On the Cubic Bottleneck in Subtyping and Flow Analysis

Nevin Heintze* David McAllester†

Abstract

We prove that certain data-flow and control-flow problems are 2NPDA-complete. This means that these problems are in the class 2NPDA and that they are hard for that class. The fact that they are in 2NPDA demonstrates the richness of the class. The fact that they are hard for 2NPDA can be interpreted as evidence they can not be solved in sub-cubic time — the cubic time decision procedure for an arbitrary 2NPDA problem has not been improved since its discovery in 1968.

1. Introduction

Cubic time complexity has become a common feature of algorithms for the automated analysis of computer programs. There is a general feeling that many of these algorithms are inherently cubic time — no sub-cubic procedure has been found. Such cubic time algorithms include Shivers' control flow analysis [17], the Palsberg and O'Keefe method of determining typability in the Amadio-Cardelli type system [15, 1], and various set-based analyses [5, 10, 11]. At an intuitive level the inherent cubic complexity in all these problems arises from the need to compute a dynamic transitive closure — one must compute the transitive closure of a directed graph while adding edges to the input graph as a consequence of edges derived for the output graph. Not only do these problems all seem inherently cubic, they all seem structurally similar and inherently cubic for the same reason.

In order to better understand the "cubic bottleneck" in flow analysis, Melski and Reps have investigated a simple data-flow reachability problem [13].[1] They relate this data-flow reachability problem to the problem of context-free-language reachability (CFL-reachability). An instance of the CFL-reachability problem consists of a context free grammar and a directed graph where each arc is labeled with a symbol from the terminal alphabet. The problem is to determine whether there is a path between two given nodes such that the sequence of labels on the arcs in that path is a string in the language generated by the given grammar. The CFL-reachability problem can be solved in $O(|G|n^3)$ time where $|G|$ is the size of the grammar (the number of productions in a Chomsky normal form grammar) and n is the number of nodes in the graph. Melski and Reps give a linear time reduction from data-flow reachability to CFL-reachability. This reduction produces a grammar of size n, so the reduction appears to yield an $O(n^4)$ method of solving data-flow reachability. However, Melski and Reps show that the reduction produces problems with special structure and that the overall running time of solving a data-flow problem by reduction to CFL-reachability is $O(n^3)$. More significantly, Melski and Reps give a reduction of CFL-reachability to data-flow reachability which runs in $O(|G|n)$ time. For a fixed grammar this reduction is linear time. If the data-flow reachability problem could be solved in sub-cubic time then the CFL-reachability problem over a fixed grammar could also be solved in sub-cubic time.

Here we investigate the cubic bottleneck by relating it to the class 2NPDA. 2NPDA is the class of languages (or problems) definable by a two way nondeterministic push-down automata. In 1968 it was shown that any problem in the class 2NPDA can be solved in cubic time [2]. But no sub-cubic procedure for an arbitrary 2NPDA problem is known. Neal has shown that a certain 2NPDA problem — ground monadic rewriting reachability (GMR-reachability) — is 2NPDA complete [14].[2] In other words, this problem is both in the class 2NPDA and is 2NPDA-hard, i.e., if GMR-reachability can be solved in sub-cubic time then all 2NPDA problems can be solved in sub-cubic time. We review Neal's result here. We also show that data-flow reachability, control-flow reachability, and the complement of Amadio-

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

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

[1] Following Heintze and Jaffar [4], Melski and Reps formulate this data-flow reachability problem as a set-constraint problem. We use the data-flow formulation here because it seems closer to applications.

[2] Neal uses a "monotone closure" formulation of GMR-problem. We find the GMR formulation more natural.

Cardelli typability are all in the class 2NPDA. Finally, by giving linear time reductions from GMR-reachability we show that each of these three problems is also 2NPDA-hard and hence 2NPDA-complete.

