Static Analysis of PostScript Code

R. Nigel Horspool and Jan Vitek

Dept. of Computer Science, University of Victoria
P.O. Box 3055, Victoria, BC, Canada V8W 3P6
nigelh@csr.uvic.ca jvitek@csr.uvic.ca

Stack-based languages, such as PostScript, present a major challenge to static analysis techniques because of their almost unlimited polymorphism. We introduce a regular expression notation that is used to represent allowed combinations of types on the stack at different points in a PostScript program. Our abstract interpretation algorithm may then be used to perform static type analysis. The analysis has applications in detecting probable errors in the PostScript code or, ultimately, in permitting full or partial compilation of portions of code.

1. Introduction

The PostScript[1] language has become prominent as a device-independent programming language used to control high-resolution graphical output devices. It is an example of a stack-based language where every operator obtains its arguments by popping values from a stack and pushes its result(s) back onto that stack. Other stack-based languages, such as Forth and the command language of the UNIX `dc` (desk calculator) program, exist. The techniques described in this paper should be applicable to most other stack-based languages too.

The PostScript language provides a challenge to static analysis techniques because it is inherently polymorphic. Not only can a function accept arguments with varying types, but a function can accept differing numbers of arguments. For example, it would be an easy matter to program a function *SumN* whose first argument is an integer *N* that specifies that *N* more numeric arguments are to be totalled and the sum returned as the function result. Thus, one example call might be

```
123  -23  100  3  SumN
```

which would leave a result of 200 on the stack. Another sample call might be

```
1.5  2.3  2  SumN
```

and that would leave a result of 3.8 on the stack. There are no type declarations in PostScript nor any declarations to indicate the number of parameters that a function expects. The only way to discover the argument structure of a function like *SumN* is to trace through and analyze its code. By observing the effect on the stack and by inference of the types of values popped from the stack, it *may* be possible to deduce a type signature for *SumN*. In general, the type inference problem for a language like PostScript is undecidable. But, as in all problems where abstract interpretation techniques are applied, it is possible to approximate and the approximate results are often adequate.

We have chosen to concentrate on type analysis. Our goal is to deduce the patterns of types for values on the stack that exist before and after each operation in a PostScript program. Although we currently handle only a subset of PostScript and our techniques are (intentionally) approximate in nature, we consider the results to be highly encouraging. Further work to extend the language subset and investigation of alternative analysis functions and alternative representations for stack states is planned. Although abstract interpretation techniques [1] have been used for type inference in functional languages and in Prolog, they have not, to our knowledge, previously been applied to a stack-based language. Stack-based languages introduce a number of interesting problems that do not have obvious solutions. Our initial implementation of a static analyzer successfully performs type inference for a subset of PostScript, even if functions accept varying numbers and types of arguments, as with the *SumN* example, above.

It might be argued that PostScript was never intended to be a normal programming language where programs are written by people. More usually, PostScript code is automatically generated by text formatting or graphics drawing software. Furthermore, the PostScript code is intended to be executed once only, rendering an image on the output device, and then discarded. Why, therefore, should we be interested in analyzing and verifying the correctness of PostScript code? Why not just execute the code and find

1. PostScript is a trademark of Adobe Systems Incorporated.

out if it works as intended? If one actually looks at the PostScript code generated by software, there is typically a fixed structure. An initial segment of code containing a standard collection of functions is defined. Following this, the actual formatted text or numeric descriptions of graphics images are supplied as arguments in calls to the previously defined functions. The functions are *not* automatically generated (formatting software is not yet that clever), they have been pre-programmed by a human programmer and they are simply copied into every PostScript file generated by the software. Since these functions will be used every time the software sends its output to the graphics device, it is vitally important that they be efficient and be correct. Static analysis can help to achieve both goals. And, although PostScript was designed as a language to be generated by software rather than people, there are indeed many programmers who use PostScript to implement graphical user interfaces (such as in the NeWS system from SUN Microsystems). These programmers should benefit from a tool that can direct their attention to possible errors in their code. A longer term goal of the research is in the development of optimization techniques and, perhaps, a form of compilation for PostScript code.

The innovative feature of our work is in the use of a restricted form of regular expression notation to describe combinations of types on the stack. These regular expressions represent particular values in the abstract domain used in the analysis. We will then show how abstract interpretation may be applied in both the forwards and the backwards directions to perform type inference on collections of PostScript functions and to derive appropriate type signatures. As a direct consequence of this analysis it is possible to identify regions of the code that are *definitely* erroneous (in the sense that if control should ever enter one of these regions then abnormal termination of the program is certain to follow). A similar analysis could also identify those regions of the program that *might* be erroneous. As a preview of the kind of analysis that we perform, consider a PostScript program that performs the following series of six actions:

1. Push a string constant.
2. Push *n* integers.
3. Push the integer *n*.
4. Invoke the *SumN* function.
5. Pop and output an integer (the result from *SumN*).
6. Pop and output a string.

In a forwards analysis of this code sequence, assuming an initially empty stack, we can derive the following sequence of stack states:

Pos	0	1	2	3	4	5	6
State	ε	S	S(I)*	S(I)*I	S(I)*	S(I)*	ε

Position *i* corresponds to the stack state immediately following step number *i* in the code sequence, with position 0 corresponding to the initial stack state. Each stack state is written as a regular expression (RE) where the rightmost symbol generated by the RE corresponds to the topmost stack element. Observe that immediately after the call to *SumN* in step 4, the analysis shows the possibility that additional integers remain on the stack below the function result. Unless the stack states retain the value of *n* and unless the analysis uses this value to track the number of times that *SumN* pops an argument integer, we cannot be sure that *SumN* pops all the integers that were supplied. However, when the analysis reaches step 6 where a string needs to be popped, the analysis deduces that the I* component of the pattern must actually be empty for the program to be correct. Backwards analysis may now be used to make the stack state descriptions more precise. Starting with the result stack state, ε, at step 6, the analysis deduces that the stack state should have been S at step 5. Then, proceeding one step further back, it deduces that the state should have been SI at step 4. (The other stack states are not changed.)

