On the Complexity of Set-Based Analysis (1288.232)

David McAllester, Nevin Heintze
AT&T Research
600 Mountain Ave
Murray Hill N.J. 07974
dmac@research.att.com
(908)582-5412
http://www.ai.mit.edu/people/dam/dam.html

Abstract: We formally define the set-based abstraction of any language whose operational semantics can be defined by environment evaluation. The Aiken-Wimmers soft type system precisely corresponds to this set-based abstraction under their operational semantics. The Heintze set-based analysis is precisely this set-based abstraction under a different operational semantics. In general, set-based abstraction determines a notion of SBA-safety. Aiken-Wimmers typability is a form of SBA-safety. SBA-safety is decidable in most cases. For monovariant functional programs with shallow case statements SBA-safety is decidable in $O(n^3)$ time under any standard operational semantics. We show here that if we allow deep patterns in the case statements of monovariant functional programs the problem of determining SBA-safety becomes complete for deterministic exponential time (under any standard operational semantics). We also systematically characterize the complexity of determining SBA-safety for monovariant programs as a function of the semantics and syntactic complexity of case statements.

1 Introduction

Recently there has been considerable interest in soft typing — type inference for languages with no type declarations of any form. The programming language ML provides automatic type inference for procedures but requires type declarations for data constructors. Set-based analysis for languages without datatype declarations has been studied both in the context of logic programming [12, 7, 4, 8] and in the context of functional programming [13, 1, 2, 5]. Many different analyses have been developed. These approaches all share a common framework in that they assign sets (or types) to program expressions while ignoring correlations. For example, consider a variable x that can take on any integer value. Although the expression $(x+1)-x$ is always equal to 1, any analysis which ignores correlations assumes that this expression can take on any integer value. Whenever two expressions can each independently take on any integer value their difference is assumed to take on any integer value. These analyses also generally ignore the correlations between the input and output of user defined procedures — each procedure is characterized by input and output sets.

Although the various approaches to soft typing seem to share certain basic assumptions, considerable energy has been put into developing appropriate set-based analyses for various languages. Here we give a general definition of the set-based abstraction of any language whose semantics can be defined by an evaluation process which operates on pairs of expressions and environments, i.e., by an environment evaluator. Our general definition of set-based analysis yields the most accurate analysis possible while ignoring correlations. This general definition allows both a systematic study of a wide variety of set-based analyses and a precise formulation of complexity theoretic questions concerning the difficult of set-based analysis for a wide variety of language features.

The general definition of set-based analysis yields a general notion of `SBA`-safety where `SBA`-safety implies safety, i.e., that there will not be a run time error. Here we address the computational complexity of determining `SBA`-safety in a functional language with case statements. In all the cases we consider `SBA`-safety is decidable in deterministic exponential time (for monovariant programs). But the complexity of determining `SBA`-safety is sensitive to both the syntactic complexity and the operational semantics of the 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 semantics and syntactic complexity of case statements (in monovariant programs).

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 [5]. The analysis is also similar to that of Shivers [16] or Jagannathan and Write [9] 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, 6, 15]. These connections are explored more deeply in [10].

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.

Our general 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 this 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 [10] that even for first order programs with shallow case statements and restricted to procedures of at most one argument, determining SBA-safety is PSPACE-hard. Presumably the problem gets even harder in the worst case when we allow deep patterns. But we leave the worst case complexity of determining SBA-safety for general polyvariant programs as an open problem. Another important open problem is the in-practice efficiency of determining polyvariant SBA-safety. Constrained types seem to provide a promising practical approach [1, 2, 15].

The technical content of this paper is given in the figures and their captions. The body of the paper is used primarily to give examples of the concepts and analyses defined in the figures. Due to space restrictions many of the proofs have been omitted here. However, proofs can be found in the full paper available on the web [11].

