2010

Path-Sensitive Analysis Using Edge Strings

Armand Navabi
Purdue University, Anavabi@cs.purdue.edu

Nicholas Kidd
Purdue University, nkidd@cs.purdue.edu

Suresh Jagannathan
Purdue University, suresh@cs.purdue.edu

Report Number:
10-006

Path-Sensitive Analysis Using Edge Strings

Armand Navabi, Nicholas Kidd, and Suresh Jagannathan

Purdue University
{anavabi,nkidd,suresh}@cs.purdue.edu

Abstract. Path sensitivity improves the quality of static analysis by avoiding approximative merging of dataflow facts collected along distinct program paths. Because full path sensitivity has prohibitive cost, it is worthwhile to consider hybrid approaches that provide path sensitivity on selected subsets of paths. In this paper, we consider such a technique based on an *edge string*, a compact abstraction of a set of static program paths. The edge string $es = [e_1, e_2, \ldots, e_k]$, where each e_i is an edge label found in a program's control-flow graph, is used to disambiguate dataflow facts that manifest *only* on paths in which es occurs as a subsequence. The length of es dictates the tradeoff between precision and analysis cost. Loosely speaking, edge strings are a path-sensitive analog to the notion of call-strings exploited by context-sensitive analyses .

We present a formalization of edge strings and discuss optimizations that incorporate additional relevance measures, based on the structure of the control-flow graph, to avoid exploring edge-string paths if no added precision accrues.

We also provide a detailed implementation study in the context of the functional SSA intermediate representation used by MLton, a whole-program optimizing compiler for Standard ML. Our results indicate that small edge strings provide the necessary precision to identify infeasible paths for functional programs that leverage complex control and dataflow.

1 Introduction

Classical dataflow analyses are structured to compute a fixpoint of a system of equations whose solution defines a conservative approximation to a program property of interest. Oftentimes, analyses sacrifice precision by (a) allowing facts collected along different unrelated paths to be merged at join points, and (b) ignoring how branches are correlated [3], leading to facts collected along infeasible paths to unfavorably influence the analysis of feasible regions.

A fully path-sensitive analysis avoids merging facts discovered along distinct control paths. Unfortunately, naive path enumeration is impractical for all but the simplest of programs. Approximations that limit the number of paths that are explored are therefore necessary. While it is especially useful to avoid exploring infeasible paths, the challenge, of course, is to identify such infeasible paths without having to first enumerate an intractably large set of potential paths, most of which may be feasible.

In this paper, we introduce a novel technique inspired by call-string approximations [15, 16] used to provide context-sensitivity by approximating the structure of a program's call stack, to build efficient *path-sensitive* approximations. An *edge string* of length k disambiguates dataflow facts based on a sequence of k edges visited by the analysis. For example, an edge string of length one parameterizes a dataflow analysis based on a specific edge encountered along any path. In this setting, suppose an analysis encounters a branch B_1 with predicate p, where the outgoing true and false edges are labeled by e_t and e_f, respectively. Dataflow analysis along the true and false paths from B_1 are parameterized with an edge string containing the corresponding true (or false) edge label. At a subsequent join, the abstract values propagated along the two outgoing branch edges e_t and e_f are not merged since their corresponding edge strings are distinct.

By associating an edge string with the abstract value(s) associated with different sequences of edges, we have a simple and useful technique to identify infeasibility. Continuing with our example, consider a subsequent post-dominating branch B_2 with predicate $\neg p$. Since a path-sensitive analysis that uses an edge string of length one does not merge results computed along the different branches of B_1 (because they are associated with distinct edge strings e_t and e_f, respectively), it can avoid propagating facts collected from e_t (which assumes p) along B_2's true edge, and *vise versa*. Let e'_t and e'_f be the edges that correspond to B_2's true and false branches. Querying the dataflow facts associated with the edge string $[e_t, e'_t]$ and $[e_f, e'_f]$ reveals that any path containing these sequence of edges is infeasible since there are no consistent set of facts that hold on any path containing these different edge pairings. (The former asserts p from e_t and $\neg p$ from e'_t, and similarly for the latter.)

Thus, like context-sensitive analyses, the value of k dictates the degree of precision achieved. Program analyses executed with edge-string length $k = 0$ are path-insensitive. Edge strings of length n ensure full path sensitivity if the control-flow graph being examined has no more than n edges that can be (statically) visited from entry to exit. We note that while the analogy to call-strings is useful, the correspondence is not precise. A call-string of length k represents an approximation of a program stack fixed to having no greater than k activation frames. Thus, successive elements in the call-string represent successive function calls. In contrast, an edge string represents a *path constraint*: the edge string $[e_1, e_2, \ldots, e_n]$ represents a set of paths all of whom include the edges found in the string *in that order*, but not necessarily as a contiguous sequence. Thus, an edge string defines an abstraction of all paths for which the edge string is a subsequence. An abstract value associated with edge string *es* represents a property derived by examining only paths that include *es*.

Edge strings with *gaps* (i.e., containing sequences of edges that are not contiguous in the control-flow graph) can be used to capture infeasibility relationships that manifest across a potentially large set of intervening nodes. In contrast, edge strings whose elements are contiguous can be used to identify infeasibility that arises within a localized region of the flow graph.

Thus, we can think of an edge string as a path filtering abstraction over a flow graph. Questions of infeasibility are addressed on this filtered graph using standard path-insensitive techniques. Our intuition, validated by experimental results, is that the benefits of full path sensitivity with respect to infeasible path detection can be achieved using small values of k.

This paper makes three contributions. First, we present a general path-sensitive dataflow-analysis framework using edge strings. Second, we consider refinements and optimizations on edge string construction to improve efficiency and scalability. These optimizations impose relevance criteria that compute edge strings only if the dataflow information associated with the paths they abstract uniquely provides additional precision. These relevance measures allow us to construct edge strings composed of contiguous sequences of relevant edges; surprisingly, the elements in such strings need not correspond to contiguous edges in the control-flow graph. Third, we evaluate the effectiveness of our approach in identifying infeasible paths over control-flow graphs expressed in the SSA intermediate representation used by MLton [13], a high-performance whole-program optimizing compiler for Standard ML [12]. Although our source language is ML, we note that the realization of the analysis on an SSA representation makes it easily applicable to other more imperative languages. Our results indicate that small edge-string lengths are sufficient to identify a large number of interesting infeasible paths.