Abstract interpretation is concerned with static determination of certain dynamic properties of programs. In other words, abstract interpretation provides information on runtime properties of a program. Following the work of Cousot [3], many practical frameworks have been specified and implemented, notably in the field of functional programming to perform strictness analysis or in-place update analysis, and in PROLOG for mode and determinacy analysis. Abstract interpretation has also been used for automatic parallelization of imperative [5] and functional [6] programs. The technique relies on a mapping from the semantics of the original program with respect to a certain property into approximate semantics. Approximation is inevitable since any non-trivial analysis would be based on dynamic control flow patterns and those are uncomputable.

A correct abstract interpretation will *always* find an answer which *includes* the true answer. Equivalently, we say that the approximation is *conservative* and that the analysis procedure is guaranteed to terminate.

2. The abstract model

An interpreter for the PostScript language deals with typed values. Amongst the simple values that it must handle are integers (such as 1, 99, -23), floating-point numbers (such as 1.5, -3.0e-5), booleans (true and false), and strings (such as "abc"). The model used for static analysis represents these values by codes that denote only their types. We use I to stand for integer values, R for floating-point values, B for boolean values and S for string values. The effect of a

function or a block of code in PostScript can be understood only in term of its effect on the value stack. In the course of execution of an actual PostScript interpreter, the entire contents of the stack might, at one execution point, be

```
<5.0, "abc", -45, 123>
```

where the integer 123 is at the top of the stack. We represent this stack by the type code sequence `RSII`. Since functions and operators[2] may be polymorphic, we need to use sets of stack states when describing the effect of a function or operator on the stack. The static semantics of our Postscript subset can be expressed in terms of operations on elements of the lattice $\boldsymbol{L}(\mathbf{S}^*)$, defined as follows.

Definition Let $\mathbf{S}$ be the set of type codes and $\mathbf{S}^*$ be the set of type code sequences. $\boldsymbol{L}(\mathbf{S}^*)$ is the complete lattice formed by all the subsets of $\mathbf{S}^*$, with partial ordering $\subseteq$, a top element or greatest upper bound $\mathbf{S}^*$, a bottom element or least upper bound $\varnothing$, and meet and join operations equivalent to the set operations $\cap$ and $\cup$, respectively.

The sets that form the elements of $\boldsymbol{L}(\mathbf{S}^*)$ are potentially unbounded in size (limited only by the maximum stack depth permitted by a particular PostScript interpreter). Therefore a representation that requires an explicit enumeration of the elements of the sets is unsuitable for use by static analysis algorithms. We believe that sequences of types that occur at the top of the stack will usually follow simple patterns, and that regular expression notation is adequate to describe the patterns. For example, the type signature of the *SumN* function, above, may be written as

```
SumN: (I|R)*I → (I|R)
```

Similarly, a function that draws lines connecting a series of points on the page would possibly accept a sequence of coordinate pairs as its arguments and the stack state prior to a call to this a function could be represented by the regular expression `(RR)*`.

In the course of performing analysis by abstract interpretation, we will need to construct unions of sets of type code sequences where control flow paths merge. Intersections of sets will be required to combine results of forward and backwards analysis. Ideally, we would like to use a lattice whose elements are REs and where the glb (meet) and lub (join) operations correspond to intersection and union of the sets. Unfortunately, we have not been able to construct a suitable lattice. Full RE notation does not guarantee compact representations for sets of stack states and also provides many representations for the same stack state. Instead, we have chosen to use a simplified and highly restricted form of RE notation, as explained below.

2. This includes the standard Postscript control flow operators such as `ifelse` and `loop`.

2.1 Alternation-free regular expressions.

In principle, algorithms to test for inclusion between two regular expressions and to construct the union or intersection of two regular expressions as a new regular expression exist. In practice, however, the algorithms are non-trivial and too inefficient for use by an abstract interpretation procedure that must iterate a large number of times over sections of PostScript code. We therefore use a subset of regular expressions, a subset that we will refer to as *Alternation-Free Regular Expression* notation or AF-RE for short. The syntax of AF-REs is defined by the context-free grammar shown in Figure 1. Curly braces are grammatical notation to represent zero or more repetitions of the enclosed symbols. The partially ordered set $\mathbf{S}^\# = \mathbf{S} \cup \{\mathtt{N},\mathtt{X}\}$ represents any of the elementary type designator codes (`I`, `R`, `S`, `B`,...) augmented by two extra type codes. `N`, for *numeric*, is equivalent to the regular expression `(I | R)` while `X`, for *unknown*, is equivalent to `( I | R | S | ...)`. The partial order relation for elements of $\mathbf{S}^\#$ is defined as follows

$$\mathtt{I} \leq_t \mathtt{N},\ \mathtt{R} \leq_t \mathtt{N},\ \mathtt{N} \leq_t \mathtt{N},\ \mathtt{N} \leq_t \mathtt{X},\ \mathtt{X} \leq_t \mathtt{X}$$

$$\alpha \leq_t \mathtt{X} \text{ and } \alpha \leq_t \alpha, \qquad \alpha \in \mathbf{S}^\#$$

The relation $\leq_s$ which compares two type codes sequences of equal length is defined as the conjunction of $\leq_t$ on pairs of corresponding type codes:

$$\alpha_1....\alpha_n \leq_s \beta_1...\beta_n, \qquad \text{if } \alpha_i \leq_t \beta_i \text{ and } i \in [1, n]$$

The implicit alternations inherent in the meanings of `N` and `X` are the only forms of alternation provided in the AF-RE notation. Recalling that AF-REs are just a notation we use to describe sets of sequences of types codes, ε represents the empty sequence, the top symbol $\top$ is used to denote the set of all possible sequences, and the bottom symbol $\bot$ is the empty set (meaning that no valid stack state exists). As is conventional in regular expression notation, a superscript asterisk denotes zero or more occurrences of the preceding term and a superscript question mark denotes zero or one occurrences. Thus the following five expressions are valid instances of AF-REs.

```
(S)?RN(SIS)*    (I)*I    (RR)*I    ε    X*
```

The final expression, `X`*, denotes the universal set of type code sequences and is equivalent to the top element, $\top$. The expression ε denotes an empty stack, which is a legitimate stack state. Note that, as well as omitting

afre $\rightarrow \bot$	**conj** $\rightarrow$ ($\mathbf{S}^\#${$\mathbf{S}^\#$})*
afre $\rightarrow \top$	**conj** $\rightarrow$ ($\mathbf{S}^\#${$\mathbf{S}^\#$})?
afre $\rightarrow \varepsilon$	**conj** $\rightarrow \mathbf{S}^\#$
afre $\rightarrow$ **conj** {**conj**}	

Figure 1 AF-RE grammar.

explicit alternation, the notation does not permit nesting of expressions. For example, the expression `(I(R)*)?` is not a valid AF-RE.