2 Results

The syntax and semantics of a simple operational language with case statements is given in figure 1. The semantics is given as a set of reduction rules which can be used to reduce a term to a "value" (which is another term). One of the classical goals of static analysis is to provide a static guarantee that certain errors can not occur at run time. Run time errors are modeled in the reduction rules by rules which convert certain terms to the constant ERROR. A given term e is called run time safe if there is no term w such that e reduces to w and w contains ERROR. We will define a notion of SBA-safety corresponding to the operational semantics defined by the reduction rules. SBA-safety will be decidable and will imply run time safety. Furthermore, determining whether a given term is SBA-safe will require an appropriate set-based analysis of the term, i.e., an assignment of of a set of values, i.e., a type, to each subterm of e. In general it seems to be impossible to determine SBA-safety without computing a set-based analysis so it seems that the complexity of determining SBA-safety can, for all practical purposes, be assumed to be the same as the complexity of performing set based analysis, i.e., computing a "type" for each program node.

Safety under arbitrary reduction is a strong safety property. Virtually all operational semantics correspond to some form of restriction on reduction. Safety under arbitrary reduction implies safety under any such operational semantics. Although the analysis derived from an arbitrary reduction semantics can be less accurate than an analysis which exploits knowledge of the reduction strategy of a particular implementation, exploitation of a particular reduction strategy seems to have undesirable consequences. For example, let WRONG be the expression $(a\ a)$ where a is a constant and let $\perp$ be the expression $((\lambda h.(h\ h))\ (\lambda h.\ (h\ h)))$. Under lazy evaluation we have that $c(\texttt{WRONG})$ is SBA-safe and under strict evaluation we have that $(c(\perp)\ u)$ is SBA-safe. If we evaluate operators before arguments then $(\perp\ \texttt{WRONG})$ is SBA-safe and if we evaluate arguments before operators then $(\texttt{WRONG}\ \perp)$ is SBA-safe. Under most operational semantics $\lambda x.\texttt{WRONG}$ is SBA-safe. However, under an arbitrary reduction semantics all of these terms become unsafe, and hence SBA-unsafe.

Although there seems to be good reasons for basing one's analysis on an arbitrary reduction semantics, our framework allows one to easily construct analyses corresponding to particular reduction

$$e ::= \quad x \mid (e_1\ e_2) \mid \lambda x.e \mid c(e_1,\ \ldots,\ e_n)$$
$$\mid \mathtt{CASE}(e_0\ \mathtt{of}\ p_1 : e_1,\ \ldots,\ p_n : e_n) \mid \mathtt{AMB}(e_1,\ e_2)$$

$$p ::= \quad x \mid c(p_1,\ \ldots,\ p_n) \mid x\ as\ p$$

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

Figure 1: Terms and term reduction rules. 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. We abbreviate the case rules as s, p, and e (for serial, parallel, and error respectively). We take the rewrite relation $\rightarrow_{p,e}$ to be the least transitive reflexive relation generated by the rules β and β-`error` plus p and e, i.e., the rules `parallel case` and `error case` (and where rewriting is allowed at arbitrary subterms). The rewrite relations $\rightarrow_p$, $\rightarrow_s$, and $\rightarrow_{s,e}$ are defined analogously. 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. Variables, applications, and case terms do not clash with patterns.

strategies. Furthermore it seems that the choice of reduction strategy does not affect the computational complexity of determining SBA-safety. We believe that all of the complexity results given here for unconstrained reduction hold for the various standard reduction strategies as well. As noted above, an analysis based on the particular reduction strategy used in a given implementation is in some sense more accurate than the analysis based on the unconstrained reduction semantics.

We consider four different semantics for case statements. Case statements can be serial or parallel. The serial case reduction rule generates the reduction $\mathtt{CASE}(c(a)\ \mathtt{of}\ c(x): x,\ y: y) \rightarrow_s a$ but not the reduction $\mathtt{CASE}(c(a)\ \mathtt{of}\ c(x): x,\ y: y) \rightarrow_p c(a)$; 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 1. In all cases the order of reduction is unconstrained.

