\ u))$ may match the pattern $c(c(x))$ while others may clash with this pattern. The abstract value $c((f\ u))$ is not sufficiently refined to allow accurate abstract evaluation. This problem is solved by adding sets of pattern skeletons to the abstract values. The skeleton of a pattern is the result of replacing each variable by the constant (universal set) 1, e.g., $c(x, b(y, z))$ becomes $c(1, b(1, 1))$. The language is restricted so that only linear patterns are allowed. The skeleton of a linear pattern completely determines the set of values which match and clash with that pattern. To handle deep patterns we construct abstract values of the form $\langle c(e_1, \ldots, e_n), \mathcal{P}, \mathcal{N}\rangle$ where $\mathcal{P}$ and $\mathcal{N}$ are sets of skeletons of patterns in the input. The abstract value $\langle c(e_1, \ldots, e_n), \mathcal{P}, \mathcal{N}\rangle$ represents the set of concrete values of the term $c(e_1, \ldots, e_n)$ that match all skeletons in $\mathcal{P}$ and clash with all skeletons in $\mathcal{N}$. We require that in abstract values of the form $\langle c(e_1, \ldots, e_n), \mathcal{P}, \mathcal{N}\rangle$ we have that each skeleton in $\mathcal{P}$ or $\mathcal{N}$ starts with the constructor c. These skeletons suffice to make match and clash operations well defined on abstract values. We now have that the abstract value $\langle c((f\ u)), \{c(c(1))\}, \{\}\rangle$ matches the pattern $c(c(x))$ and the abstract value $\langle c((f\ u)), \{\}, \{c(c(1))\}\rangle$ clashes with the pattern $c(c(x))$. Note that in practice there is likely to be only a small number of skeletons of patterns in the input starting with a given constructor. Note that let-expansion does not introduce new pattern skeletons. This implies that even after let-expansion there is only a singly exponential number of abstract values in the worst case.

Figure 9 gives the abstract evaluation process for deep patterns. The rules exploit a variety of simplifications. First, they assume that only binary constructors are used in the program. Standard methods of representing higher arity

$$\text{BETA}\quad \frac{\begin{array}{l}\mathrm{EVAL}((f\ w))\\ f \to \lambda x.u\end{array}}{x \to w,\ \ (f\ w) \to u}$$

$$\text{CASE}\quad \frac{\begin{array}{l}\mathrm{EVAL}(\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n))\\ u \to v\\ v \text{ clashes with } p_j \text{ for } 1 \leq j < i\\ v \text{ matches } p_i \text{ with substitution } \sigma\end{array}}{\begin{array}{l}\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n) \to e_i\\ x \to \sigma(x) \text{ for all } x \text{ in } p_i\end{array}}$$

$$\text{AMB}\quad \frac{\mathrm{EVAL}(\mathrm{AMB}(e_1,\ e_2))}{\mathrm{AMB}(e_1,\ e_2) \to e_1,\ \mathrm{AMB}(e_1,\ e_2) \to e_2}$$

$$\text{BERR}\quad \frac{\begin{array}{l}\mathrm{EVAL}((f,\ w))\\ f \to c(u_1,\ \ldots,\ u_n)\end{array}}{\mathrm{ERROR}}$$

$$\text{DOWN}\quad \frac{\mathrm{EVAL}(e)}{\mathrm{EVAL}(u)\ \ u \text{ a subterm of } e}$$

$$\text{IDENT}\quad \frac{\mathrm{EVAL}(u)}{u \to u}$$

$$\text{TRANS}\quad \frac{e \to u,\ \ u \to w}{e \to w}$$

$$\text{CERR}\quad \frac{\begin{array}{l}\mathrm{EVAL}(\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n),\ \rho)\\ u \to v\\ v \text{ clashes with } p_i \text{ for all } 1 \leq i \leq n\end{array}}{\mathrm{ERROR}}$$

Figure 8: Flow rules for shallow patterns. When we only allow shallow patterns, monovariant SBA-safety can be determined by a "flow analysis" which generates arcs of the form $u \to w$ where u and w are subterms of the input expressions. In this case the assertion EVAL(u) simply means that u appears in the input, i.e., is a program node. The full deductive closure of the above rules on a given input program can be computed in $O(n^3)$ time. A monovariant term u with shallow case statements is SBA-safe if and only if the above rules fail to derive ERROR from EVAL(u).

$$\text{BETA}\quad \frac{\begin{array}{l}\mathrm{EVAL}((f\ w))\\ f \to \lambda x.u\end{array}}{x \to w,\ (f\ w) \to u}$$

$$\text{CASE}\quad \frac{\begin{array}{l}\mathrm{EVAL}(\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n),\ \rho)\\ u \to v\\ v \text{ clashes with } p_j \text{ for all } 1 \leq j < i\\ v \text{ matches } p_i\end{array}}{\begin{array}{l}\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n) \to e_i\\ \mathrm{BIND}(p_i,\ v)\end{array}}$$

$$\text{CONS}\quad \frac{\begin{array}{l}\mathrm{EVAL}(\langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle)\\ e_1 \to v_1,\ e_2 \to v_2\\ \mathrm{EVAL}(v_1),\ \mathrm{EVAL}(v_2)\\ c(v_1,\ v_2) \text{ matches every element of } \mathcal{P}\\ c(v_1,\ v_2) \text{ clashes with every element of } \mathcal{N}\end{array}}{\begin{array}{l}c(e_1,\ e_2) \to \langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle\\ \langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle \to c(v_1,\ v_2)\end{array}}$$