The remainder of the paper is structured as follows. §2 presents additional motivation. §3 formalizes our approach. We also discuss various techniques that allow us to avoid including edges in the construction of new edge strings if no additional precision is achieved, leading to a more efficient analysis. Experimental results are given in §4. We discuss related work in §5, and present conclusions and future work in §6.

2 Motivating Example

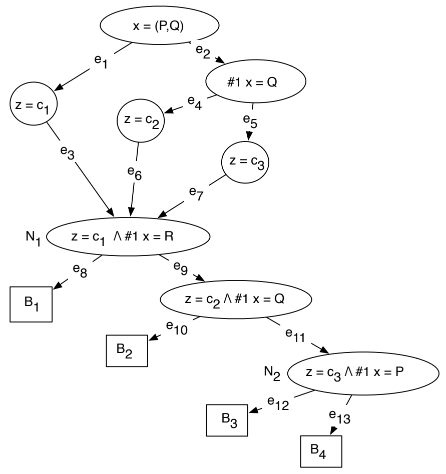
While path-sensitive analyses have been mostly studied in the context of imperative programs [9, 3, 6, 2, 8], they have significant utility for optimizing functional programs as well. For example, functional programs often use pattern-matching to express complex data and control flow. Given a subject term t and set of patterns P, a pattern-matching compiler [1, 10] yields a decision procedure that compares the equivalence of the tree representations between t and each of the patterns in P. The implementation of this procedure is typically in terms of a complex series of bindings and conditional checks that often include infeasible paths that arise because of inconsistent pattern-matching assumptions. Consider the following ML program fragment:

```
datatype t = P | Q | R
let val z = (case x of (P, Q) => c1 | (Q, _) => c2 | _ => c3)
in (case (z,#1 x) of
      (c1, R) => B1 | (c2, Q) => B2 | (c3, P) => B3 |  _ =>  B4)
end
```

The first case deconstructs expression `x` whose type is `(t * t)` where `t` is a datatype consisting of nullary constructors `P`, `Q`, and `R`. The second case expression uses the result of the first to impose additional flow constraints. A control-flow graph for the program fragment is shown in Fig. 1. Observe that there exists both infeasible paths and (dynamically) unreachable code:[1]

– Expression B_1 is unreachable—i.e., all paths to B_1 are infeasible—because `z` is bound to c_1 precisely when the first component of `x` is `P`, which conflicts with the second constraint in the conjunct: `z =` $c_1 \land$ `#1 x = R`.

[1] The program fragment is contrived to be small and contain both infeasible paths and unreachable code. However, redundant tests like the second case statement are not uncommon for a pattern-matching compiler that performs aggressive inlining of (helper) functions (e.g. MLton).



$$
\begin{aligned}
N_1, [e_1, e_3], [e_1], [e_3] &\mapsto \{\mathtt{z} = c_1, \mathtt{x} = (\mathtt{P}, \mathtt{Q})\}\\
N_1, [e_2] &\mapsto \{\mathtt{z} = \top, \mathtt{x} = (\top, \top)\}\\
N_1, [e_2, e_6], [e_4], [e_6] &\mapsto \{\mathtt{z} = c_2, \mathtt{x} = (\mathtt{Q}, \top)\}\\
N_1, [e_2, e_7], [e_5], [e_7] &\mapsto \{\mathtt{z} = c_3, \mathtt{x} = (\top, \top)\}\\[1ex]
N_2, [e_1, e_9] &\mapsto \{\mathtt{z} = c_1, \mathtt{x} = (\mathtt{P}, \mathtt{G})\}\\
N_2, [e_6, e_{11}] &\mapsto \emptyset\\
N_2, [e_7, e_9] &\mapsto \{\mathtt{z} = c_3, \mathtt{x} = (\top, \top)\}\\[1ex]
B_1, [e_2] &\mapsto \{\mathtt{z} = \top, \mathtt{x} = (\top, \top)\}\\
B_1, [e_3], [e_4], [e_6, e_8] &\mapsto \emptyset\\
B_1, [e_3, e_8] &\mapsto \emptyset\\[1ex]
B_2, [e_2, e_4], [e_4, e_{10}] &\mapsto \{\mathtt{z} = c_2, \mathtt{x} = (\mathtt{Q}, \top)\}\\
B_2, [e_1, e_{10}] &\mapsto \emptyset\\[1ex]
B_3, [e_6, e_{12}] &\mapsto \emptyset\\
B_3, [e_7, e_{12}] &\mapsto \{\mathtt{z} = c_3, \mathtt{x} = (\mathtt{P}, \top)\}\\[1ex]
B_4, [e_1, e_{13}] &\mapsto \{\mathtt{z} = c_1, \mathtt{x} = (\mathtt{P}, \mathtt{Q})\}\\
B_4, [e_6, e_{12}] &\mapsto \emptyset
\end{aligned}
$$

Fig. 1. On the left is the control-flow graph (CFG) for the ML program fragment in §2. On the right is a listing of CFG node, a list of potential edge strings that reach a node (i.e., the list is non-exhaustive), and dataflow facts associated with the edge strings in the list at the node.

- Expression B_2 can be reached via several paths. One, $e_2 \to e_4 \to e_6 \to e_9 \to e_{10}$ is feasible, and occurs when the second component of `x` is the nullary constructor `Q`. Path $e_1 \to e_3 \to e_9 \to e_{10}$ is infeasible, however. Discovering this infeasibility requires that dataflow facts propagated along edge e_3 not be merged with facts propagated along edges e_6 or e_7 at the entry to node N_1.
- Expressions B_3 and B_4 can be reached via several paths, both feasible and infeasible. Path $e_5 \to e_7 \to e_9 \to e_{11} \to e_{12}$ is the only feasible path that leads to B_3. Path $e_5 \to e_7 \to e_9 \to e_{11} \to e_{13}$ is a feasible path that reaches B_4. Paths through e_3 or e_6 that reach B_3 are infeasible because of the constraint `z`=c_3, which only arises on paths that include edge e_7.