2.2 Notational convention

In the remainder of the paper we adopt the following conventions: *r* and *s* denote arbitrary AF-REs, *c* and *d* are used for conjuncts that make up an AF-RE (e.g. `(I)*`), Greek letters α and β represent single type codes (elements of $\mathbf{S}^{\#}$) and ρ and σ represent sequences of type codes. Superscript hash marks are used to indicate abstraction. Juxtaposition of terms, e.g. *rs*, denotes concatenation of AF-REs.

2.3 AF-RE operations

The AF-RE notation, as presented above, still provides more then one expression to denote the same set of type code sequences. To simplify algorithms that manipulate AF-REs, it is desirable to work with a simplified subset of the AF-RE notation that restricts the number of different representations of a single set. A simplification procedure that converts an arbitrary AF-RE into a restricted AF-RE may be written as a series of rewrite rules. These rules are shown in Figure 2.

An example of simplification for `(SI)*S(IS)*` is

`(SI)*S(IS)*` $\Rightarrow$ `(SI)*(SI)*S` $\Rightarrow$ `(SI)*S`

The $\leq_a$ relation between two AF-REs denotes inclusion of the corresponding sets of type code sequences. The truth of the relationship $r \leq_a s$ for two simplified AF-REs may be tested by the rules shown in Figure 3. In these rules a sequence of zero conjuncts should be replaced by ε when appropriate. The rules may be used to derive, for example, `I` $\leq_a$ `(I)?(S)*`.

The AF-RE notation cannot describe all possible sets of type code sequences. A consequence is that, given two AF-REs *r* and *s*, AF-REs that are equivalent to $r \cup s$ or to $r \cap s$ may not exist. Algorithms that compute $r \cup s$ and $r \cap s$ must, in general, be approximate. We require such algorithms for the abstract interpretation and we are therefore forced to provide conservative approximations to the results. These approximations to $\cup$ and $\cap$, together with the $\leq_a$ relation of Figure 3, do not form a lattice where the

$(\rho)^*(\sigma)^*$	$\Rightarrow$	$(\rho)^*$,	if $\sigma \leq_s \rho$
$(\rho)^*(\sigma)^*$	$\Rightarrow$	$(\sigma)^*$,	if $\rho \leq_s \sigma$
$(\rho)^*(\sigma)^?$	$\Rightarrow$	$(\rho)^*$,	if $\sigma \leq_s \rho$
$(\rho)^?(\sigma)^*$	$\Rightarrow$	$(\sigma)^*$,	if $\rho \leq_s \sigma$
$\alpha(\alpha\rho)^*$	$\Rightarrow$	$(\rho\alpha)^*\alpha$	
$\alpha(\alpha\rho)^?$	$\Rightarrow$	$(\rho\alpha)^?\alpha$	

Figure 2 AF-RE Simplification Rules.

$\perp$	$\leq_a r$	
r	$\leq_a \top$	
ε	$\leq_a \varepsilon$	
$r\alpha$	$\leq_a s\beta$	if $\alpha \leq_t \beta$ and $r \leq_a s$
$r(\rho)^*$	$\leq_a s(\sigma)^*$,	iff ($\rho \leq_s \sigma$ and $r \leq_a s$) or $r(\rho)^* \leq_a s$
$r(\rho)^?$	$\leq_a s$,	if $r\rho \leq_a s$ and $r \leq_a s$
$r(\rho)^*$	$\leq_a s$,	if $r(\rho)^*\rho \leq_a s$ and $r \leq_a s$
r	$\leq_a s(\alpha)^?\sigma$,	if $r \leq_a s\alpha$ or $r \leq_a s$
r	$\leq_a s(\alpha)^*\sigma$,	if $r \leq_a s(\alpha)^*\alpha$ or $r \leq_a s$

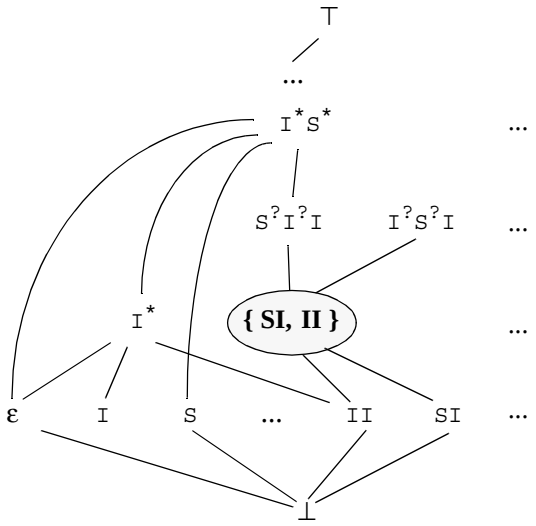


Figure 3 Inclusion Relation for AF-REs.

Figure 4 A Small Part of the $\boldsymbol{L}(\mathbf{S}^{\#*})$ Lattice.

The shaded set of stack states does not have an AF-

elements are AF-REs. It is to be understood that the abstract domain for our analysis is $\boldsymbol{L}(\mathbf{S}^{\#*})$, but that our analysis uses only elements of the domain that have representations as AF-REs. Note that the only difference between $\boldsymbol{L}(\mathbf{S}^{*})$ and $\boldsymbol{L}(\mathbf{S}^{\#*})$ is that the latter possesses additional `N` and `X` codes. A partial diagram of the $\boldsymbol{L}(\mathbf{S}^{\#*})$ lattice indicating some lattice elements that have equivalent AF-RE representations and one that does not is shown in Figure 4. As the figure shows, the `{II,IS}` value cannot be represented by an AF-RE. If this value should arise in an analysis, it must be approximated. Any higher value that can be represented by an AF-RE would be suitable.

We use the notation $\vee$ to represent our conservative approximate implementation of the join, $\cup$, and $\wedge$ to represent the conservative approximate implementation of the meet, $\cap$. In general, the implementations ensure that $r \cup s \subseteq r \vee s$ and $r \cap s \subseteq r \wedge s$ hold.