$$\text{AMB}\quad \frac{\mathrm{EVAL}(\mathrm{AMB}(e_1,\ e_2))}{\mathrm{AMB}(e_1,\ e_2) \to e_1,\ \mathrm{AMB}(e_1,\ e_2) \to e_2}$$

$$\text{BERR}\quad \frac{\begin{array}{l}\mathrm{EVAL}((f,\ w))\\ f \to c(v_1,\ v_2)\end{array}}{\mathrm{ERROR}}$$

$$\text{CERR}\quad \frac{\begin{array}{l}\mathrm{EVAL}(\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n),\ \rho)\\ u \to v\\ v \text{ clashes with } p_i \text{ for all } 1 \leq i \leq n\end{array}}{\mathrm{ERROR}}$$

$$\text{DOWN}\quad \frac{\mathrm{EVAL}(e)}{\mathrm{EVAL}(u),\ u \text{ a subterm of } e}$$

$$\text{TEMP}\quad \frac{\begin{array}{l}\mathrm{EVAL}(c(e_1,\ e_2))\\ \mathcal{P} \text{ and } \mathcal{N} \text{ subsets of the skeletons of}\\ \text{patterns in the input starting with } c\end{array}}{\mathrm{EVAL}(\langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle)}$$

$$\text{TRANS}\quad \frac{u \to w,\ w \to v}{u \to v}$$

$$\text{IDENT}\quad \frac{\mathrm{EVAL}(u)}{u \to u}$$

$$\text{BIND1}\quad \frac{\begin{array}{l}\mathrm{BIND}(c(p_1,\ p_2),\ v)\\ v \to c(v_1,\ v_2)\end{array}}{\mathrm{BIND}(p_1,\ v_1),\ \mathrm{BIND}(p_2,\ v_2)}$$

$$\text{BIND2}\quad \frac{\mathrm{BIND}(x,\ v)}{x \to v}$$

$$\text{BIND3}\quad \frac{\mathrm{BIND}(x \textit{ as } p,\ v)}{x \to v,\ \mathrm{BIND}(p,\ v)}$$

Figure 9: The general case. These rules are to be applied to the let-expansion of the input term. The *skeleton* of a pattern is the result of replacing variables by 1, e.g., $c(x,\ b(y,\ z))$ becomes $c(1,\ b(1,\ 1))$. The rules define an evaluation process over abstract values. Each abstract value is either a subterm of the input or a "template" of the form $\langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle$ where $c(e_1,\ e_2)$ is a subterm of the input and $\mathcal{P}$ and $\mathcal{N}$ are sets of skeletons. The template $\langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle$ represents the set of values of $c(e_1,\ e_2)$ that match elements of $\mathcal{P}$ and clash with elements of $\mathcal{N}$. A template value $\langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle$ matches a pattern p if the skeleton of p is in $\mathcal{P}$ and clashes with a pattern q if either q starts with a constructor other than c or the skeleton of q is in $\mathcal{N}$. A pair $c(v_1,\ v_2)$ of abstract values v_1 and v_2 matches $c(p_1,\ p_2)$ if v_1 matches p_1 and v_2 matches p_2 and clashes with $c(p_1,\ p_2)$ if v_1 clashes with p_1 or v_2 clashes with p_2. The rules manipulate only abstract values and pairs of abstract values. Since let-expansion does not introduce new pattern skeletons, the number of abstract values is bounded by a single exponential in input term size.

BETA $$\frac{\mathrm{EVAL}((f\ w)),\ f \rightarrow \lambda x.u}{x \rightarrow w,\ \ (f\ w) \rightarrow u}$$

CASE $$\frac{\begin{array}{l}\mathrm{EVAL}(\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n))\\ \mathrm{INTER}(u,\ p_i)\end{array}}{\begin{array}{l}\mathrm{CASE}(u \text{ of } p_1 : e_1,\ \ldots,\ p_n : e_n) \rightarrow e_i\\ \mathrm{BIND}(p_i,\ u)\end{array}}$$

AMB $$\frac{\mathrm{EVAL}(\mathrm{AMB}(e_1,\ e_2))}{\mathrm{AMB}(e_1,\ e_2) \rightarrow e_1,\ \mathrm{AMB}(e_1,\ e_2) \rightarrow e_2}$$

BERR $$\frac{\mathrm{EVAL}((f,\ w)),\ f \rightarrow c(u_1,\ \ldots,\ u_n)}{\mathrm{ERROR}}$$

DOWN1 $$\frac{\mathrm{EVAL}(e)}{\mathrm{EVAL}(u)\ \ u \text{ a subterm of } e}$$

DOWN2 $$\frac{\mathrm{EVAL}(e)}{\mathrm{SKEL}(\overline{p})\ \ p \text{ a pattern in } e}$$

IDENT $$\frac{\mathrm{EVAL}(u)}{u \rightarrow u}$$

TRANS $$\frac{e \rightarrow u,\ \ u \rightarrow w}{e \rightarrow w}$$

INT1 $$\frac{\mathrm{EVAL}(u)}{\mathrm{INTER}(u,\ 1)}$$

INT2 $$\frac{\begin{array}{l}\mathrm{EVAL}(c(e_1,\ \ldots,\ e_n)),\ \mathrm{SKEL}(c(s_1,\ \ldots,\ s_n))\\ \mathrm{INTER}(e_1,\ s_1),\ \ldots,\ \mathrm{INTER}(e_n,\ s_n)\end{array}}{\mathrm{INTER}(c(e_1,\ \ldots,\ e_n),\ c(s_1,\ \ldots,\ s_n))}$$