Our general definition of set-based abstraction applied to environment evaluation, i.e., to any semantics where values are computed for a given expression under a given binding environment. In order to construct a set-based analysis for the arbitrary reduction semantics we first recast that semantics as environment evaluation. This is done in figures 2 and 3. The figures contain inference rules which should be understood as bottom-up (forward chaining) rules which start from a single premise of the form $\mathtt{EVAL}(e,\ 0)$ where 0 is the empty environment. Assertions of the form $\mathtt{EVAL}(u, \rho)$ should be viewed as request for the evaluation of the program expression u in the context of the environment ρ. The rules generate assertions of the form $\langle u,\ \rho\rangle \rightarrow v$ which should be viewed as stating that the evaluation of u in binding environment ρ produces value v. Since the language contains the nondeterministic primitive AMB we must expect that a given pair $\langle u,\ \rho\rangle$ can have more than one value. Because we want to model arbitrary reduction we also get values in many different stages of "unfinished" computation.

Suppose we seed the inference rules with the initial assertion $\mathtt{EVAL}((\lambda x.x\ a),\ 0)$, where a is a constant (a constructor of no arguments). Then by BETA$'$, it follows that $\mathtt{EVAL}(\lambda x.x,\ 0)$ and $\mathtt{EVAL}(a,\ 0)$. By IDENT, it follows that $\langle \lambda x.x,\ 0\rangle \rightarrow \langle \lambda x.x,\ 0\rangle$ and $\langle a,\ 0\rangle \rightarrow \langle a,\ 0\rangle$. By CONS we get that $\langle a,\ 0\rangle \rightarrow a$. Hence, by BETA, $\langle(\lambda x.x\ a),\ 0\rangle \rightarrow \langle x,\ [x := a]\rangle$. By CALL we get $\mathtt{EVAL}(x,\ [x := a])$. By VAR, $\langle x,\ [x := a]\rangle \rightarrow a$. And finally, by RETURN we get $\langle(\lambda x.x\ a),\ 0\rangle \rightarrow a$.

A fundamental fact of environment evaluation is that in all generated assertions of the form $\mathtt{EVAL}(u,\ \rho)$ or $\langle u,\ \rho\rangle \rightarrow v$ we have that u is a subterm of the input. The fundamental idea behind set-based abstraction is associate each subterm of the input with a set of possible values and ignore correlations between the value of the expression and the value of the variables in the environment. This can be done by adding the single inference rule in figure 5 to any environment evaluator. This inference rule defines SBA-safety for any language whose operational semantics can be defined by environment evaluation (see the caption of figure 5).

SBA-safety is decidable in most cases. Figure 6 gives a systematic account of the complexity of determining SBA-safety as a function of the semantics and syntactic complexity of case statements.

The decidability of SBA-safety (in most cases) arises from the fact that the SBA rule allows one to think of each subexpression u of the input as being associated with a set of values. The SBA rule states that any call to the environment evaluator on the expression u can return any value in the set associated with u independent of the environment passed in the call. The simplest type of SBA analysis, i.e., decision procedure for SBA-safety, makes the assumption that all case statements are

BETA

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

CASE

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

CONS

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

AMB

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

BETA′

$$\frac{\mathtt{EVAL}((f\ w),\ \rho)}{\mathtt{EVAL}(f,\ \rho),\quad \mathtt{EVAL}(w,\ \rho)}$$

CASE′

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

CONS′

$$\frac{\mathtt{EVAL}(c(e_1,\ \ldots,\ e_n),\ \rho)}{\mathtt{EVAL}(e_1,\ \rho),\ \ldots,\ \mathtt{EVAL}(e_n,\ \rho)}$$

VAR

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

CALL

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

RETURN

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

IDENT

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

Figure 2: Environment evaluation rules defining the semantics of lazy evaluation. 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. For example $\langle \lambda x.x,\ 0\rangle$ and $\langle (f\ w),\ 0\rangle$ are values where 0 denotes the empty substitution. 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. Suspended computations other than procedures never clash. 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. A value will be called *complete* if it does not contain suspended computations other than procedures. If we had restricted v to range only over complete values then these same rules would have defined strict evaluation.