The meet and join operations are essential to the abstract interpretation: the join is used whenever two control flow paths merge to unify information on the two paths. The meet is used to refine the approximation obtained from backward and forward passes of the analysis algorithm. Meets and joins will be performed with great frequency, so their cost dominates the overall cost of the analysis. Efficiently computable conservative approximations to the true meet and join operations of the $\boldsymbol{L}(\mathbf{S}^{\#*})$ lattice are therefore desirable. The actual meet operation, $\wedge$, used in our analysis is defined in terms of a disjointness relation, $r \# s$, by the four rules shown in Figure 5. The disjointness relation provides a quick test as to whether two sets of stack states are incompatible – that is, whether their intersection is empty. In the interests of execution efficiency, the test is not required to return true in *all* cases when the intersection is empty. That is, the # relationship is defined so that $r \# s$ implies that the intersection of the corresponding sets in $\boldsymbol{L}(\mathbf{S}^{\#*})$, $r \cap s$, is empty. The converse does not necessarily hold. One suitable, simple, definition is provided by the five rules given in Figure 6.

Similarly, the join operation on AF-REs is a conservative approximation of set union. As an example of the possibilities, `BI` $\vee$ `SI` may be joined as `(B)`$^?$`(S)`$^?$`I`, `(BI)`$^?$`(IS)`$^?$ or even `XI`. All are valid but different approximations to the true set union, {`BI`,`SI`}. Depending on the purpose of the analysis, we may prefer one approximation over another. For instance, the third form preserves information on the depth of the stack, the second keeps related sequences of type codes in the same conjunct, while the first and third retain the fact that the top stack element has the type code `I`.

A non-deterministic definition for the join operation is provided by the transformations of Figure 7 plus three more transformations that are the same as rules 6-8 with the operands of $\vee$ interchanged.

Of these transformations, the first two are the only ones that do not introduce any approximation in the result. Hence, when several results are possible, a result that uses the first two transformations the most often is preferred.

$r \wedge s = r$,	if $r \leq_a s$	
$r \wedge s = s$,	if $s \leq_a r$	
$r \wedge s = \perp$,	if $r \# s$	
$r \wedge s = r$ or s,	*otherwise*	

Figure 5 The Meet Operation $\wedge$

$r\alpha \# \varepsilon$		
$\varepsilon \# r\alpha$		
$r\alpha \# s\beta$,	if $r \# s$ or	
	not ($\alpha \not\leq_t \beta$ or $\beta \not\leq_t \alpha$)	
$rc \# s$	if $r \# s$	
$r \# sc$	if $r \# s$	

Figure 6 The Disjointness Function #

1	$r\alpha \vee s\beta \Rightarrow (r \vee s)\alpha$,		if $\beta \leq_t \alpha$
2	$r\alpha \vee s\beta \Rightarrow (r \vee s)\beta$,		if $\alpha \leq_t \beta$
3	r`I` $\vee$ s`R` $\Rightarrow$	$(r \vee s)$`N`	
4	r`R` $\vee$ s`I` $\Rightarrow$	$(r \vee s)$`N`	
5	$r\alpha \vee s\beta \Rightarrow (\alpha \vee s)$`X`		
6	$r\alpha \vee s\beta \Rightarrow (r \vee s)\,(\beta)^?(\alpha)^?$		
7	$r \vee s(\rho)^? \Rightarrow (r \vee s)(\rho)^?$		
8	$r \vee s(\rho)^* \Rightarrow (r \vee s)(\rho)^*$		

Figure 7 The Join Operation $\vee$

3. The abstract interpretation algorithm

The PostScript language was designed for interpretation rather than compilation. The goal is most apparent in the language's elegant but spartan syntax: a program is simply a sequence of operations, be they integers, operators or code blocks. Each operation has a meaning that can be expressed in terms of simple stack operations. An integer constant, for example, should be viewed as an operation that pushes an integer onto the stack.

The abstract interpretation algorithm will take advantage of that simple structure. We perform static analysis by propagating abstract stack states through the program, alternating between forward and backward passes. The AF-RE notation is used to represent these stack states. For each point in the program, forward analysis computes the effect of the next operation in the sequence on the current stack state, the result is then intersected with the previous estimate. This guarantees that each pass can only give a more accurate result, in other words that we progress down the lattice. Thus, for a program consisting of a linear sequence of n operations, we have n formulae of the form:

$$pp'_{i+1} := \mathbf{c}_i(pp_i) \wedge pp_{i+1}$$

which, given initial approximations to two consecutive stack states pp_i and pp_{i+1}, computes a better approximation pp'_{i+1}. Of course, each operation $\mathbf{c}_i$ is defined on the abstract domain.

For non-sequential programs such as programs using the conditional operator `ifelse`, the generalization is obvious: we take the union, $\vee$, of all predecessors of a point.

Backwards analysis reconstructs possible stack states from the results of applying an operator. We need to per-

form backwards passes for two reasons. First, to analyze the type of a function we need to know what it expects on the stack before being invoked; forwards analysis can only give answers on the state of the stack *after* the function has been executed. Second, to improve the result of the analysis by propagating information on how values are used. For instance, consider a program that contains a conditional branch which, if executed, leaves a string on the stack. If the conditional branch is followed by an arithmetic operation, we can conclude that the program is erroneous (or, at least, redundant) since, if that branch is ever taken, a runtime error is guaranteed to occur. Backwards analysis uses abstract inverse functions, $\mathbf{c}_i^{-1}$, to describe the inverse effects of operations on the stack. For each program point, pp_i, we apply a formula

$$pp'_i := \mathbf{c}_i^{-1}(pp_{i+1}, pp_i) \wedge pp_i$$

that yields a more precise estimate of the stack state at pp_i. For backwards analysis in a conventional language like Pascal, the inverse function used in the formula would take the form $\mathbf{c}_i^{-1}(pp_{i+1})$. However, as we explain later, the inverse functions for certain PostScript operations (such as the `ifelse` operator) require additional information to be able to compute a useful result.