INT3 $$\frac{\mathrm{EVAL}(u),\ u \rightarrow v,\ \mathrm{INTER}(v,\ s)}{\mathrm{INTER}(u,\ s)}$$

BIND1 $$\frac{\begin{array}{l}\mathrm{BIND}(c(p_1,\ \ldots,\ p_n),\ v),\ v \rightarrow c(e_1,\ \ldots,\ e_n)\\ \mathrm{INTER}(e_1,\ \overline{p_1}),\ \ldots,\ \mathrm{INTER}(e_n,\ \overline{p_n})\end{array}}{\mathrm{BIND}(p_1,\ e_1),\ \ldots,\ \mathrm{BIND}(p_n,\ e_n)}$$

BIND2 $$\frac{\mathrm{BIND}(x,\ v)}{x \rightarrow v}$$

Figure 10: Abstract Evaluation for monovariant terms without "as" using nonstrict parallel case statements. We write $\overline{p}$ to denote the skeleton of a pattern p. The assertion INTER(u, s) represents the statement that some value of the term u matches the skeleton s, i.e., the value sets represented by u and s intersect. These rules can be run to completion (forward chaining) in $O(n^3)$ time.

constructors in terms of pairing constructors turn out to preserve SBA-safety. So without loss of generality we can assume that all constructors are binary. The second simplification involves the use of expressions of the form $c(v_1,\ v_2)$ where v_1 and v_2 are abstract values. Such expressions are not themselves considered to be abstract values but rather simply a way of holding a pair of abstract values. The abstract values are those expressions v such that the inference rules prove EVAL(v). The rules ensure that the expression $c(v_1,\ v_2)$ is constructed only when v_1 and v_2 are abstract values. The notion of match and clash is defined in the obvious way for value pairs of the form $c(v_1,\ v_2)$. The rules in figure 9 can be run (forward chaining) to completion in time singly exponential in the size of the (polyvariant) input term.

Figure 10 gives an abstract evaluation process which is used for lines 5 through 8 in figure 7. In all these cases we consider monomorphic programs without "as". In the case of non-overlapping patterns (lines 7 and 8) parallel and serial case statements behave identically. So lines 5 through 8 can all be viewed as having parallel case statements. Lines 5 and 7 are essentially identical. In line 5 overlapping patterns are allowed but the case statements must be parallel. In line 7 the patterns must be non-overlapping but the case statements can be sequential. But sequentiality is irrelevant for non-overlapping patterns. For lines 5 and 7 a given term is SBA-safe if and only if the rules in figure 10 do not derive ERROR when applied to that term. In lines 6 and 8 the case statements are strict and in order or a term to be safe we need perform an additional "escape analysis" as described below.

Before considering escape analysis we will give some comments about the inference rules in figure 10. For parallel case statements without "as" we can take the abstract values to be subterms of the input of the form $\lambda x.u$ and $c(e_1,\ \ldots,\ e_n)$. Because we are allowing deep patterns, match and clash operations are not well defined on these abstract values. However, because we consider only parallel case statements we do not need to test for clashes while computing values. Furthermore, because patterns do not contain "as" we do not need to consider binding variables to proper subsets of the sets represented by subterms. Note that matching the pattern x *as* $c(a)$ to the abstract value $c((f\ u))$ may involves binding x to a *proper subset* of the values of the term $c((f\ u))$. But if the patterns do not contain "as" then no such proper subsets are needed. One must still be careful to avoid extraneous bindings. For example, consider matching $c(a,\ y)$ to the term $c((f\ u),\ w)$. This should generate $y \rightarrow w$ provided that $(f\ u) \rightarrow a$. Note that the set based abstraction rule, combined with the rule CONS in figure 6, ensures that the set represented $c(e_1,\ e_2)$ corresponds to the cross product of the sets represented by e_1 and e_2. Matching $c(p,\ y)$ to $c(u,\ w)$ should generate $y \rightarrow w$ provided that the set of values represented by u contains a value matching the pattern p. The rules INT1, INT2, and INT3 in figure 10 are used to keep track of "intersections" between the sets represented by subterms of the input and the skeletons of patterns.

Lines 6 and 8 in figure 7 require the case statements to be strict, i.e., an error is generated when a value clashes with all patterns in a case statement. The inclusion of this error condition for case statements does not change the set of values associated with any given term. Hence the rules in figure 10 can still be used to compute a representation of the set of values associated with each subterm of the input term. We can now associate each subterm u with a regular term language generated by a nonterminal X_u in a certain regular grammar. The grammar for a given input term e is defined by the following productions.

$$X_u \to \bot$$

$$X_u \to \Lambda \ \text{ if } \mathtt{EVAL}(e) \vdash u \to \lambda x.w$$

$$X_u \to c(X_{e_1},\ \ldots,\ X_{e_n}) \ \text{ if } \mathtt{EVAL}(e) \vdash u \to c(e_1,\ \ldots,\ e_n)$$

In these productions $\bot$ and Λ are special tokens representing divergent computations and procedures respectively. The input term e is SBA-safe if the rules do not derive ERROR form $\mathtt{EVAL}(e)$ and for each subterm of e of the form $\mathtt{CASE}(u \ \mathtt{of}\ p_1 : e_1,\ \ldots,\ p_n : e_n)$ no term generated by X_u under the above grammar clashes with all p_i.