By avoiding a merge of flow information associated with paths that do not include all of the edges in an edge string, the dataflow facts collected using edge strings allow infeasible paths to be discovered. Fig. 1 also illustrates flow information (represented as a set of abstract bindings for `z` and `x`) found at the entry to various nodes associated with selected edge strings. For this example, we assume a standard join-over-all-paths analysis, suitably parameterized by an edge string, in which $c_i \sqcup c_j = \top$ if $i \neq j$, `P` $\sqcup$ `Q` = $\top$, and so on. The abstract-value domain is a simple flat lattice that consists of constants c_1, c_2, and c_3, and tuples built from constructors `P`, `Q`, and `R`, i.e., abstract values are merely equality constraints on the variables `x` and `z`. (We do not consider inequality constraints in the example.) Not all edge strings have sensible interpretations based on the facts associated with each component edge in the string. Indeed, paths associated with such edge strings are infeasible, and have an empty fact set ($\emptyset$).

For example, parameterizing the analysis with edge strings e_1 or e_3 allows the conclusion that `z` is c_1 and `x` is `(P,Q)`. Similarly, the dataflow facts associated with edge string e_9 binds `z` to $\top$ and `x` to $(\top, \top)$. This is because paths that emanate from edges e_1, e_4 and e_5 can also include e_9. Each of these paths define potentially different bindings for `z` and `x`. On the other hand, dataflow information associated with paths that include the sequence $[e_1, e_9]$ is more refined since paths are constrained by $[e_1]$ to map `z` to c_1 and `x` to `(P,Q)`. Note also that the edge string $[e_2]$ defines a join over dataflow facts collected along paths that include e_4 or e_5. In other words, computing flow information using edge string $[e_2]$ provides no additional precision beyond what is provided by computing flow information for edge string $[e_4]$ and $[e_5]$ separately. Some of the infeasible paths found in the graph are listed in the table. For example, paths that include edges e_6 and e_8 (leading to B_1), or e_1 and e_{10} (leading to B_2) are infeasible.

Smaller length edge strings subsume information from a larger number of paths. In this example, we see that even with an edge string of length two, a number of infeasible paths can be detected. By using edge strings as a simple parameterizable mechanism to identify infeasible paths, we can subsequently refine a control-flow graph using techniques like syntactic language refinement [2] to eliminate the infeasible path from consideration by subsequent optimizations. Our experimental results reveal that small edge-string lengths are sufficient, in general, to identify many infeasible paths. Thus, edge strings are a *compact* encoding of a set of paths that can be leveraged by *both* static and dynamic program analyses.

3 Edge Strings

Before presenting a formal definition of edge strings, we first give background definitions necessary for computing a join-over-all-paths dataflow analysis of a control-flow graph.

3.1 Background

A control-flow graph (CFG) $G = (N, n_0, n_f, E)$ is a directed graph where N is a finite set of nodes, $n_0 \in N$ is a distinguished start node, $n_f \in N$ is a distinguished exit node, and $E \subseteq N \times N$ is an edge relation on nodes. We use n, n', n_1,... to range over nodes in N. For two nodes n and n' such that $(n, n') \in E$, denoted by $n \to n'$, we say there is an edge from n to n' whose head is n and whose tail is n'. We use e, e', $e_1, \ldots$ to range over edges in E. For an edge $e \in E$, we use $\mathsf{hd}(e)$ and $\mathsf{tl}(e)$ to denote the head and tail of e, respectively. Finally, we assume that n_0 has no incoming edge and that n_f has no outgoing edge, i.e., $\nexists\, e \in E \ :\ \mathsf{tl}(e) = n_0 \vee \mathsf{hd}(e) = n_f$.

The abstract meaning of a CFG G is given by an abstract interpretation [5] $\mathcal{S} = (\mathcal{L}, F_N, F_E)$, whose components are defined as follows:

- $\mathcal{L} = \langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ is a lattice where D is a partially ordered set under relation $\sqsubseteq$; $\bot \in D$ and $\top \in D$ are a least and greatest element, respectively; $\sqcup$ is a least upper bound operation; and $\sqcap$ a greatest lower bound operation.
- $F_N : N \times D \to D$ is a monotonic function that gives the abstract interpretation for executing a node $n \in N$ from abstract state $d \in D$. For the CFGs of interest, F_N models the allocation and assignment of the variables of G.

- $F_E : E \times D \rightarrow D$ is a monotonic function that gives the abstract interpretation for an execution (path) of G to follow an edge $e \in E$ from abstract state $d \in D$. F_E models the constraints that might be imposed on an abstract state d in order for an execution to follow a branching edge.

A valid path p in G is a sequence of edges $e_1, \ldots, e_m$ such that for all $0 < i < m$, $\mathsf{tl}(e_i) = \mathsf{hd}(e_{i+1})$. For two nodes $n, n' \in N$, the set of all valid paths from n to n' in G is denoted by $\mathsf{paths}(n, n')$. Because of cycles in G, $|\mathsf{paths}(n, n')|$ can be infinite. The meaning of a valid path $p = e_1, \ldots, e_m$ when executed from initial abstract state d_I, denoted by $\mathsf{val}(p, d_I)$, is defined over $\mathcal{S}$:

$$\mathsf{val}(p, d_I) \triangleq F_N(n_1, d_I) \gg F_E(e_1) \gg F_N(n_2) \gg \ldots \gg F_N(n_m) \gg F_E(e_m) \gg F_N(\mathsf{tl}(e_m)),$$

where for $1 \leq i \leq m$, $n_i = \mathsf{hd}(e_i)$, and $\gg$ is a left-associative binary operator that threads the output of its left-hand-side expression as input to its right-hand-side expression.

For a given CFG G and abstract interpretation $\mathcal{S}$, a path-insensitive dataflow analysis computes for each node $n \in N$, the *join-over-all-paths value* from the start node n_0 to n with initial abstract state d_I, denoted by $\mathsf{JOP}(n_0, n)$ and defined as:

$$\mathsf{JOP}(n_0, n) \triangleq \bigsqcup \{ \ \mathsf{val}(p, d_I) \mid p \in \mathsf{paths}(n_0, n) \ \}. \tag{1}$$

From the definition of $\mathsf{JOP}(n_0, n)$, one can see that the computed value does not retain any path information. Such an analysis is sometimes referred to as a *first-order collecting semantics* [14] because the JOP-value at a node $n \in N$ is the *collection* of the set of (abstract) states that arise at n via any path $p \in \mathsf{paths}(n_0, n)$.