The overall analysis is performed by a simple iterative algorithm (similar to one given in [2]). The initial assumption is that at each program point any stack state is possible. (If a complete program is being analyzed, the initial stack state may be set to empty.) The algorithm stops if no stack state changes in the course of one iteration.

```
for i := 0 to n do
      pp_i := X* -- Recall that X* is
                 -- equivalent to ⊤.
repeat
      for i := 0 to n-1 do
           pp_{i+1} := c_i(pp_i) ∧ pp_{i+1}
      for i := n downto 1 do
           pp_i := c_i^-1(pp_{i+1}, pp_i) ∧ pp_i
until a fixpoint is achieved
```

3.1 Abstract operators

So far we have not discussed the meanings of individual PostScript operators. Abstract interpretation requires that all operations in the concrete domain be mapped into corresponding operations in the abstract domain with respect to their effects on the stack. In addition, backwards analysis requires that the inverse of every operation be defined.

All of these operations may be defined in terms of two stack manipulation operations: pushing a value onto the stack, and popping a value off the stack. Our push$^{\#}$ and pop$^{\#}$ operations have unconventional definitions, however, because they operate on sets of stack states. That is, push$^{\#}$ must prefix a type code to every element of the set of states, whereas pop$^{\#}$ must attempt to remove a leading type code from every element of the set – discarding elements that would be inconsistent with the pop operation. To improve the precision of the analysis, pop$^{\#}$ tries to remove a particular type code from the stack. For example pop$^{\#}$(`I`, `S`*) = $\bot$, because no stack state described by `S`* has an `I` at its top. On the other hand, pop$^{\#}$(`I`, `I(S)`$^?$) = ε and pop$^{\#}$(`I`, `I(S)`$^?$`I`$^?$) = `I`$^?$`S`$^?$.

$$
\begin{array}{ll}
\text{pop}^{\#}(\alpha, \top) = \top & \\
\text{pop}^{\#}(\alpha, s\beta) = s, & \text{if } \alpha \not\leq_t \beta \\
\text{pop}^{\#}(\alpha, s(\rho\beta)^?) = \text{pop}^{\#}(\alpha, s) \vee s\rho, & \text{if } \alpha \not\leq_t \beta \\
\text{pop}^{\#}(\alpha, s(\rho\beta)^?) = \text{pop}^{\#}(\alpha, s), & \text{if not } \alpha \not\leq_t \beta \\
\text{pop}^{\#}(\alpha, s(\rho\beta)^*) = \text{pop}^{\#}(\alpha, s) \vee s(\rho\beta)^*\rho, & \text{if } \alpha \not\leq_t \beta \\
\text{pop}^{\#}(\alpha, s(\rho\beta)^*) = \text{pop}^{\#}(\alpha, s), & \text{if not } \alpha \not\leq_t \beta \\
\text{pop}^{\#}(\alpha, s) = \bot, & \textit{otherwise}
\end{array}
$$

$$
\begin{array}{l}
\text{push}^{\#}(\alpha, \bot) = \bot \\
\text{push}^{\#}(\alpha, s) = s\alpha
\end{array}
$$

Figure 8 Abstract Stack Manipulation Operations

Definitions for push$^{\#}$ and pop$^{\#}$ are given in Figure 8. As before, we use ρ to denote a sequence of conjuncts, α as an arbitrary sequence of type codes, and *t* and *u* stand for single type codes. Needless to say, the results of the push$^{\#}$ and pop$^{\#}$ operations usually require simplification.

The analysis of a simple PostScript function, *triple*,

```
/triple  { 3  mult }  def
```

which multiplies its argument by three can now be demonstrated. This function first pushes the integer 3 on the stack, then the multiplication operator pops the two topmost elements and pushes their product back on the stack. The semantic equations for *triple* are shown in Figure 9. In the semantic equations, `3`$^{\#}$, `mult`$^{\#}$, `3`$^{\#-1}$ and `mult`$^{\#-1}$ represent the given abstract functions and their inverses. (The definitions for `mult` and `mult`$^{\#-1}$ assume that both arguments are integers; a more general definition for the polymorphic case is given later.)

The stack states at the three program points marked in the code are named pp_1, pp_2 and pp_3. If we begin with the assumption that the stack state at program point pp_1 is `X`* (representing an unknown stack state), just one iteration with the forwards and backwards equations will deduce the stack states shown in the following table.

	0	**1f**	**1b**
pp_1	⊤	⊤	`X`*`I`
pp_2	⊤	`X`*`I`	`X`*`II`
pp_3	⊤	`X`*`I`	`X`*`I`

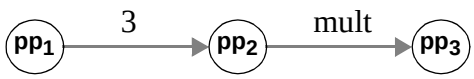
The column headed **0** shows the initial stack states, the column headed **1f** shows the states after the forward pass (computed in the order pp_1, pp_2, pp_3), while the **1b** column shows them after the backward pass (computed in the order pp_3, pp_2, pp_1). It is possible to deduce from the pp_1 and pp_3 states (and the fact that the $\mathtt{X}^*$ component of the stack description is never expanded) that the signature of the *triple* function is $\mathtt{I} \rightarrow \mathtt{I}$.

4. The extended abstract model

4.1 Using values

Some primitive operators in PostScript require more precise analysis than that permitted by AF-RE descriptions of stack states if we wish to derive useful signatures for user-defined functions. PostScript has some polymorphic operators whose effect on the stack depend on the *value*s of one or more of their arguments. To be able to perform useful analyses, it is essential to know these values. The analysis used in our implementation attempts to keep track of two kinds of values – integer values and code block values.

Integer values are essential for determining the effect of the `roll` operator. It is used to perform a circular shift on a segment of the stack. The `roll` operation first pops two integers *j* and *n*. It then rotates the next *n* stack elements by *j* positions. For example applying the `roll` operation to the stack state `<"b",4,"a", 3, 1>` results in the state `<"a","b", 4>` and applying it to `<"b",4,"a", 2, 1>` results in `<"b","a", 4>`. Clearly, we would usually need to know the values of the two control integers in order to be able to construct a description of the stack state after `roll` is executed.