We now define an ***escape problem*** to consist of a regular language defined by a pair $\langle X,\ G\rangle$ where X is a nonterminal symbol and G is a regular term grammar (a set of productions) plus a set of patterns (first order terms) $p_1,\ \ldots,\ p_n$. An escape problem is *satisfiable* if there exists a term t generated by X under G such that t clashes with all p_i. An escape problem is called *disjoint* if the patterns do not overlap, i.e., no patterns share a common instance. An escape problem is called *linear* if each p_i is linear, i.e., no variable in p_i occurs multiple times in p_i. Using the method just described for constructing grammars, each case statement in an input term defines a linear escape problem. For lines 7 and 8 in figure 7 a term is SBA-safe if and only if the rules in figure 10 fail to derive ERROR and the escape problem defined by each case statement in the input term is unsolvable. So we are left with the problem of determining solvability of escape problems.

Note that with a linear number of nondeterministic choices one can guess the escaping term, i.e., the one that clashes with all patterns. So the set of satisfiable escape problems in NP and hence the set of SBA-safe terms is in coNP. This gives the upper bound in line 6. The lower bound in line six follows from the fact that satisfiability for linear escape problems is NP hard. This can be proved by a simple reduction from Boolean satisfiability where truth assignments correspond to terms and clashing corresponds to satisfying a clause.

The situation is quite different for the linear disjoint escape problems that arise in line 8. In this case satisfiability is polynomial time decidable. The decision procedure involves a mathematically curious non-constructive algorithm. Intuitively, the algorithm computes the "volume" of the set of terms generated by the grammar and the volume covered by each pattern. Since the patterns are non-overlapping, the volume covered by all patterns together is just the sum of the volumes of each pattern individually. The escape problem is satisfiable if and only if the sum of the volumes of the patterns is strictly less than the volume of the grammar. However, this calculation does not provide an escaping term. An escaping term can be found in polynomial time by using the decision procedure for satisfiability to filter a binary search for the escaping term. But the fundamental decision procedure is non-constructive. The details of the proof, including the definition of volume, are given in section 7.

6 EXPTIME Hardness

In this section we prove the EXPTIME hardness results listed in figure 7. These hardness results are based on a lemma of possibly independent interest. We start with the notion of a "disjunctive pattern".

Definition: A disjunctive pattern is an expression of form $p_1 \ \vee\ \ldots\ \vee p_n$ where each p_i is a first order term possibly containing variables. A disjunctive pattern is linear if each p_i is linear, i.e., any given variable occurs at most once in p_i. A term t is said to ***satisfy*** a disjunctive pattern $p_1 \ \vee\ \ldots\ \vee p_n$ if t is a substitution instance of some p_i.

For example we have that $c(a,\ x) \vee c(x,\ b)$ is a linear disjunctive pattern and $c(a,\ d)$ and $c(d,\ b)$ both satisfy this pattern.

Definition: A disjunctive pattern problem is a set $D_1, \ldots, D_n$ of disjunctive patterns plus a first order term p possibly containing variables. The disjunctive pattern problem is solvable if there exists a substitution instance t of p such that every subterm of t satisfies every D_i.

Lemma: Determining satisfiability of a disjunctive pattern problem is EXPTIME hard.

The proof is by reduction of linear space alternating Turing machines. A configuration of a machine with n tape positions can be represented by a tuple $\langle s,\ u_1,\ \ldots,\ u_{n+1}\rangle$ where s is a machine state, each u_i is either the special constant H (for head) or a constant representing a tape symbol. It is EXPTIME hard to determine whether a given initial machine configuration has an accepting computation. We use terms to represent proofs that a given configuration is accepting. There are two kinds of proofs depending on whether the machine sate is universal or existential. For existential states the proof terms have the form $E(s,\ u_1,\ \ldots,\ u_{n+1},\ p)$ where $\langle s,\ u_1,\ \ldots,\ u_{n+1}\rangle$ is a machine configuration and p is a proof for one of the two possible successor configurations. For universal states proofs have the form $A(s,\ u_1,\ \ldots,\ u_{n+1},\ p_1,\ p_2)$ where p_1 and p_2 are proofs for the two possible successor configurations. We can assume that successor states of universal sates are existential and that successor states of existential states are universal. We can also assume that all accepting states are existential. We translate a given machine into a set of disjunctive patterns.

First we construct a set of "transition patterns". For each right-moving universal state s, each head position i, and each pair a_1, a_2 of tape symbols there is a transition pattern of the form

$$\begin{aligned}&A(s,\ x_1,\ \ldots,\ x_{i-2},\ a_1,\ H,\ a_2,\ x_{i+2},\ldots,\ x_{n+1},\\ &\quad E(w_1,\ y_1,\ \ldots,\ y_{i-2},\ a_1,\ b_1\ H,\ y_{i+2},\ldots,\ y_{n+1},\ y_{n+2}),\\ &\quad E(w_2,\ z_1,\ \ldots,\ z_{i-2},\ a_1,\ b_2\ H,\ z_{i+2},\ldots,\ z_{n+1},\ z_{n+2}))\end{aligned}$$