3.2 Edge String

An *edge string* is an *abstraction* of a program path. The goal of edge strings is to be a simple yet effective mechanism to extend Eqn. (1) with a form of k-limited path sensitivity. The idea is as follows: instead of extending the (likely already complex) abstract domain to include path information, we make a minor extension to the definition of JOP.

Formally, an edge string *es* is an ordered sequence of edges in E, $[e_1, \ldots, e_k]$, that serves as an abstraction of a set of program paths. For a given value of k, the set of all possible edge strings is then bounded by E^k. Because there are (potentially) infinitely-many paths, there are (potentially) infinitely-many relations on paths and edge strings that one can define. In this paper, we consider two such relations, α^k_{last} and α^k_{gap}.

The relation $\alpha^k_{\text{last}} \subseteq E^* \times E^k$ is the edge-string equivalent of call strings—it records the k-most recent edges of a valid path. Specifically, for valid path $p = e_1, \ldots, e_m$,

$$\begin{aligned} \alpha^0_{\text{last}}(e_1, \ldots, e_m) &= \qquad\qquad \{[]\} \\ \alpha^k_{\text{last}}(e_1, \ldots, e_m) &= \alpha^{k-1}_{\text{last}}(e_1, \ldots, e_{m-1}) \bowtie \{[e_m]\}, \end{aligned}$$

where the $\bowtie$ function returns the cross product of two sets (e.g., $\{[e], [e']\} \bowtie \{[e_1, e_2]\} = \{[e, e_1, e_2], [e', e_1, e_2]\}$). For the case where $m < k$, $\alpha^k_{\text{last}}(p) = [p]$. Note that if $m < k$, then α^k_{last} is not a path abstraction but the identify function. Thus, α^k_{last} is fully path sensitive for paths of length less than k.

The relation $\alpha^k_{\text{gap}} \subseteq E^* \times E^k$ associates with every valid path $p = e_1, \ldots, e_m$, a set of edge strings of length k such that $(p, \mathit{es}) \in \alpha^k_{\text{gap}}$ iff the constituent edges $[e_1, \ldots, e_k]$ of *es* occur in p in that order. We say that the relation is "gappy" because, for member edges e_i and e_{i+1} of *es*, $1 \leq i < k$, α^k_{gap} allows for an arbitrary-length sequence of edges to occur between them in p. Thus,

$$\begin{aligned} \alpha^0_{\text{gap}}(e_1, \ldots, e_m) &= \qquad \{[]\} \\ \alpha^k_{\text{gap}}(e_1, \ldots, e_m) &= \alpha^{k-1}_{\text{gap}}(e_1, \ldots, e_{m-1}) \bowtie \{[e_m]\} \cup \underline{\alpha^k_{\text{gap}}(e_1, \ldots, e_{m-1})}. \end{aligned}$$

The underlined portion above highlights the extension to α^k_{last} that results in α^k_{gap}. It is precisely this extension that allows for there to be (arbitrary-length) gaps between two consecutive edges of an edge string *es* that models a valid path p. Moreover, the extension is also the reason that α^k_{gap} is a many-to-many relation (whereas α^k_{last} is a many-to-one relation). By definition, α^k_{gap} is a partial function because it is undefined when p has length less than k. We account for this case by defining $\alpha^k_{\text{gap}}(p)$ to merely return $[p]$, and thus, like α^k_{last}, be fully path sensitive for such paths.

For CFG G, the edge-string abstraction function $\alpha^k \in \{\alpha^k_{\text{last}}, \alpha^k_{\text{gap}}\}$ defines an abstract interpretation of the paths in G defined by $\mathcal{S}^k = (\mathcal{L}^k, F^k_N, F^k_E)$, where $\mathcal{L}^k = \langle D^k = \mathcal{P}(\cup^k_{i=0} E^i), \subseteq, \emptyset, \cup^k_{i=0} E^i, \cap, \cup \rangle$ is a complete lattice with elements being sets of edge strings; $F_N = \lambda n, d.d$ is the identity function; and $F_E = \lambda e, d.\alpha_k(d \bowtie \{[e]\})$ is the function that extends an existing set of edge strings with the edge e and then abstracts this new set via α^k. Thus edge strings provide an infinite family of abstract interpretations parameterized by k. Moreover, two abstract interpretations $\mathcal{S}^{k+1}$ and $\mathcal{S}^k$ are related via a *Galois connection* $f : D^{k+1} \to D^k$ and $g : D^k \to D^{k+1}$, where $f = \alpha^k$ removes one edge from an edge string, thereby reducing precision; $g_{\text{last}} = \lambda d^k.\alpha^{k+1}_{\text{last}}(\{[e] \mid e \in E\} \bowtie d^k)$ merely prepends an edge e to each edge string in d^k to generate a set of length $k+1$ edge strings, thereby increasing precision; and $g_{\text{gap}} = \lambda d^k.\alpha^{k+1}_{\text{gap}}(\{[e] \mid e \in E\} \sqcup\!\sqcup\, d^k)$ inserts an edge e at every position in an edge string in d^k via $\sqcup\!\sqcup$, which denotes the shuffle operation on strings—$\epsilon \sqcup\!\sqcup\, u = u = u \sqcup\!\sqcup\, \epsilon$ and $au \sqcup\!\sqcup\, bv = a \bowtie (u \sqcup\!\sqcup\, bv) \cup b \bowtie (au \sqcup\!\sqcup\, v)$—extended to operate on sets of (edge) strings in the natural way.

Edge-String Analysis. For CFG G, abstract interpretation $\mathcal{S}$, bound k, and edge-string abstraction function α^k, where the edge-string abstraction functions of interest are either α^k_{gap} or α^k_{last}, the goal of an *edge-string analysis* (ESA) is to compute for each $\mathit{es} \in E^k$ the edge-string JOP value from n_0 and to every node $n \in N$ defined as follows:

$$\mathsf{JOP}_{\mathsf{ESA}}(n_0, n, \mathit{es}) \triangleq \bigsqcup \{\, \mathsf{val}(p, d_I) \mid p \in \mathsf{paths}(n_0, n) \wedge \mathit{es} \in \alpha^k(p) \,\}. \qquad (2)$$