Forwards Equations

$pp_2 = \mathbf{3}^{\#}(\mathbf{pp}_1) \wedge \mathbf{pp}_2$

$pp_3 = \mathbf{mult}^{\#}(\mathbf{pp}_2) \wedge \mathbf{pp}_3$

Backwards Equations

$pp_1 = \mathbf{3}^{\#-1}(\mathbf{pp}_2,\mathbf{pp}_1) \wedge \mathbf{pp}_1$

$pp_2 = \mathbf{mult}^{\#-1}(\mathbf{pp}_3,\mathbf{pp}_2) \wedge \mathbf{pp}_2$

where

$\mathbf{3}^{\#} = \lambda s.\mathrm{push}^{\#}(\mathtt{I},s)$

$\mathbf{3}^{\#-1} = \lambda s.\lambda s'.\mathrm{pop}^{\#}(\mathtt{I},s)$

$\mathbf{mult}^{\#} = \lambda s.\mathrm{push}^{\#}(\mathtt{I},\mathrm{pop}^{\#}(\mathtt{I},\mathrm{pop}^{\#}(\mathtt{I},s)))$

$\mathbf{mult}^{\#-1} = \lambda s.\lambda s'.\mathrm{pop}^{\#}(\mathtt{I},\mathrm{push}^{\#}(\mathtt{I},\mathrm{push}^{\#}(\mathtt{I},s)))$

Figure 9 Semantic Equations for Triple Function

Code block values are needed if we are to be able to handle control constructs as simple as an if-then-else or a while loop. Any sequence of operations may be enclosed by curly braces to form a code block. When the bracketed group is encountered in a program, the interpreter pushes a reference to that code block on to the stack. The operations inside that code block are not executed at that time. A control flow operation, such as `ifelse`, may be used to select a code block and execute it. The `ifelse` operation takes two code blocks and a boolean as its arguments. Depending on the boolean value, it executes one of the two code blocks, discarding the other. Consider, for example, a function that takes three integers as its arguments. If the first integer is zero, it returns the sum of the two other integers. Otherwise, it returns the difference of the two other integers. The signature and the code for the function are

```
add_or_sub:    I I I → I
/add_or_sub {0 eq {add} {sub} ifelse}
def
```

where `eq` is the equality test. As well as illustrating code blocks, the example is also intended to convince you that it is impossible to construct a standard control flow graph for PostScript without first performing reaching analysis on code block values. If it fails to do that, consider the following (ugly) example:

```
{add}{sub} 3 1 roll 0 eq 3 1 roll ifelse
```

Here the code block values accessed by the `ifelse` cannot be discerned without analyzing the effects of the two uses of the `roll` operator. Even more inscrutable examples would be easy to construct.

We handle values by extending $\mathbf{S}^{\#}$, making it a lattice. For instance, each `I` type code that appears in an AF-RE has an associated value attribute. The attribute may hold an integer value, or it may hold a code to indicate that no value is known, or finally it may hold a code to indicate that there is no unique integer value. The attribute values of course form a trivial lattice, with a suitable $\leq_t$ relation. The abstract semantics of an integer constant operator such as `2` are expanded to set the value attribute of the `I` type code in the resulting stack state description. Similarly, the abstract semantics of arithmetic operations such as `add` and `mult` are expanded to compute resultant value attributes when possible. Whenever two `I` type codes with conflicting values must be unified (perhaps as a result of simplifying), the value attribute is tagged to show multiple values are possible. This simple analysis of integer values is normally sufficient to handle an operator like `roll`. (Inspection of typical PostScript code shows that `roll` is almost always used with integer constants for its first two arguments.) When the value attribute is important to the analysis, we will show the value as a subscript. Thus, the

stack state at the program point immediately before the `roll` operation in

```
"b"  4  "a"  3  1  roll
```

would be written as $\mathtt{SI}_4\mathtt{SI}_3\mathtt{I}_1$.

Code blocks are handled in an analogous manner. Each code block that appears in the PostScript program is simply numbered. A reference to a code block on the stack is represented by a type code of `C` with a subscript to identify the particular code block. Thus, if the two code blocks in the *add_or_sub* example are numbered 1 and 2, the stack state immediately before the `ifelse` operation would be described by $\mathtt{BC}_1\mathtt{C}_2$. Forwards analysis of the `ifelse` operation requires that the operations contained in both code blocks be analyzed and the resulting stack states are then joined. If either of the code blocks cannot be identified, the resulting stack state will be represented by $\mathtt{X}^*$. Backwards analysis is a little more complicated because the stack state that follows the `ifelse` does not show which code blocks form the *then* and *else* components of the operation. Our solution is to include the stack state immediately before the `ifelse` in the backwards equation.

4.2 Using function signatures

PostScript functions and operators only affect the top of the stack, so when representing their type signatures only the topmost elements of the stack state that are actually used will be represented, the remainder of the stack will be assumed to be unchanged. As an example, the primitive operator for addition requires nine type signatures to describe it fully:

$\mathtt{add:\ II} \rightarrow \mathtt{I}\quad \mathtt{add:\ IR} \rightarrow \mathtt{R}\quad \mathtt{add:\ RI} \rightarrow \mathtt{R}$
$\mathtt{add:\ RR} \rightarrow \mathtt{R}\quad \mathtt{add:\ IN} \rightarrow \mathtt{N}\quad \mathtt{add:\ RN} \rightarrow \mathtt{R}$
$\mathtt{add:\ NI} \rightarrow \mathtt{N}\quad \mathtt{add:\ NR} \rightarrow \mathtt{R}\quad \mathtt{add:\ NN} \rightarrow \mathtt{N}$

The signatures determine the corresponding abstract functions for addition and its inverse (used in the backwards analysis). The forwards equation may be written as:[3]

$$\mathtt{add}^{\#} = \lambda \mathtt{s}.(\ \mathtt{push}^{\#}(\mathtt{I},\mathtt{pop}^{\#}(\mathtt{I},\mathtt{pop}^{\#}(\mathtt{I},\mathtt{s})))$$
$$\vee\ \mathtt{push}^{\#}(\mathtt{R},\mathtt{pop}^{\#}(\mathtt{R},\mathtt{pop}^{\#}(\mathtt{I},\mathtt{s})))$$
$$\vee\ \mathtt{push}^{\#}(\mathtt{R},\mathtt{pop}^{\#}(\mathtt{I},\mathtt{pop}^{\#}(\mathtt{R},\mathtt{s})))$$
$$\vee\ \mathtt{push}^{\#}(\mathtt{R},\mathtt{pop}^{\#}(\mathtt{R},\mathtt{pop}^{\#}(\mathtt{R},\mathtt{s})))\ )$$

And the inverse operation used in the backwards analysis may be written as:

$$\mathtt{add}^{\#-1} = \lambda \mathtt{s}.(\ \mathtt{push}^{\#}(\mathtt{I},\mathtt{push}^{\#}(\mathtt{I},\mathtt{pop}^{\#}(\mathtt{I},\mathtt{s})))$$
$$\vee\ \mathtt{push}^{\#}(\mathtt{N},\mathtt{push}^{\#}(\mathtt{N},\mathtt{pop}^{\#}(\mathtt{R},\mathtt{s})))\ )$$

The numeric type code `N` must be used in the inverse function because a result type of `R` does not unambiguously imply the types of the two input arguments. However, intersection of the AF-RE generated by $\mathtt{add}^{\#-1}$ with the previous estimate of the stack state would often cause the `N` to be replaced by `I` or `R` again.