Where w_1, b_1, w_2, and b_2 are derived from the transition table of the machine. Note that this pattern is linear and does not enforce the constraint that the tapes of the sub-configurations match the tape of the top level configuration. This constraint is expressed elsewhere. Left moving symbols are treated analogously. For each right moving existential state s, head position i, tape symbols a_1, a_2, possible successor state w, and appropriate symbol b derived from the transition table we have a transition pattern of the form

$$\begin{aligned}&E(s,\ x_1,\ \ldots,\ x_{i-2},\ a_1,\ H,\ a_2,\ x_{i+2}, \ldots,\ x_{n+1},\\ &\quad A(w_1,\ y_1,\ \ldots,\ y_{i-2},\ a_1,\ b\ H,\ y_{i+2}, \ldots,\ y_{n+1},\ y_{n+2},\ y_{n+3}))\end{aligned}$$

Left moving existential states generate similar patterns. For each accepting state s we have the "transition pattern"

$$E(s,\ x_1,\ \ldots,\ x_{n+1},\ x_{n+2})$$

The first disjunctive pattern is the disjunction of all constants (representing machine states and tape symbols) and all transition patterns — every term is required to be either a constant or an instance of a transition pattern.

Now we must ensure that machine configurations are preserved as one descends into the structure of the proof terms. For each tape position i we construct the disjunction of all constants plus all patterns of the following forms.

$$\begin{array}{ll}A(& s,\ x_1,\ \ldots,\ x_{i-1},\ a,\ x_{i+1}, \ldots,\ x_{n+1},\\ & E(w_1,\ y_1,\ \ldots,\ y_{i-1},\ a,\ y_{i+1}, \ldots,\ y_{n+1},\ y_{n+2}),\\ & E(w_2,\ z_1,\ \ldots,\ z_{i-2},\ a,\ z_{i+1}, \ldots,\ z_{n+1},\ z_{n+2}))\\ E(& s,\ x_1,\ \ldots,\ x_{i-1},\ a,\ x_{i+1}, \ldots,\ x_{n+1},\\ & A(w_1,\ y_1,\ \ldots,\ y_{i-1},\ a,\ y_{i+1}, \ldots,\ y_{n+1},\ y_{n+2})),\\ A(& s,\ x_1,\ \ldots,\ x_{i-1},\ H,\ x_{i+1}, \ldots,\ x_{n+3})\\ A(& s,\ x_1,\ \ldots,\ x_{i-2},\ H,\ x_{i}, \ldots,\ x_{n+3})\\ A(& s,\ x_1,\ \ldots,\ x_{i},\ H,\ x_{i+2}, \ldots,\ x_{n+3})\\ E(& s,\ x_1,\ \ldots,\ x_{i-1},\ H,\ x_{i+1}, \ldots,\ x_{n+2})\\ E(& s,\ x_1,\ \ldots,\ x_{i-2},\ H,\ x_{i}, \ldots,\ x_{n+2})\\ E(& s,\ x_1,\ \ldots,\ x_{i},\ H,\ x_{i+2}, \ldots,\ x_{n+2})\end{array}$$

The disjunction of these patterns implies that the ith position behaves correctly. The final set of disjunctive patterns consists of the transition disjunction and, for each position i, the above disjunction. The top level term is $E(s_0, H, a_1,\ \ldots,\ a_n,\ x)$ where $\langle s_0, H, a_1,\ \ldots,\ a_n\rangle$ is the initial configuration of the Turing machine. ■

Now we will argue that we can assume without loss of generality that the patterns involve only a single binary constructor p and a single constant a. Starting with an arbitrary set of disjunctive patterns and we represent a constructor application $c(e_1,\ \ldots,\ e_n)$ by $p(e_c,\ \langle e_1,\ \langle \ldots\ \langle e_{n-1},\ e_n\rangle\rangle\rangle)$ where e_c is a term built from p and a which is unique to the constructor c and $\langle e_1,\ e_2\rangle$ abbreviates $p(p(e_{new},\ e_1),\ e_2)$ where e_{new} is a term distinct from all e_c and which clashes with $p(e_c\ x)$ for all e_c and where e_{new} is used to flag syntax nodes which are introduced by this translation. The syntax nodes $\langle e_1,\ e_2\rangle$ introduced by the translation need not satisfy (the translation of) the given disjunctive patterns. However, we can add $p(e_{new},\ x)$ and $p(p(e_{new},\ x)\ y)$ to each disjunctive pattern so that the newly introduced syntax nodes trivially satisfy all patterns. We then have that translation of the disjunctive pattern problem is solvable if and only if the original is solvable.

We now prove that determining SBA-safety is EXPTIME hard by reducing satisfiability of disjunctive pattern problems to SBA-safety. Consider a disjunctive pattern problem consisting of disjunctive patterns $D_1\ \ldots,\ D_n$ and top level term p. We need to determine if there is a ground substitution instance of p every subterm of which satisfies every D_i. For technical reasons described below we add the following special disjunctive pattern which we denote by D_0.

$$a \vee p(a,\ a) \vee p(p(x,\ y),\ a) \vee p(a,\ p(x,\ y)) \vee p(p(x,\ y),\ p(z,\ w))$$

Since we are only concerned with terms constructed from p and a this final pattern is tautological, i.e., every term satisfies it. The need for this tautological pattern is discussed below. For each disjunctive pattern D_i we define f_i to be the following term where $\perp$ denotes the divergent term $((\lambda x.(x\ x))(\lambda x.(x\ x)))$ and D_i is the disjunction $q_1\ \vee\ \ldots\ \vee\ q_n$.