Edge strings provide a limited form of path-sensitivity because they delay joins. For example, consider two valid paths p and p' in $\mathsf{paths}(n_0, n)$ such that $(p, \mathit{es}) \in \alpha^k$ and $(p', \mathit{es}') \in \alpha^k$, and $\mathit{es} \neq \mathit{es}'$. Because *es* and *es'* are not equal, the abstract values for p and p', $\mathsf{val}(p, d_I)$ and $\mathsf{val}(p', d_I)$, respectively, will not be joined together as

```
input  : CFG $G$, $\mathcal{S}$, and bound $k$
output: JOP$_{\mathsf{ESA}}$ result map $\mathcal{M}_N : N \mapsto E^k \mapsto D$
let Worklist $= \{n \mid (n_0, n) \in E\}$; let $\mathcal{M}_N(n_0)([]) = d_I$;
while Worklist $\neq \emptyset$ do
    (Worklist, $n$) $=$ Choose(Worklist);
    foreach $e' = (n', n) \in E$ do // Each predecessor $n'$ of $n$
        foreach $es \in \mathcal{M}_N(n')$ do // Each edge string that reaches $n'$
            $d = \mathcal{M}_N(n')(es) \gg F_E(e') \gg F_N(n)$;
            foreach $es' \in \alpha^k(es \cdot [e'])$ do // Each extended edge string
                $\mathcal{M}_N(n)(es') = \mathcal{M}_N(n)(es') \sqcup d$;
    if Changed($\mathcal{M}_N(n)$) then // Add $n'$s successors upon change
        Worklist $=$ Worklist $\cup \{n'' \mid (n, n'') \in E\}$;
```

Algorithm 1: Worklist algorithm to compute $\mathsf{JOP}_{\mathsf{ESA}}$ for a given bound k.

in $\mathsf{JOP}(n_0, n)$, but will only be joined with the abstract values for paths that have a common edge string in the edge-string relation α^k. For α^k_{last}, only those paths that take the exact same *last-k* edges will have their values joined. For α^k_{gap}, paths that contain common k-length "gappy" sequences will have their abstract values joined.

Alg. 1 presents a worklist algorithm to compute Eqn. (2).[2] A few points are worth mentioning. First, $\mathcal{M}_N$ is a map from a node $n \in N$, edge string $es \in E^k$, to a lattice element $d \in D$. Second, the function Choose chooses an element n from the worklist Worklist and returns n and a new worklist without n. Third, the use of '$\cdot$' in $es \cdot [e]$ denotes sequence concatenation. Because the edge-string relations α^k_{gap} and α^k_{last} are defined inductively, they are used to naturally extend the standard worklist algorithm with a third loop over generated edge strings.

3.3 Relevant Edge

Revisiting Fig. 1, one can see that not all edges in the CFG add (flow) constraints. For example, the edges e_3, e_6, and e_7 are the result of a direct control transfer control to the second `case` statement. Because edge strings provide only bounded path sensitivity, it would be unproductive to use a *non-branching* edge, i.e., an edge that imposes no flow constraints, as one of the precious few k edges in an edge string.

The function $\mathsf{relevant} : E \to \{\mathsf{tt}, \mathsf{ff}\}$ specifies whether an edge e is relevant (tt) or not (ff). Given a definition of relevant, we specialize the concatenation function '$\cdot$' in Alg. 1 for edge string $es = [e_1, \ldots, e_k]$ and edge e as follows:

$$es \cdot [e] = \begin{cases} [e_1, \ldots, e_k, e] & \mathsf{relevant}(e) = \mathsf{tt} \\ es & \mathsf{relevant}(e) = \mathsf{ff} \end{cases}$$

One natural definition of relevant is that for every non-branching edge e, $\mathsf{relevant}(e) = \mathsf{ff}$, and $\mathsf{relevant}(e) = \mathsf{tt}$ otherwise (i.e., e is a branching edge). For the CFG in Fig. 1,

[2] Alg. 1 is precise when F_N and F_E are distributive, and a safe approximation otherwise.

this definition of relevant would have the desired effect of eliminating e_3, e_6, and e_7 from being included in an edge string.

Although the straightforward definition of relevant is useful, it can be further improved. Specifically, observe the flow constraints $[e_2] \mapsto \{\mathtt{z} = \top, \mathtt{x} = (\top, \top)\}$ that arise at the first case statement when using edge-string relation α^1_{gap}. This constraint is quite imprecise as every binding maps its variable to $\top$. Due to the imprecision, the edge-string analysis will soundly, yet unhelpfully, determine that expression B_1 is reachable since there is a valid path to B_1 that includes edge e_2.[3]

We can eliminate this imprecision by refining the relevant function. Intuitively, for a node $n \in N$ and edge $e \in E$, if for every $p \in \mathsf{paths}(n_0, n)$, edge e is a member of p, then the flow constraint $F_E(e)$ is contributed by every path p because e is always traversed. Returning to the consideration of edge e_2, every path from the root to its children must include e_2; thus, the flow constraint that e_2 contributes, namely `x <> (P,G)`, will flow along those paths. This insight leads to the *must-traverse-edge* analysis, which computes at each node $n \in N$ the set of edges that all forward paths from n_0 to n must traverse. This is a simple and efficient path-insensitive analysis that is solved by giving as inputs to Alg. 1: (i) the CFG G; (ii) $\mathcal{S}_{\mathsf{MTE}} = (\mathcal{L} = \langle 2^E, \supseteq, E, \emptyset, \cap, \cup \rangle, F_N, F_E)$, where lattice elements are edge sets ordered by superset; F_N is the identity function $\lambda n.\lambda d.d$; and F_E is the function $\lambda e.\lambda d.d \cup \{e\}$ that always adds the edge e to the MTE-abstract state; and (iii) path-insensitive bound $k = 0$.

The result of Alg. 1 instantiated with $\mathcal{S}_{\mathsf{MTE}}$ is a map $\mathsf{MTE} : N \rightarrow 2^E$ such that for a node n, $\mathsf{MTE}(n)$ is the set of edges that all paths from n_0 must traverse to reach n. We refine relevant then to be: $\mathsf{relevant}(e) = e \notin \mathsf{MTE}(\mathsf{tl}(e))$, i.e., e is relevant if it is *not* a must-traverse edge. For the CFG in Fig. 1, an MTE analysis would compute that the relevant edges are $\{e_3, e_6, e_7\}$, and all other edges are implied by the structure of the CFG. An interesting side effect is that MTE analysis allows for α^k_{last} to allow for a certain degree of "gappyness" (since relevant edges need not be contiguous), which greatly increases the precision of α^k_{last}, and also reduces the cost for α^k_{gap}. For example, at expression B_4 of the CFG in Fig. 1, only the edge strings $[e_3]$, $[e_6]$, and $[e_7]$ will arise.