3. A variant of $\mathtt{pop}^{\#}$ that extracts two type codes from the stack would (sometimes) yield more precise results. It would also simplify these semantic equations.

5. Abstract interpretation of loops and recursion

5.1 Widening

A simple loop that pushes a value on the stack with each iteration is a potential problem. For example, if the extra value is an integer, the stack state after one iteration might be `I`. A second iteration would lead to the stack state `II`, and joining with the result of the first iteration leads to the set of possible stack states being {`I`, `II`} (to allow for the possibility that the loop exits after either one or two iterations). Similarly, a third iteration leads to {`I`, `II`, `III`}, and so on. The join operation that is used to combine the states into a single AF-RE would generate successive results of `I`, $\mathtt{I}^?\mathtt{I}$, $\mathtt{I}^?\mathtt{I}^?\mathtt{I}$, and so on. It is clear that no matter how many iterations are analyzed, we will not reach a fixpoint, and the analysis will never terminate. It is a well-known problem in abstract interpretation [6].

A standard solution to the termination problem is to work with a lattice with a finite height. You would then be guaranteed that a fixed number of analysis iterations through any loop will reach a fixpoint, even if that fixpoint is $\top$. In our case, we could make the lattice finite by restricting the number of terms in an AF-RE. However, that would introduce an arbitrary degree of approximation in our static analysis. An alternative approach, named *widening*, has been proposed by Cousot [3]. Stransky has used the technique for analyzing Lisp [6]. Widening corresponds to the intuitive approach of guessing a pattern to the sequence of abstract values and thereby allowing a direct jump to the limiting value. In the case of the particular sequence `I`, $\mathtt{I}^?\mathtt{I}$, $\mathtt{I}^?\mathtt{I}^?\mathtt{I}$, a human would probably have no trouble in guessing that $(\mathtt{I}^?)^n\mathtt{I}$ would describe the *n*-th value. The limit of the sequence is obviously $\mathtt{I}^*\mathtt{I}$, and that would be the fixpoint.

We implement the widening approach too, and that forced us to specify the most general patterns that we would look for in successive stack state representations. We check for two different patterns. The first pattern corresponds to a stack that is growing because of values being left on the stack by each iteration or recursive call. It is:

$$r_1\, r_2\, r_3 \ \Rightarrow\ r_1\, \rho\, r_2\, \sigma\, r_3 \ \Rightarrow \ldots \Rightarrow\ r_1\, (\rho)^*\, r_2\, (\sigma)^*\, r_3$$

where each *r* represents a group of conjuncts in our AF-RE notation and Greek letters represent sequences of type codes. The r_1 is intended to correspond to the bottom region of the stack that is unaffected by the loop or recursion. The r_3 is intended to correspond to, for example, loop

control values that are pushed onto the stack at the end of an iteration in order to be consumed by the loop control test. The fundamental pattern is really

$$r \quad \Rightarrow \quad \rho\, r\, \sigma \quad \Rightarrow ... \Rightarrow \quad (\rho)^*\, r\, (\sigma)^*$$

The ρ term corresponds to values that are pushed onto the stack before a recursive call, and the σ term corresponds to values pushed after a recursive call. Of course, either or both of the ρ and σ terms may be empty. The same pattern should describe the effect of a typical loop that generates values on the stack.

The second pattern corresponds to a loop or function that consumes values (our *SumN* function would be an example). The pattern we look for is:

$$r_1\, \rho\, r_2 \quad \Rightarrow \quad r_1\, (\rho)^?\, r_2$$

In this case, we do not need to predict the fixpoint because only a finite number of iterations (determined by the complexity of the initial AF-RE) will be needed before a limiting value is reached. For example, an analysis of a recursive formulation of *SumN* called with four integers initially on the stack would likely progress through the sequence of states `IIII`, `III`$^?$`I`, `II`$^?$`I`$^?$`I`, and `I`$^?$`I`$^?$`I`$^?$`I`. The final expression is the limit value. (The final state is not `I` because, unless we use the value of the count integer, we cannot be sure that all the integers are arguments consumed by *SumN*.)

We implement widening by a function ∇ that takes three arguments: two AF-RE values and an integer *N*, representing the number of iterations that have been performed. The AF-REs represent the stack states on the latest two iterations.

The result of ∇ is an approximation to these AF-RE arguments, with a precision controlled by *N*. If our implementation of $\nabla(S_1, S_2, N)$ finds that the second pattern (a contracting stack) is applicable, the result is simply S_2. If our implementation finds the first pattern (a growing stack) is applicable, the result for N=1 and N=2 is S_2. For N=3, the result is the predicted limit value of the sequence. For N>3, the result is S_1 if S_1 and S_2 are equal, otherwise the result is `X`* (a totally arbitrary stack). The N>3 case handles the situation when the predicted limit value turns out not to be a fixpoint. Since `X`* is the top of the lattice, it is guaranteed to be a fixpoint, but one that unfortunately throws away all information about the loop or function. Finally, if the sequence of stack states does not fit either of our patterns, the result is again `X`*.

If we keep iterating through the code of a loop or a recursive function and apply ∇ after each iteration, we are guaranteed that we will reach a fixpoint in a finite number of steps (at most four steps for the growing stack pattern). We cannot, of course, guarantee that the fixpoint is the *least* fixpoint (which would give the most precise description of the effect of the code).