$$\begin{array}{rl}f_i = \lambda x.\mathtt{AMB}(& \mathtt{CASE}\ x\ \mathtt{of}\ z_1\ as\ q_1 :\ z_1,\ w : \perp\\ & \vdots\\ & \mathtt{CASE}\ x\ \mathtt{of}\ z_n\ as\ q_n :\ z_n,\ w : \perp)\end{array}$$

For any first order term t we have that $f_i(t) = t$ if t satisfies D_i and otherwise $f_i(t) = \perp$ (no error is ever generated). Note that this construction works under any of the four semantics of the case statement. Under the set based abstraction the use of AMB is not essential — for the purposes of analysis we can replace AMB by $\lambda xy.((\lambda I.(\lambda zw.w\ (I\ x)\ (I\ y)))\ \lambda v.v)$. We now let F be the following procedure that filters its input over all patterns.

$$F = \lambda x.\ (f_0\ (f_1\ \ldots\ (f_n\ x)\ldots))$$

Here f_i is the filter for pattern D_i and note that we have included f_0 which filters for the tautological disjunctive pattern D_0.

Next we construct a term that generates those terms t such that every subterm of t satisfies every disjunctive pattern. We let Y be the traditional recursion combinator $\lambda f((\lambda h.(f\ (h\ h)))(\lambda h.(f\ (h\ h))))$.

$$\mathtt{TERM} = (Y \lambda t.\mathtt{AMB}(a,\ (F\ p(t,\ t))))$$

This should be viewed as a recursive definition of the set of possible values of the variable t. The values of t are precisely the values of $\mathtt{AMB}(a,\ (F\ p(t,\ t)))$. We can assume that a satisfies all disjunctive patterns and all other terms are filtered by F so they must also satisfy all disjunctive patterns. The inclusion of the tautological pattern D_0 prevents the creation of incomplete values. Note that any constructor application returned by the tautological filter f_0 is either the constant a or an expression of the form $p(e_1,\ e_2)$ where e_1 and e_2 are constructor applications. If D_0 were not included then we could get values of the form $p(e_1,\ e_2)$ where e_1 and e_2 where suspended computations. But D_0 ensures that there are no suspended computations as arguments to the top level constructor. This property is satisfied by every value of t. Hence values of t in $p(t,\ t)$ also have the property that no suspended computations appear near the root. In fact, it is easy to show by induction on the process of generating values of t that any value which is a constructor application has no suspended computations whatsoever. It is important that suspended computations inside the data structures are avoided to ensure that the term can be completed in a way that satisfies all the constraints.

Finally we construct the following term.

$$((\mathtt{CASE\ TERM\ of}\ p : a,\ x : \perp)\ a)$$

Here p is the target term of the given disjunctive pattern problem — the problem is determining if there an instance of p such that every subterm of that instance satisfies every disjunctive constraint. This term is SBA-safe if and only if the given disjunctive pattern problem is unsolvable.

We have now shown that SBA-safety is EXPTIME-hard for deep patterns with "as" under any semantics of case statements. It also possible to modify the construction to use non-overlapping patterns. Consider any linear term p, i.e., term where each variable in the term occurs at most once. When working with a finite set of constructor symbols the set of terms that *clash* with p can be expressed by a number of patterns linear in the size of p. For example, the set of terms clashing with $p(p(x, y), p(z, w))$ is the set of instances of a, $p(a, x)$ and $p(x, a)$. The patterns covering the complement of a given pattern can be made to be disjoint — no two of them share a common instance. To modify the construction to use disjoint patterns we simply replace the case statements of the form

$$\texttt{CASE}\ x\ \texttt{of}\ z_j\ \textit{as}\ q_j\ :\ z_j,\ w : \perp$$

in the definition of the filter f_i by

$$\texttt{CASE}\ x\ \texttt{of}\ z_j\ \textit{as}\ q_j\ :\ z_j,\ w_{j,1} : \perp,\ \ldots,\ w_{j,m} : \perp$$

where $w_{j,1}, \ldots, w_{j,m}$ are patterns covering the complement of q_j. A similar transformation can be applied to the case statement in the top level term. Now the construction is both insensitive to the semantics of case and uses only non-overlapping patterns.

When using sequential case with overlapping patterns we can also eliminate the use of "as". We simply replace each case statement

$$\texttt{CASE}\ x\ \texttt{of}\ z_j\ \textit{as}\ q_j\ :\ z_j,\ w : \perp$$

in the definition of the filter f_i by

$$\texttt{CASE}\ x\ \texttt{of}\ w_{j,1} : \perp,\ \ldots,\ w_{j,m} : \perp,\ z_j : z_j$$

where $w_{j,1}, \ldots, w_{j,m}$ are patterns covering the complement of q_j. Note that the EXPTIME-hardness is independent of whether the case statements are strict, i.e., generate an error when no pattern matches.

7 A Non-constructive Algorithm

We now return to the escape problems defined at the end of section 5. Recall that an escape problem consists of a pair $\langle X, G\rangle$ where X is a nonterminal symbol and G is a regular grammar plus a set of patterns (terms) $p_1, \ldots, p_n$. The escape problem is satisfiable if some term generated by X under G escapes (clashes with) all the p_i. An escape problem is called linear if all p_i are linear and disjoint if not two p_i share a common instance. Here we prove that satisfiability of linear disjoint escape problems can be determined in $O(|G||p_1, \ldots, p_n|^4 \log |p_1, \ldots, p_n|)$ time where $|G|$ is the sum over all production in G of the size of the production, i.e., the number of arguments to the constructor on the right hand side of the production, and $|p_1, \ldots, p_n|$ is the sum over expressions in the patterns p_i (including subexpressions) of the arity (number of arguments) in that expression.