The fact that a variety of natural problems are in 2NPDA demonstrates the richness of this class. Intuitively, 2NPDA should be thought of as a "problem class" analogous to NP rather than a "language class" analogous to context free languages. We believe that 2NPDA-hardness can be used in practice in much the same way that NP-hardness is used — once a problem has been shown to be 2NPDA-hard one seems justified in abandoning the search for a sub-cubic algorithm.

2. GMR-Reachability is in 2NPDA

In this section we review the definition of the class 2NPDA and Neal's result that ground monadic rewriting reachability (GMR-reachability) is in the class 2NPDA. Before giving Neal's proof we show that traditional graph reachability is in 2NPDA. Graph reachability is the simple problem of determining whether one can go from a given start node to a given finish node in a given directed graph.

2NPDA is the class of languages (problems) accepted by a two way nondeterministic pushdown automata. The input string to a pushdown automata is written on a tape where the ends of the input are marked by a special tape symbol. The input tape is read only but the automata has access to a symbol stack of unbounded depth. The transition table of the (nondeterministic) automaton specifies a set of possible actions for each triple of a machine state, top stack symbol, and an tape symbol. An action can push or pop a symbol from the stack, move the read head left or right, and enter a new specified state. The automaton starts in a specified initial machine state, reading the first symbol in the input string, and with an empty stack. The machine accepts the input if there exists a sequence of allowed actions leading to an accepting machine state and an empty stack.

Languages defined by (one way) nondeterministic pushdown automata (NPDA) are precisely the context free languages. Membership in a context free language can be determined in sub-cubic time using sub-cubic procedures for matrix multiplication. But a 2NPDA can define languages such as $a^n b^n c^n$ which are not context free. It has been known since 1968 that, for any language defined by a 2NPDA, membership in that language is decidable in cubic time [2]. But no sub-cubic membership test for an arbitrary 2NPDA language is known.

We now show that simple directed graph reachability is in the class 2NPDA. In constructing a 2NPDA for graph reachability we assume the input is written on a tape using bit strings to represent nodes. For example, consider the graph with arcs $start \rightarrow a$, $b \rightarrow c$ and $a \rightarrow b$, $c \rightarrow finish$ where $start$ and $finish$ are the start and finish nodes respectively. This instance of the graph reachability problem can be represented by the following input tape.

$$\& * start \rightarrow 10 * 01 \rightarrow 11 * 10 \rightarrow 01 * 11 \rightarrow finish\&$$

Here & is the symbol delimiting the start and end of the input tape; the symbol $*$ is used to separate the representations of edges in the input graph; and each edge is represented by a string of the form $\alpha \rightarrow \beta$ where α and β are either one of the symbols $start$ or $finish$ or bit strings (strings in $\{0, 1\}^*$). We now describe a 2NPDA which will accept exactly those strings of this form in which the finish node is reachable from the start node in the represented graph. The automaton initially writes the symbol $start$ on the stack. It then enters a loop which iteratively rewrites the stack contents. In each iteration of the loop it nondeterministically moves the head to any position occupied by the symbol $\rightarrow$. In the above example it can move to the position of $\rightarrow$ in the substring $start \rightarrow 10$. It then walks to the left popping the stack as it goes and checking that the string on the stack matches the string to the left of the $\rightarrow$. In this case the string is the symbol $start$. If the match fails then the automaton fails — it enters an infinite loop failing state. If the match succeeds then the automaton walks right to the symbol $\rightarrow$ and then walks over the string to the right of $\rightarrow$ pushing that bit string on to the stack. In this example, the stack then contains 10. Another iteration of this nondeterministic rewriting loop can replace 10 by 01. A third iteration can replace 01 by $finish$. When the rewriting results in the stack containing the symbol $finish$ the automaton pops the stack and enters an accepting state. It is not difficult to see that the set of inputs accepted by this nondeterministic automaton are exactly those representations of graph reachability problems where $finish$ is reachable from $start$.