As a final remark, we observe that for independent-attribute domains such as the one (briefly) discussed in §2, ESA extends such domains with limited relational information of the form "variables x and y have abstract values d_x and d_y flow to node n along a path that is abstracted to edge string *es*", i.e., the edge string *es* disambiguates dataflow information and this disambiguation serves as the relation between abstract bindings that may arise during program execution.

4 Experiments

We implemented ESA in the MLton compiler [13], a whole-program optimizing compiler for SML. An SSA program in MLton consists of datatype declarations, a sequence of global statements and a collection of functions. Each function consists of a flat vector of blocks which take arguments and contain statements. Each block contains a transfer

[3] In this case, an edge-string analysis using α^1_{last} will correctly determine that expression B_1 is unreachable.

Table 1. Benchmark Characteristics: Along with the number of blocks, edges, branches and back-edges, we report the percentage of relevant edges as determined by MTE.

Benchmark	Blocks	Edges	Branches	Back Edges	MTE Relevant Edges
barnes-hut	475	730	246	49	45%
count-graphs	285	424	147	47	46%
knuth-bendix	209	317	101	28	47%
lexgen	1068	1753	629	121	47%
md5	243	363	116	31	45%
nucleic	124	179	51	22	38%
tsp	217	328	104	32	46%
tyan	759	1132	379	100	44%
zern	315	465	143	43	41%

identifying control to other blocks and passing arguments to the blocks. Some transfers define conditional control to multiple blocks (i.e., branches).

Our edge-string abstractions record the k-most recent, relevant edges of a path. The implementation uses the must-traverse-edge (MTE) analysis discussed in §3.3 to determine the relevance of edges and only extends edge strings with relevant edges, thus capturing infeasibility relationships across a potentially large number of path edges. Our analysis tracks dataflow through blocks, and control-flow through conditional transfers, using a join-over-paths abstraction that supports equality and inequality constraints over base types and datatype constructors.

Because MLton performs aggressive inlining of functions, leading to larger functions with potential many blocks, and a correspondingly large number of infeasible paths, we run the analysis after inlining. For the remainder of the section we write IESA_k to refer to a last relevant-k edge string analysis for detecting infeasible paths, where edge relevance is determined by MTE.

4.1 Benchmarks

We present results for nine of the benchmarks in the MLton benchmark suite. The benchmarks considered represent various programming styles and constructs that SML programs typically exhibit. In particular, `count-graphs` uses continuations extensively, `barnes-hut` and `nucleic` contain a large number of tail calls (i.e., loops) and `lexgen` and `tyan` perform repeated operations on lists.

After SSA inlining, programs typically contain a few large functions with more than 100 blocks and many small functions. Our analysis did not identify infeasible paths in many of the smaller functions. It did identify infeasible paths in all but one of the 20 functions we analyzed with more than 100 blocks. We present results for the `main` function—the function with the most blocks after inlining—for each benchmark (see Table 1). In §4.4 we examine what size k achieves the needed precision for infeasible path detection.

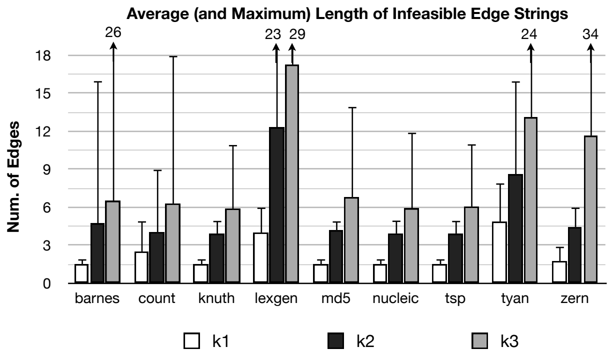


Fig. 2. We report the average and maximum number of edges along the shortest path from the first edge string to an infeasible edge for all infeasible paths identified by IESA_k.

4.2 Implementation Details

Alg. 1 was extended to explore the CFG G in reverse post-order, collapsing strongly connected components in G. Moreover, inner loops are explored before propagating the dataflow facts to the outer loop. We found that this ordering was essential for performance because it expedites the generation of edge strings. We also optimized the abstract domain to only interpret facts for variables that are in scope. In MLton's SSA, variable definitions dominate all uses, and thus traversing the dominator tree is sufficient to compute the in-scope variables at a CFG node. Restricting facts to in-scope variables greatly reduces the size of variable maps, resulting in better memory and time performance.

IESA_k uses a path-insensitive *must-traverse-edge* analysis (MTE) (explained in §3.3) to record only relevant edges for edge strings. MTE leads to increased precision for a given k. By only recording edges that are relevant, an edge string of length k may contain edges that are further than k edges away. Fig. 2 reports the average and maximum number of edges along the shortest path in the CFG from the the head of the first edge in the edge string to the head of an infeasible edge for all infeasible paths identified at $k = 1$ to 3. For example, in `barnes_hut` for $k = 3$ (i.e., IESA_3) the average length is roughly 7 and the maximum length is 26.

Fig. 3 illustrates the effectiveness of MTE. The figure shows infeasible paths corresponding to edge string $[e', e'']$ in `tyan`. The edge string implies that the edge e is infeasible for all paths that traverse edge e' and then e''. Note that the shortest path from e' to the infeasible edge e contains 6 edges, but the infeasible path is identified with IESA_2 because there are only two relevant edges, e' and e'', along the path of interest (i.e. traversal of the other edges is implied by the edge string). The infeasible path is on a branch on variable x_1963. Note that constraint-inducing edge, shown in box A, which produces constraint $x\_1963 = nil\_5$ is not in the edge string. This is because the edge must be traversed every time that e' is traversed and thus is not relevant according to MTE. Any edge string with edge e' will contain the constraint $x\_1963 = nil\_5$ causing the interpretation of edge e to be infeasible (i.e., $\perp$) for the edge string.