Note that in the escape problems generated in determining SBA-safety we have that $|G|$ can be at most $O(n^2)$ where n is the size of the input term. Also, $|p_1, \ldots, p_n|$ can be at most $O(n)$. So, as a function of program size we get a running time which is $O(n^6 \log n)$. Of course in practice this analysis is applied to each individual case statement separately. The complexity of case statements tends to be bounded so the cost of the escape analysis should be linear in the size of the program in practice.

The basic idea behind the decision procedure is simple. Each pattern can be associated with a number which intuitively counts the number of instances of that pattern. We might call this the "volume" of the pattern. The total volume of the terms generated by the grammar can also be computed. Since the patterns are disjoint the volume covered by the patterns is just the sum of the volume of each individual pattern. The escape problem is solvable if and only if the sum of the volume of the patterns is strictly less than the volume of the grammar. Note that this analysis can determine that an escape is possible but provides no direct way of finding an escaping term.

Before formalizing this argument in general, it is instructive to consider a simple special case. Consider lists of n Boolean values. A Boolean value can be modeled as being one of two constant symbols T and F. A list of n Boolean values can be represented using a binary constructor function by a term of the form $p(x_1, p(x_2, \ldots x_n))$ where each x_i is either T or F. It is easy to write a simple grammar involving $O(n)$ productions which generates exactly the set of 2^n lists of n Boolean values. Now we consider patterns where each pattern specifies some subset of the Boolean values. In this case the volume of the grammar is 2^n and the volume of each pattern is 2^m where m is the number of Boolean values left unspecified by the pattern. The patterns must be disjoint (an easily verified condition) so an escape is possible if and only the sum of the volumes of the patterns is strictly less than 2^n. If an escape is possible then we can find an escaping value by testing the Boolean values one at a time to see if an escape remains possible when that value is restricted to, say, T. Finding the witness involves running the escape test n separate times. The escape test is fundamentally non-constructive.

Now we consider the general case — nested patterns and arbitrary grammar. Let $p_1, \ldots, p_n$ be linear non-overlapping patterns and let G be a regular grammar where the patterns and the grammar involving only constants and a single binary constructor. First we rewrite both the patterns and the grammar so that they use only constants and a single binary constructor. Under the measure of grammar and pattern size given above this transformation increases the size of both the grammar and patterns by at most a linear factor. Next we convert both the patterns and the grammar to use a three argument constructor where the first argument is always a constant symbol. We replace each production in G of the form $X \to p(Y, Z)$ by $X \to p(a, Y, Z)$ where a is a constant unique to that production. We then add a fresh variable as the first argument to each use of the binary constructor in each pattern. The new patterns cover the new grammar if and only if the original patterns cover the original grammar. The advantage of the new grammar is that parse trees are unique — a given term has at most one parse. This means that counting terms is the same as counting parses.

A nonterminal will be called "empty" if it does not generate any term. We now remove all productions involving empty nonterminals. To determine which nonterminals are empty we run the following inference rule on the grammar where the assertion $\exists X$ indicates that the nonterminal X is nonempty.

$$\frac{X \to c}{\exists X} \qquad \frac{X \to c(Z,\ W),\ \exists Z,\ \exists W}{\exists X}$$

These inference rules have $O(|G|)$ prefix firings where $|G|$ is the number of productions in (the binary constructor grammar) G. Since inferential closures can be computed in time linear in the number of prefix firings [11] we can compute the inferential closure of these rule in $O(|G|)$. Hence in linear time we can identity and remove all empty nonterminals.

A "position" is a (possibly empty) finite sequence of integers indicating how one should descend into a term. For example, the position 2.1 — the place one gets by taking the second argument and then the first argument of that — in the term $c(a,\ g(b))$ is occupied by the constant b. The position 2 in this term is an application of the constructor g. The root position is occupied by an application of c. We say that the position 3 in the term $c(a,\ g(b))$ is occupied by $\perp$. In general, a ground term can be viewed as a function from positions to constructors plus $\perp$ — if t is a term and p is a position then t_p is either the constructor applied at position p in t or $\perp$ if the position p is not occupied in t.

Let P be a set of positions. We will consider two terms to be P-equivalent if they agree on all positions in P. The number of P-equivalence classes is at most $|C+2|^{|P|}$ where C is the number of constant symbols (not counting the three place constructor). Let P_0 be the set of positions occupied by a constructor symbol in at least one of the given patterns. Note that if s and t are P_0-equivalent then they are either both instances or both non-instances of each pattern. So there exists a term generated by the grammar but not covered by the patterns if and only if there exists a P_0-equivalence class of terms generated by the grammar and not covered by the patterns. Our procedure will count P_0-equivalences classes. We use the number of P_0 equivalence classes as a measure of volume. The patterns cover the grammar if and only if the number of P_0-equivalence classes in sets of the form $p_i \cap G$, where p_i is one of the given patterns, equals the total number of P_0-equivalences classes of terms generated by G.

For any nonterminal X in G, any linear pattern p, and any upward-closed set of positions P, we define $V[X,\ p,\ P]$ to be the number of P-equivalence classes of terms generated by G and which are instances of p. For the grammar constructed above where parses are unique and where every nonterminal is nonempty the function V satisfies the following conditions.