Of course simple graph reachability can be solved in linear time on a RAM by enumerating the reachable nodes. Ground monadic rewriting reachability is a more challenging problem. An instance of GMR-reachability consists of a set of rewrite rules of the form $L \rightarrow R$ where L and R are ground terms constructed from constants and monadic function symbols. The instance is solvable if one can use the given rewrite rules to rewrite the constant $start$ to the constant $finish$. For example, the following rewrite rules allow us to rewrite $start$ to $finish$.

$$start \rightarrow f(a)$$

$$a \rightarrow g(b)$$

$$b \rightarrow c$$

$$g(c) \rightarrow d$$

$$f(d) \rightarrow finish$$

In constructing a 2NPDA for GMR-reachability we represent both the constant symbols and the function symbols with bit strings. A monadic ground term is represented by a string of the form $\alpha_1(\alpha_2(\ldots\alpha_n(\beta)\ldots))$ where each α_i is a bit string and β is either one of the symbols $start$ or $finish$ or a bit string. A rewrite rule is represented by a string of the form $L \to R$ where L and R are strings representing terms. An instance is represented by a string of the form $*R_1 * R_2 * \ldots * R_n*$ where each R_i is a rewrite rule. The 2NPDA for solving GMR-reachability is very similar to the 2NPDA for solving simple graph reachability. The primary difference is that in the case of GMR-reachability the rewrite loop rewrites a fraction of the stack rather than the entire stack. The stack is used to represent a term reachable (under rewriting) from the symbol $start$. Trailing parentheses are omitted from the representation on the stack — a term on the stack is represented by a string of the form $\alpha_1(\alpha_2(\ldots\alpha_n(\beta$. The automaton starts by placing the symbol $start$ on the stack. It then enters an iterative rewrite loop. In each iteration it nondeterministically moves to some occurrence of $\to$ in a rewrite rule $L \to R$. It then moves left, popping the stack as it goes, and checking that L represents a topmost fraction of the current term. It then move right over the symbol $\to$ and replaces the removed term with the term R (minus the trailing parentheses). If the automaton ever reaches a state where the stack contains just $finish$ then it accepts. It is not hard to see that this automaton accepts exactly those inputs representing solvable instances of GMR-reachability.

We have now shown that GMR-reachability is in the class 2NPDA. This implies that GMR-reachability is solvable in cubic time. In the next section we review Neal's proof that GMR-reachability is 2NPDA-hard — if it can be solved in sub-cubic time then all problems in 2NPDA can also be solved in sub-cubic time.

3. GMR-Reachability is 2NPDA-Hard

Here we review Neal's result that GMR-reachability is 2NPDA hard. First we give some definitions.

Definition: A problem P is 2NPDA-hard if for any problem Q in 2NPDA there exists a many-one reduction from Q to P which can be computed on a RAM in $O(nR(\log n))$ time where R is a polynomial.

Definition: A problem P can be solved in sub-cubic time if there exists a RAM algorithm for solving P which runs in time $O(n^k)$ with $k < 3$.

Observation: If a 2NPDA-hard problem is solvable in sub-cubic time then all 2NPDA problems are solvable in sub-cubic time.

Of course the above observation does not imply that 2NPDA-hard problems are inherently cubic — sub-cubic procedures may exist. But no sub-cubic procedure has been found since the problem of 2NPDA membership was first investigated 30 years ago.