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

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

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

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

Figure 3: The rules for detecting errors. The rules SYM1 and SYM2 cause symbolic evaluation of the the bodies of procedures and the branches of case statements even if the procedures are never applied and the branches are never taken. For any pattern p we define $\rho[p := ?]$ to be a substitution identical to ρ except that variables occurring in p are bound to "fresh" variables, i.e., ones not occurring free in values in ρ. Suspended computations of the form $\langle u,\ \rho[x := ?]\rangle$ represent open terms. We define $\vdash_{s,e}$ to be the inference relation defined by the rules in figure 2 plus the above rules. Figure 4 defines an invariant relating the inference relation $\vdash_{s,e}$ and the reduction relation $\to_{s,e}$. This invariant implies that $\mathtt{EVAL}(e,\ 0) \vdash_{s,e} \mathtt{ERROR}$ if and only if $e \to_{s,e} w$ where w contains ERROR. The relation $\vdash_{p,e}$ is defined by dropping the clash antecedent in the rule CASE in figure 2. The relations $\vdash_s$ and $\vdash_p$ are defined by dropping the rule CERR above. These inference relations bear the same relationship to the corresponding rewrite relation that $\vdash_{s,e}$ bears to $\to_{s,e}$.

CONS
$$\frac{\langle e_1,\ \rho\rangle \Rightarrow w_1,\ \ldots,\ \langle e_n,\ \rho\rangle \Rightarrow w_n}{\langle c(e_1,\ \ldots,\ e_n),\ \rho\rangle \Rightarrow c(w_1,\ \ldots,\ w_n)}$$

APP
$$\frac{\langle f,\ \rho\rangle \Rightarrow f',\ \langle u,\ \rho\rangle \Rightarrow u'}{\langle (f\ u),\ \rho\rangle \Rightarrow (f'\ u')}$$

AMB
$$\frac{\langle e_1,\ \rho\rangle \Rightarrow w_1,\ \langle e_2,\ \rho\rangle \Rightarrow w_2}{\langle \mathtt{AMB}(e_1,\ e_2),\ \rho\rangle \Rightarrow \mathtt{AMB}(w_1,\ w_2)}$$

VAR1
$$\frac{\rho(x) \Rightarrow w}{\langle x,\ \rho\rangle \Rightarrow w}$$

VAR2
$$\frac{\rho(x) \text{ is a variable}}{\langle x,\ \rho\rangle \Rightarrow \rho(x)}$$

LAM
$$\frac{\langle u,\ \rho[x := ?]\rangle \Rightarrow w}{\langle \lambda x.u,\ \rho\rangle \Rightarrow \lambda\rho[x := ?](x).w}$$

CASE
$$\frac{\begin{array}{l}\langle u,\ \rho\rangle \Rightarrow u'\\ \langle e_1,\ \rho[p_1 := ?]\rangle \Rightarrow w_1\\ \vdots\\ \langle e_n,\ \rho[p_n := ?]\rangle \Rightarrow w_n\end{array}}{\begin{array}{l}\langle \mathtt{CASE}(u\ \mathtt{of}\ p_1 : e_1,\ \ldots,\ p_n : e_n),\ \rho\rangle\\ \Rightarrow \mathtt{CASE}\ u'\ \mathtt{of}\\ \qquad \rho[p_1 := ?](p_1) : w_1\\ \qquad \vdots\\ \qquad \rho[p_n := ?](p_n) : w_n\end{array}}$$

RED
$$\frac{\begin{array}{l}\langle u,\ \rho\rangle \to \langle s,\ \sigma\rangle\\ \langle s,\ \sigma\rangle \Rightarrow w\end{array}}{\langle u,\ \rho\rangle \Rightarrow w}$$