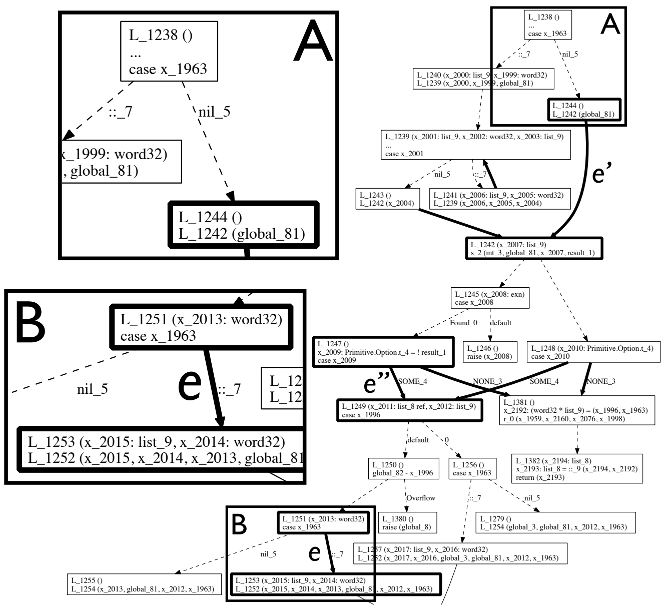


Fig. 3. Infeasible paths in `tyan` identified with IESA_2 with the MTE relevance. Relevant edges are bold; irrelevant edges are dashed. The edge string $[e', e'']$ creates constraints which results in the identification of edge e as infeasible. Interesting parts of the infeasible path are shown larger to demonstrate the path is infeasible on a branch on variable x_1963.

It is worth noting that IESA_2 identifies edge e as infeasible for all edge strings which contain e' as the first edge. Also note that IESA_1 would not identify edge e as being an infeasible branch, because all paths from edge e' to e traverse at least one relevant edge. This example demonstrates how a larger k results in greater precision.

4.3 Performance

Fig. 4 shows the performance for running the analysis on the main function of the benchmarks we considered. We report analysis costs for IESA_k for $k = 0$ to 3. The data was generated on a 2.53GHz Intel Core 2 Duo Processor with 4GB of RAM running Mac OS X 10.5.8.

The experimental results find that in general, the analysis time for IESA_k is exponential in k (i.e., the length of the edge string). The reason is that with MTE edge strings are only extended at join points (see the bold edges in Fig. 3). In the worst case all paths in the CFG join at the same join points. ESA provides a way to limit the analysis cost by using smaller edge strings (i.e., smaller k).

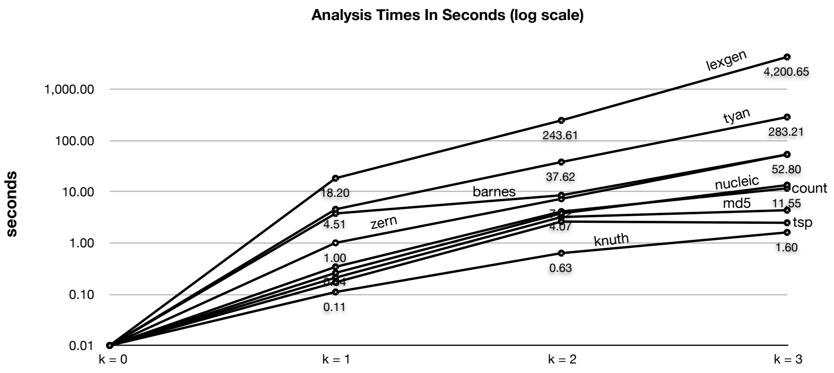


Fig. 4. Analysis times for IESA_k.

4.4 Edge String Length

We ran IESA_k for $k = 0$ to 3. As a property of ESA, all infeasible paths identified with edge strings of length k are identified with edge strings of length $k + 1$. While increasing k improves precision, it decreases performance. Thus, it is interesting to quantify increased precision as k increases.

For each k, we report the number and percentage of infeasible edge strings detected which were not detected (i.e., *covered*) at $k - 1$. We say that an infeasible edge string es of length k is covered by an edge string es' of length $k - 1$, if 1) both es and es' imply the same infeasible edge, and 2) the paths represented by es are a subset of the paths represented by es'. For example, the set of paths represented by edge string $[e_1, e_2, e_3]$ of length 3 is a subset of the paths represented by edge string $[e_1, e_2]$ of length 2 since the set of all paths that traverse edges e_1 and e_2 includes those paths that traverse e_3.

Consider the edge string $es = [e', e'']$ in Fig. 3 and the infeasible path through edge e. While the paths represented by es are a subset of paths represented by edge string $es' = [e'']$ of length one, we *do not* say that es' covers es because es' does not identify edge e as infeasible (the first condition we listed above). Thus es identifies new infeasible paths. On the other hand, all edge strings of length 3 which contain edge e' would be covered by the edge string $[e', e'']$ of length 2 for infeasible paths that traverse edge e.

Fig. 5 depicts how much precision is gained by increasing k for IESA_k. Recall that the paths represented by a single edge string of length $k - 1$ includes the paths represented by *many* edge strings of length k. We are interested in determining when more specific path data leads to the identification of infeasible paths not identified by the less specific edge strings. The percentage of edge strings that represent infeasible paths *not* discovered (i.e., covered) at $k - 1$ is presented in a logarithmic-scale bar graph. The number of new edge strings is also included directly above each bar.

Our results show that all infeasible paths identified by IESA_1 are not found by a path-insensitive (i.e., IESA_0) analysis. More interestingly our experiments show that 1) a large majority of infeasible paths identified by IESA_2 are found by IESA_1, and 2) an even larger majority of infeasible paths identified by IESA_3 are found with IESA_2.

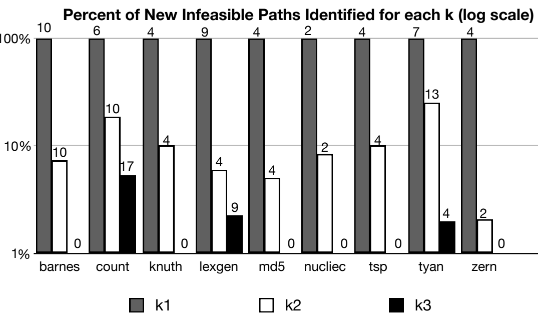
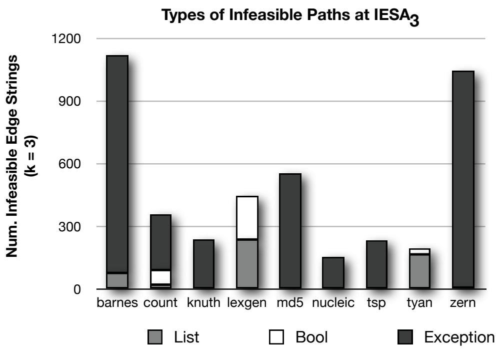


Fig. 5. The figure shows the percent of new infeasible edge strings discovered by increasing k for each benchmark (the number is also given above each bar). For all benchmarks, the infeasible paths identified by IESA_1 were not identified by a path insensitive (i.e., IESA_0) analysis. In all but 3 benchmarks (i.e., `count-graphs, lexgen`, and `tyan`) all infeasible paths identified by IESA_3 where covered by IESA_2.