To prove that GMR-reachability is 2NPDA hard we consider an arbitrary 2NPDA M. We give a procedure for reducing any string s to an instance of GMR-reachability which runs in time $O(|M||s|)$ where $|M|$ is the size of the transition table of M and $|s|$ is the length of the string s. This reduction produces a representation of the GMR-reachability problem appropriate for a RAM — the constants and function symbols are represented by numbers and we assume that a fresh constant or function symbol can be created in $O(1)$ time. Writing the instance on a tape using bit strings for constant and function symbols adds a logarithmic factor to the running time of the reduction. The automaton M uses a finite set of allowed string symbols, a finite set of states, and a particular finite set of stack symbols. Consider a particular input string s. We represent a configuration of a run of M on s by a term of the form $f_1(f_2(\ldots f_k(c_{i,\gamma})))$ where $c_{i,s}$ is a constant symbol representing the head position i and the machine state γ, and f_1, f_2, ..., f_k is a sequence of function symbols representing the state of the push-down store with f_k at top of the store (the last symbol pushed). The set of possible configuration transitions during a run of M on s can be encoded in ground monadic rewrite rules. For example, suppose the given symbol at position i in s allows the machine to pop the symbol g from the push-down store when the machine is in state γ. Then we have the rewrite rule $g(c_{i,\gamma}) \to c_{j,\delta}$ where j is the next head position and δ is the next machine state. Rewrite rules representing push actions have the form $g(c_{i,\gamma}) \to g(f(c_{j,\delta}))$. The 2NPDA can always be modified so that if it accepts a given string then it reaches a standard accepting configuration with an empty stack and with the head positioned at the initial tape symbol. Now the given machine accepts the given input if and only if $c_{0,\delta}$ can be rewritten to $c_{0,\gamma}$ where δ is the initial state and γ is the accepting state. Note that the number of rewrite rules generated is at most $|M||s|$.

By increasing the number of machine states by a factor of $\log n$ we can convert any given machine to one that accepts the same language but uses only two push down symbols. When the above construction is applied to a machine with only two push down symbols the resulting monotone reachability problem has only two function symbols. This implies that GMR-reachability with only two function symbols is also 2NPDA-hard.

4. Data-Flow Reachability

In this section we show that the data-flow reachability problem studied by Melski and Reps [13] is 2NPDA-complete — it is both in 2NPDA and is 2NPDA-hard. First we show that the problem is linear time equivalent to GMR-reachability. This establishes 2NPDA-hardness. Unfortunately, it does not immediately imply membership in 2NPDA — the class 2NPDA appears not to be closed under linear-time equivalence in general. In this case however, the linear equivalence between data-flow reachability and GMR-reachability is very direct and leads immediately to a 2NPDA for data-flow reachability.

An instance of data-flow reachability is a set of "assignment statements" of the form $\langle y,\ z\rangle \Rightarrow x$, $\Pi_1(y) \Rightarrow x$, and $\Pi_2(y) \Rightarrow x$. Intuitively, $e \Rightarrow x$ means that the variable x can be assigned the value of the expression e. Π_1 and Π_2 are the left and right projection functions respectively, e.g., $\Pi_1(\langle x,\ y\rangle) = x$. To define solvability for data-flow reachability we use the following inference rules to derive new assignment statements of the form $e \Rightarrow x$ where e is any expression constructed from variables and the pairing function. [3]

$$\frac{e_1 \Rightarrow y \quad e_2 \Rightarrow z \quad \langle y,\ z\rangle \Rightarrow x}{\langle e_1,\ e_2\rangle \Rightarrow x} \qquad \frac{\langle e_1,\ e_2\rangle \Rightarrow y \quad \Pi_1(y) \Rightarrow x}{e_1 \Rightarrow x} \qquad \frac{\langle e_1,\ e_2\rangle \Rightarrow y \quad \Pi_2(y) \Rightarrow x}{e_2 \Rightarrow x}$$

$$x \Rightarrow x \qquad \frac{e \Rightarrow x \quad x \Rightarrow y}{e \Rightarrow y}$$

Let Σ be an instance of data-flow reachability, i.e., a set of assignment statements. Σ is solvable if the assignment $start \Rightarrow finish$ can be derived from Σ using the above rules where $start$ and $finish$ are distinguished variables. For example, the following instance of the data-flow reachability problem is solvable.