Figure 4: Relating environment evaluation and term reduction. We assume that environments of the form $\rho[p := ?]$ are constructed deterministically from ρ and p — multiple uses of this notation for a given ρ and p refer to the same environment. We let $\vdash_{s,e,\Rightarrow}$ be the inference relation defined by extending $\vdash_{s,e}$ with the above rules. For any value v we define the term represented by v, denoted $T(v)$ by the conditions that $T(x) = x$, $T(c(v_1,\ \ldots,\ v_n)) = c(T(v_1),\ \ldots,\ T(v_n))$ and $T(\langle u,\ \sigma\rangle) = \sigma'(u)$ where $\sigma'(x) = T(\sigma(x))$. By a "soundness argument" for the inference rules in figures 2 and 3 one can show that if $\mathtt{EVAL}(e,\ 0) \vdash_{s,e} v_1 \to v_1$ then $T(v_1) \to_{s,e} T(v_2)$. By a similar soundness argument we have that if $\mathtt{EVAL}(e,\ 0) \vdash_{s,e,\Rightarrow} \langle u,\ \rho\rangle \Rightarrow w$ then $T(\langle u,\ \rho\rangle) \to_{s,e} w$. Conversely, it is possible to show by induction on the number of rewrite steps that if $e \to_{s,e} w$ then $\mathtt{EVAL}(e,\ 0) \vdash_{s,e,\Rightarrow} \langle e,\ 0\rangle \Rightarrow w'$ where w' is α-equivalent to w. Hence there exists a tight correspondence between $\vdash_{s,e}$ and $\to_{s,e}$.

$$\frac{\begin{array}{l}\mathtt{EVAL}(u,\ \rho)\\ \langle u,\ \rho\rangle \to v\\ \mathtt{EVAL}(u,\ \gamma)\end{array}}{\langle u,\ \gamma\rangle \to v}\ \mathtt{SBA}$$

Figure 5: The set-based abstract of an environment evaluator. The first argument to EVAL in an environment evaluator is always a subexpression of the top level input. The abstraction rule states, in essence, that each subexpression of the input must be treated as having a single set of values independent of the environment. For any environment evaluation inference relation $\vdash$ we take $\vdash_A$ to be the inference relation generated by $\vdash$ plus the above abstraction rule. A term e is called SBA-safe under $\vdash$ if $\mathtt{EVAL}(e,\ 0) \not\vdash_A$ ERROR. SBA-safety is usually decidable. SBA-safety is complete for (deterministic) exponential time under any of $\vdash s$, $\vdash p$, $\vdash_{s,e}$, and $\vdash p, e$. Restricting the form of the patterns in the case statements makes SBA-safety more tractable as shown in figure 6.

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

Figure 6: 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 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 first column means we allow only shallow patterns in case statements. A minus in the second column means that we do not allow "as" in patterns. A minus in the third 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 fourth column means that we drop the clash antecedent from the rule CASE in figurefig:semantics. A minus in the last column means that we drop the rule CERR form figure 3. 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 over this space of design decisions.

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

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

AMB
$$\frac{\mathtt{EVAL}(\mathtt{AMB}(e_1,\ e_2))}{\mathtt{AMB}(e_1,\ e_2) \to e_1,\ \mathtt{AMB}(e_1,\ e_2) \to e_2}$$

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

IDENT
$$\frac{\mathtt{EVAL}(u)}{u \to u}$$

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

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

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

Figure 7: Flow rules for shallow patterns. When we only allow shallow patterns 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 $\mathtt{EVAL}(u)$ 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.

shallow. In other words all patterns are either variables, applications of constructors to variables, or of the form y *as* $c(x_1,\ \ldots,\ x_n)$. For shallow case statements we can perform set-based analysis by simply dropping the environments from the inference rules defining the environment evaluator. Evaluation assertions of the form $\mathtt{EVAL}(u,\ \sigma)$ become assertions of the form $\mathtt{EVAL}(u)$ and rewrite assertions of the form $\langle u,\ \rho\rangle \to \langle w,\ \gamma\rangle$ become rewrite assertions of the form $u \to w$. Figures 7 contains an $O(n^3)$ procedures for deciding SBA-safety when all patterns are shallow. The inference rules then generate a directed graph on the subterms of the input expression and the analysis can be viewed as a form of flow analysis.