The largest percentage of new infeasible edge strings discovered at $k = 2$ was 25% for `tyan`, and at $k = 3$ it was 5% for `count-graphs`. For 6 out of the 9 benchmarks, no new infeasible paths were discovered by IESA_3. While IESA_3 did identify new infeasible paths in the other 3 benchmarks, the number of such new paths is very small. For example, in `lexgen` IESA_3 identifies 444 infeasible edge strings. Only 2% (i.e., 9) of those infeasible edge strings are not represented by infeasible edge strings in IESA_2.

The experiments validate our intuition that while increasing k provides for greater precision, most of the benefits of full path sensitivity with respect to infeasible path detection can be achieved using ESA with small values of k (i.e., 2), avoiding greater analysis costs associated with larger values of k.

4.5 Infeasible Path Characteristics



Our experiments show that a large percentage of infeasible paths in MLton's SSA manifest on dispatches of datatype constructors (as shown in Fig. 1). For example, the infeasible edge string in `tyan` illustrated in Fig. 3 is on a dispatch of a list datatype.

Fig. 4.5 presents characteristics of detected infeasible paths. For each edge string of length $k = 3$ that detects infeasible paths, we report the different types of variables on which the infeasible path was detected. We distinguish between lists, exceptions, and booleans, which are all treated as datatypes in SSA. These account for more than 99% of detected infeasible paths.

The experiments found that 93% of all infeasible edge strings were a result of an infeasible branch deconstructing an exception type. Lists accounted for most of the remaining 7%. `Lexgen` and `tyan` were the only two benchmarks to mostly identify infeasible paths that deconstructed list types. `Lexgen`, which produces random lists of words based on a grammar, and `tyan`, which performs a Grobner Basis calculation to generate subsets in a polynomial ring, use lists extensively. We believe that elimination of infeasible paths that arise because of infeasible branches, as illustrated in our motivating example, and evidenced experimentally, detected by our analysis can be profitably used by subsequent optimization passes.

5 Related Work

There are many general frameworks for performing a path-sensitive analysis (cf. [9, 7, 6, 11]); in each case an original abstract domain is (more or less) made more precise by further partitioning the set of abstract states with an element from a finite set of qualifiers: a finite relation in [9]; a 4-tuple representing a set of program paths in [7]; a state from a property automaton in [6]; and a combination of user-specified partitions along with $\{0, 1, \infty\}$-partitioning of loops in [11]. For each of these techniques, edge strings, and hence edge-string analysis, could be obtained via the appropriate encoding. The main purpose of edge strings, however, is not to specify a generic framework, but to provide an effective path-sensitive extension to standard dataflow analyses found in modern compilers. Moreover, the design of edge strings is so that they are lightweight for both static *and* dynamic analysis (see §6).

Compared to heavyweight model-checking techniques that enumerate all paths in an abstract program (cf. [4, 2, 8]), we view edge strings as an efficient pre-enumeration analysis that can eliminate many infeasible paths before explicit path enumeration begins. In fact, the observation in [2] that many infeasible paths are witnessed by just two edges was one of our motivations for exploring edge-string analysis.

[3] presents an infeasible-path analysis that is then used to improve the precision of dataflow analysis. Their algorithm begins by working backwards from a particular branch to determine the set of predecessor nodes (if any) that imply the direction that the branch will take, i.e., to find correlated branch predecessors. Correlated-predecessor information is then propagated forwards through a CFG by marking paths in the CFG.

6 Conclusions and Future Work

This paper discusses edge strings, a technique inspired by call strings used to achieve context-sensitivity, that provides k-limited path-sensitivity to an existing dataflow analysis. We have implemented an edge-string analysis in MLton, and used it to detect many infeasible paths in SML programs that have been compiled into MLton's first-order SSA intermediate representation. Our results show that a substantial number of infeasible paths can be detected with just $k = 2$.

With respect to extensions of this work we are currently considering, we note that Sumner et al. [17] present a technique to efficiently track call-string information in a dynamic analysis. We intend to investigate notions of edge-string encodings, which would make available both interprocedural (via call strings) and intraprocedural (via edge strings) path information.

References

1. Augustsson, L.: Compiling Pattern Matching. In: Functional Programming Languages and Computer Architecture. pp. 368–381. Springer-Verlag New York, Inc. (1985)
2. Balakrishnan, G., Sankaranarayanan, S., Ivančić, F., Wei, O., Gupta, A.: SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In: SAS. pp. 238–254 (2008)
3. Bodík, R., Gupta, R., Soffa, M.L.: Refining data flow information using infeasible paths. In: FSE. pp. 361–377 (1997)
4. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252 (1977)
6. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: PLDI. pp. 57–68. ACM, New York, NY, USA (2002)
7. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: SAS. pp. 200–214 (1998)
8. Harris, W.R., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: POPL. pp. 71–82 (2010)
9. Holley, L.H., Rosen, B.K.: Qualified data flow problems. In: POPL. pp. 68–82 (1980), http://doi.acm.org/10.1145/567446.567454
10. Le Fessant, F., Maranget, L.: Optimizing Pattern Matching. In: ICFP '01. pp. 26–37. ACM (2001)
11. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: ESOP. pp. 5–20 (2005)
12. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press (1997)
13. MLton, http://mlton.org
14. Schmidt, D.A.: Data flow analysis is model checking of abstract interpretations. In: POPL. pp. 38–48 (1998)
15. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications. pp. 189–233 (1981)
16. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University (1991)
17. Sumner, W.N., Zheng, Y., Weeratunge, D., Zhang, X.: Precise calling context encoding. In: ICSE (May 2010)