$$\langle start,\ x\rangle \Rightarrow y$$
$$\langle s,\ y\rangle \Rightarrow z$$
$$\Pi_2(z) \Rightarrow w$$
$$\Pi_1(w) \Rightarrow finish$$

First we give linear time reductions from data-flow reachability to GMR-reachability with the two function symbols Π_1 and Π_2. To reduce data-flow reachability to GMR-reachability we map an assignment of the form $\langle y,\ z\rangle \Rightarrow x$ to the two rewrite rules $y \rightarrow \Pi_1(x)$ and $z \rightarrow \Pi_2(x)$. We map an assignment of the form $\Pi_i(y) \Rightarrow x$ to the rewrite rule $\Pi_i(y) \rightarrow x$. The variables of the data-flow problem are treated as constants in the rewrite rules. For example, the translation of the above problem instance includes the rewrite rules $start \rightarrow \Pi_1(y)$ and $y \rightarrow \Pi_2(z)$. We leave the proof of correctness of this reduction as an exercise for the reader. This reduction implies that data-flow reachability is solvable in cubic time.

Next we give a linear time reduction of GMR-reachability with the two functions Π_1 and Π_2 to data-flow reachability. We first iteratively replace terms of the form $\Pi_i(t)$ where t is not a constant by $\Pi_i(c)$ where c is a fresh constant and then add the rules $t \rightarrow c$, and $c \rightarrow t$. This process terminates in linear time and yields a GMR-reachability instance where all terms appearing in the rewrite rules are either constants or terms of the form $\Pi_i(c)$. We then replace $\Pi_i(c) \rightarrow \Pi_j(b)$ by $\Pi_i(c) \rightarrow d$ and $d \rightarrow \Pi_j(b)$ and replace $c \rightarrow b$ by $c \rightarrow \Pi_1(d)$ and $\Pi_1(d) \rightarrow b$ where d is a fresh constant in both cases. So in linear time we can convert any instance of GMR-reachability with two function symbols to an instance of GMR-reachability where all rules are of the form $c \rightarrow \Pi_i(b)$ or $\Pi_i(b) \rightarrow c$. We then map a rule of the form $c \rightarrow \Pi_1(b)$ to the assignment $\langle c,\ d\rangle \Rightarrow b$ where d is fresh. Similarly, rewrite rules of the form $c \rightarrow \Pi_2(b)$ are mapped to $\langle d,\ c\rangle \Rightarrow b$ where d is fresh. Rewrite rules of the form $\Pi_i(c) \rightarrow b$ are mapped to $\Pi_i(b) \Rightarrow b$.

As mentioned above, the class 2NPDA appears not to be closed under linear time equivalence — 2NPDA problems can be linear time equivalent to problems which appear not to be in 2NPDA. So the linear time equivalence of data-flow reachability and GMR-reachability does not immediately imply that data-flow reachability is in 2NPDA. We now prove that data-flow reachability is in 2NPDA by giving an automaton for solving this problem. The Automaton is essentially derived from the reduction of data-flow reachability to GMR-reachability. We give only a brief description. As in the case of GMR-reachability, variables are represented by the symbols $start$ and $finish$ plus strings of the form $*\alpha*$ where α is a bit string. Assignments are represented by strings of the form $\langle\alpha, \beta\rangle \Rightarrow \gamma$ and $\Pi_i(\alpha) \Rightarrow \gamma$ where α, β and γ are strings representing variables and Π_1 and Π_2 are symbols. The problem instance is a string of the form $*A_1 * A_2 * \ldots * A_n*$ where each A_i is a string representing an assignment. The automaton uses the stack to represent terms of the form $\Pi_{i_1}(\Pi_{i_2}(\ldots\Pi_{i_n}(x)\ldots))$ in a manner analogous to that used in the automaton for GMR-reachability. In fact the automaton is identical to the one for GMR-reachability except that it treats an assignment statement $\langle\alpha, \beta\rangle \Rightarrow \gamma$ as a representation of the two rewrite rules $\alpha \rightarrow \Pi_1(\gamma)$ and $\beta \rightarrow \Pi_2(\gamma)$.