Figure 8 contains a deterministic exponential time procedure for determining SBA-safety in the general case. The rules are similar to those of figure 7 but the space of abstract values is expanded to include intersections of program nodes with program patterns and negations of program patterns.

Proofs of the other results in figure 6 can be found on the web in the full paper [11].

3 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

BETA

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

CASE

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

CONS

$$\frac{\begin{array}{l}\mathtt{TEMPLATE}(\langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle) \\ e_1 \to v_1 \\ e_2 \to v_2 \\ c(v_1,\ v_2) \text{ matches every pattern in } \mathcal{P} \\ c(v_1,\ v_2) \text{ clashes with every pattern in } \mathcal{N}\end{array}}{\begin{array}{l}c(e_1,\ e_2) \to \langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle \\ \langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle \to c(v_1,\ v_2)\end{array}}$$

AMB

$$\frac{\mathtt{EVAL}(\mathtt{AMB}(e_1,\ e_2))}{\mathtt{AMB}(e_1,\ e_2) \to e_1,\ \mathtt{AMB}(e_1,\ e_2) \to e_2}$$

BERR

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

CERR

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

DOWN

$$\frac{\mathtt{EVAL}(e)}{\mathtt{EVAL}(u),\ u \text{ a subterm of } e}$$

TEMP

$$\frac{\begin{array}{l}\mathtt{EVAL}(c(e_1,\ e_2)) \\ \mathcal{P} \text{ and } \mathcal{N} \text{ subsets of} \\ \text{the patterns in } e \text{ starting with } c\end{array}}{\mathtt{TEMPLATE}(\langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle)}$$

TRANS

$$\frac{u \to w,\ w \to v}{u \to v}$$

IDENT

$$\frac{\mathtt{EVAL}(u)}{u \to u}$$

BIND1

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

BIND2

$$\frac{\mathtt{BIND}(x,\ v)}{x \to v}$$

BIND3

$$\frac{\mathtt{BIND}(x\ as\ p,\ v)}{x \to v,\ \mathtt{BIND}(p,\ v)}$$

Figure 8: Abstract evaluation rules for determining SBA-safety of an input term e in the general case. We can assume without loss of generality that all constructors take two arguments. Here we take the abstract values to be either subterms of the input term e or "templates" of the form $\langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle$ where $c(e_1,\ e_2)$ is a subterm of e and $\mathcal{P}$ and $\mathcal{N}$ are sets of patterns occurring in e of the form $c(p_1,\ p_2)$ or $x\ as\ c(p_1,\ p_2)$. The abstract value $\langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle$ represents the set of values of $c(e_1,\ e_2)$ that match all patterns in $\mathcal{P}$ and clash with all patterns in $\mathcal{N}$. The rules also manipulate value pairs of the form $c(v_1,\ v_2)$ where v_1 and v_2 are abstract values. There are only a singly exponential number of abstract values and pairs of abstract values. We use appropriate definitions of clash and match for abstract values. A template $\langle c(e_1,\ e_2),\ \mathcal{P},\ \mathcal{N}\rangle$ matches a pattern p if $p \in \mathcal{P}$ and clashes with a pattern q if either q starts with a constructor other than c or $q \in \mathcal{N}$. A value pair $c(v_1,\ v_2)$ matches a pattern $c(p_1,\ p_2)$ id v_1 matches p_1 and v_2 matches p_2. A value pair $c(v_1,\ v_2)$ clashes with a pattern $c(p_1,\ p_2)$ if v_1 clashes with p_1 or v_2 clashes with p_2.

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 several open problems. First there, is the problem of working out the worst case complexity of determining polyvariant SBA-safety. To our knowledge the only result in this are is the theorem that polyvariant SBA-safety for shallow case statements is decidable in exponential time and is PSPACE hard even for first order programs of arity one [10]. 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 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] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] S. Jagannathan and A. Write. 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.

[10] D. McAllester. Inferring recursive types. http://www.ai.mit.edu/people/dam/rectypes.ps.

[11] D. McAllester and N. Heintze. On the complexity of set based analysis. http://www.ai.mit.edu/people/dam/setbased.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.