- $V[X,\ c,\ P] = \begin{cases} 1 \text{ if } X \to c \\ 0 \text{ otherwise} \end{cases}$
- If P is nonempty, i.e., if P contains the root position, then
$$V[X,\ p(x,\ q,\ w),\ P] =$$
$$\sum_{X \to p(a,\ Z,\ W)} V[Z,\ q,\ \{r : 2.r \in P\}]V[W,\ w,\ \{r : 3.r \in P\}].$$
- If P is nonempty, i.e., if P contains the root position, then
$$V[X,\ z,\ P] = \sum_{X \to c} 1 \quad +$$
$$\sum_{X \to p(a,\ Z,\ W)} V[Z,\ z,\ \{r : 2.r \in P\}]V[W,\ z,\ \{r : 3.r \in P\}].$$
- $V[X,\ z\ ,\{\}] = 1.$

Now let X be the distinguished nonterminal appearing in the input pair $\langle X,\ G\rangle$ of the escape problem. The volume of the language represented by $\langle X,\ G\rangle$ is $V[X,\ z,\ P_0]$. The escape problem is solvable if this grammar volume is strictly larger than the sum over the given patterns p_i of the pattern volume $V[X,\ p_i,\ P_0]$. The above equations can be used to compute these quantities. During the recursive calculations we get expressions of the form $V[X,\ p,\ P]$ where P is a smaller set of positions. But we only encounter expressions of the form we $V[X,\ c,\ P]$ and $V[X,\ p(z,\ q,\ w),\ P]$ where P is nonempty. When we "run off" the bottom of the pattern we encounter expressions of the form $V[X,\ z,\ P]$. Computing the value of these expressions involves computing values of the form $V[X,\ z,\ Q]$ for smaller position sets Q until the set of positions becomes empty at which case we can use the final equation above as a base case.

To do the complexity analysis let $|G|$ be the number of productions in G. Let $|p_1 ... p_n|$ be the total number of occurrences of expressions in $p_1,\ \ldots,\ p_n$. The computation cost is dominated by the cost of the multiplications in the above conditions. Each multiplication corresponds to a unique triple of a production in G, a pattern occurring in $p_1,\ \ldots,\ p_n$ (possibly as a subpattern), and a position in P. So the total number of multiplications is bounded by $|G||p_1,\ \ldots,\ p_n|^2$. The integers involved in the multiplications get large so we will use a bit complexity analysis of the cost of each multiplication. We have that $V[X,\ p,\ P] \leq (|C|+2)^{|P_0|}$ for all the calculations involved. Hence the number of bits in the binary notation for the numbers is bounded by $\log((|C|+2)^{|P_0|})$ which equals $P_0 \log(|C|+2)$. Multiplication takes quadratic time so each multiplication can take $O(|p_1,\ \ldots,\ p_n|^2 \log |p_1,\ \ldots,\ p_n|)$ time. So the overall time complexity is $O(|G||p_1,\ \ldots,\ p_n|^4 \log |p_1,\ \ldots,\ p_n|)$.

8 Conclusions

This paper contains two significant results. The first is a unification of various notions of set-based analysis into a single definition which can be applied to any language whose operational semantics can be defined by environment evaluation. The second is a systematic treatment of the complexity of determining SBA-safety as a function of the semantics and syntactic complexity of case statements in a functional language.

We leave some open problems. First there, is the problem of working out the worst case complexity of determining polyvariant SBA-safety for shallow case statements. We now have a lower bound The second major open problem is harder to state technically. It is the in-practice efficiency of determining SBA-safety. It seems that in practice the presence of deep case statements is not a problem. Monovariant programs, even with deep patterns, can apparently be analyzed in cubic time or better in practice. Worst case complexity analysis aside, it seems that polyvariance is the more significant performance issue in practice.

References

[1] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, June 1993.

[2] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *ACM Symposium on Principles of Programming Languages*, pages 163–173. Association for Computing Machinery, 1994.

[3] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Also in Proc. POPL91.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[5] Thom Frühwirth, Ehud Shapiro, Moshe Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 75–83. IEEE Computer Society Press, 1991.

[6] N. Heintze. Set based analysis of ml programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.

[7] N. Heintze. Control flow analysis and type systems. In *Second Static Analysis Symposium (SAS95)*, pages 189–206. Springer Verlag, 1995. Lecture Notes in Computer Science 983.

[8] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *ACM Symposium on Principles of Programming Languages*, pages 197–209. Association for Computing Machinery, 1990.

[9] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3), March 1995. Also in PLDI-94.

[10] S. Jagannathan and A. Wright. Effective flow analysis for avoiding run time checks. In *Second Static Analysis Symposium (SAS95)*, pages 207–224. Springer Verlag, 1995. Lecture Notes in Computer Science 983.

[11] David McAllester. Inferring recursive data types. http://www.ai.mit.edu/people/dam/rectypes.ps.

[12] P. Mishra. Towards a theory of types in prolog. In *International Symposium on Logic Programming*, pages 289–298. IEEE, 1984.

[13] P. Mishra and U. S. Reddy. Declaration-free type checking. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 7–21. ACM, 1985.

[14] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *POPL95*, pages 367–378, 1995.

[15] J. Palsberg and S. Smith. Constraned types and their expressivelness. *Transactions on Programming Languages and Systems.* to appear.

[16] Olin Shivers. Data flow analysis and type recovery in scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 47–87. MIT Press, 1991.