5.2 An example

Our example is a function *ReadList* that inputs a series of numbers, until a zero is read. The result of the function is the count of the number of values read followed by those values. (Thus, *ReadList* could be used to set up argument values for the *SumN* function used as an earlier example.) The code for *ReadList* uses an auxiliary function *ReadList1*. They are defined as follows:

```
/ReadList  {0  ReadList1} def
/ReadList1  {ReadInt dup 0 eq
      {2 1 roll 1 add ReadList1} {pop}
      ifelse} def
```

(*ReadInt* is assumed to be a function that reads an integer value onto the stack.)

We begin with a brief explanation of the code. On entry to the *ReadList1* function, the stack is assumed to already hold a count followed by that number of integers. For example, the stack state might be <23, 7, 15, 3>. Suppose that *ReadInt* obtains 17 as the next value, leaving it on the stack. Then `dup` will duplicate that value, and 0 will push a zero to obtain the state <23, 7, 15, 3, 17, 17, 0>. Next, `eq` pops and compares the top two values for equality, replacing them by the result *false*. The `ifelse` operator is therefore reached with the stack holding <23, 7, 15, 3, 17, false, c_1, c_2> where c_2 and c_1 denote references to the two code blocks. The `ifelse` operator pops three values and selects code block c_1 for execution. The code block is entered with a stack state of <23, 7, 15, 3, 17>. It uses `roll` to rotate the top two values, yielding <23, 7, 15, 17, 3>. It adds one to the top value yielding <23, 7, 15, 17, 4> and then recursively calls *ReadList1* again. The function is re-entered with the stack again holding a count followed by that number of values. Finally, when a zero is read, the `ifelse` executes the `pop` to remove the zero value from the stack and the recursion unwinds.

If our algorithm is asked to analyze the calling code:

```
" total"  ReadList SumN PrintInt PrintStr
```

it will compute the sequence of stack state values shown in the table below. These are the values at a program point located just after a return from the call to *ReadList1* located inside *ReadList.*.

Iteration, N	Previous Value, S_1	New Value, S_2	∇(N,S1,S2)
1	`X`*	`SI`	`SI`
2	`SI`	`SII`$^?$	`SII`$^?$
3	`SII`$^?$	`SII`$^?$`I`$^?$	`SII`$^?$`I`* ⇒ `SII`* ⇒ `SI`*`I`
4	`SI`*`I`	`SI`*`I`	`SI`*`I`

The effect of the widening function ∇ is shown explicitly in the table. Initially, the stack state at the program point is shown as X^*. When forwards analysis reaches that program point for the first time, a stack state of IS is obtained. These two stack states, plus the iteration number (1), are supplied as arguments to ∇ yielding a result of IS. Two more iterations through the function definition generate the states shown, reaching line 3 where widening extrapolates the effect of the function to $SII^?I^*$. Simplification of this expression yields the form SI^*I and one final iteration verifies that this is indeed a fixpoint.

6. Discussion

The results achieved so far are only a beginning. The problem of analyzing a stack-based language like PostScript has turned out to be an order of magnitude more difficult than analysis of a typical imperative language like Pascal. It is also considerably more difficult than for a polymorphic language like ML or Prolog. We feel, however, that we have achieved our initial goals successfully. We are able to analyze a rich subset of PostScript, describing the stack states using a notation, AF-RE, that is readily intelligible to people. The analysis techniques achieve precise results on simple code and approximate, but conservative, results on more involved code. The results of the analysis can assist program verification by identifying erroneous code and could form the basis for a PostScript compiler or optimizer by finding function arguments that have fixed types.

In the future, we intend to explore alternative analysis techniques to see whether they can achieve more precise results without becoming computationally infeasible. One possibility is to drop the use of regular expression notation for stack states and use finite state automata (FSAs) instead. Since the regular languages are closed under union and intersection, we could use FSAs as the lattice of values in the abstract domain, (We wonder, however, whether we would be able to invent a suitable widening operator for use with FSA values.)

We are, of course, still a long way from handling the full PostScript language. Variables are, perhaps, the biggest omission. In full PostScript, a value may be bound to a name in the same way as a function definition is bound to the function's name. For example,

```
/counter 0 def
```

(the / prefix indicates a name as opposed to a value). Subsequently, the identifier *counter* (without the prefix) may be used to refer to the associated value, 0. Since the `def` operator may also be used to re-define the value of *counter*, *counter* acts like a variable. Our current subset of PostScript does not include variables because of the dynamic scope structure of PostScript. The PostScript interpreter maintains a separate stack of dictionaries, and each dictionary holds bindings of names to values. A name lookup involves a search of these dictionaries in stack order. A PostScript program may create new dictionary objects dynamically and may manipulate the dictionary stack. We therefore cannot handle variables in a manner that would be true to the spirit of PostScript until we have incorporated dictionary objects into our analysis. (The NeWS dialect encourages extensive use of dictionaries to emulate objects in an object-oriented style of programming.)

Static analysis of the *full* PostScript language is an impossible goal. In principle, it is possible to re-bind any predefined operator to make it execute different code. When combined with the possibility that a program may read or create a text string and dynamically convert that string into a code block with the `cvx` operator (convert to executable), a safe static analysis could not continue after a use of `cvx`. After that point, you could not be sure that any operator still performed the same operation. A permanent restriction to a subset of PostScript where operators may only be re-bound to code with identical function signatures would appear to be necessary (and quite sensible too).

PostScript is truly an interesting language.

References

[1] S. Abramsky and C. Hankin, An Introduction to Abstract Interpretation in *Abstract Interpretation of Declarative Languages*, Abramsky and Hankin, Ellis Horwood, 1987.

[2] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley (1986).

[3] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Program by Construction or Approximation of Fixpoints," *4th POPL* , Los Angeles, CA (January 1977).

[4] P. Cousot, "Semantic Foundations of Program Analysis," in *Program Flow Analysis: Theory and Applications*, S. S. Munchnick, and N. D. Jones (editors), Prentice Hall, 1981.

[5] L. J. Hendren, "Parallelizing Programs with Recursive Data Structures," Ph.D Thesis, TR-90-1114, Cornell University, April 1990.

[6] J. Stransky, "Analyse sémantique de structures de données dynamiques avec applications au cas particulier de langages LISPiens," PhD Thesis, Université de Paris-Sud, Centre d'Orsay, June 1988.