[3] To simplify the proofs we have used a formulation of data-flow reachability which is somewhat simpler than the one considered by Melski and Reps — they use data constructors of arbitrary arity and their flow problem corresponds to rules deriving assignments of the form $t \Rightarrow x$ where t is a closed term (one not containing variables). It is not difficult to show that the Melski-Reps formulation and the one given here are linear time equivalent. Furthermore, under an appropriate encoding of an instance of the Melski-Reps problem as an string, it is possible to show that the Melski-Reps version is also in the class 2NPDA.

5. Control-Flow Reachability is in 2NPDA

Next we consider a control-flow reachability problem. In this section we define the problem and show that it is in the class 2NPDA. The following two sections establish that control-flow reachability is also 2NPDA-hard.

The problem we formulate here is a pure λ-calculus version of Shivers' 0-CFA [17, 16]. The pure λ-calculus is defined by the following grammar.

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

Bound and free variables are defined in the usual way. An instance of the control-flow reachability problem is simply a term of the pure lambda calculus. A set of derived "flow arcs" between subterms of e is defined by the following three inference rules where antecedents of the form $\mathtt{INPUT}(u)$ signify that u is a subterm of the input term.

$$\frac{\mathtt{INPUT}((f\ w)) \quad f \rightarrow \lambda x.u}{x \rightarrow w,\ \ (f\ w) \rightarrow u} \qquad \frac{\mathtt{INPUT}(u)}{u \rightarrow u} \qquad \frac{e \rightarrow u \quad u \rightarrow w}{e \rightarrow w}$$

We can think of the arcs in the flow analysis as defining a kind of abstract rewrite relation, i.e., $u \rightarrow w$ signifies that if we repeatedly reduce the input term using β-reductions then some substitution instance of u may rewrite to a substitution instance of w. A formal relationship to environment evaluation can be found in [11].

An instance of the control flow reachability problem is solvable if the above rules can derive the flow arc $start \rightarrow finish$ where $start$ and $finish$ are distinguished variables occurring in the input λ-term.

All arcs derivable by the above rules are between subterms of the input term. The $O(n^2)$ derivable arcs can be enumerated by simply running the inference rules to completion in a forward chaining manner. A naive implementation of the transitivity rule takes $O(n^3)$ time so we get an $O(n^3)$ procedure.

We now describe a 2NPDA algorithm for control-flow reachability. We represent variables by strings of the form $*\alpha*$ where α is a bit string. We represent λ-expressions by strings of the form $\lambda\gamma\alpha\beta$ where γ is a bit string called a "label" and described below, α is a string representing a variable, and β is a string (recursively) representing a term. We represent an application as a string of the form $(\gamma\alpha\beta)$ where γ is a bit string representing a label and α and β are strings representing terms. Labels are often used in formulations of the control-flow problem and the flow arcs are often formulated as arcs between labels. However, the above flow rules manipulate terms rather than term occurrences or labels. We will assume that multiple occurrences of the same term on the input tape are given the same label. This allows a 2NPDA to nondeterministically move from one position on the tape at which a term u is written to any other position on the tape at which another copy of u is written. This is necessary if we want the automaton to implement flow rules which compute a direct graph on terms rather than a directed graph on term occurrences. If the control-flow problem were formulated in such a way as to generated directed arcs between term occurrences rather than terms, then the labels in the string representation of the problem would not be necessary. If we remove the labels from the string representation of the problem instance then the problem as formulated here (on terms rather than term occurrences) appears not to be in 2NPDA. This shows that membership in 2NPDA can be sensitive to the choice of the concrete representation and that 2NPDA appears not to be closed under simple problem reformulations.

The concrete representation of λ-expressions as strings uniquely determines the abstract syntax tree — every expression has a well defined terminating symbol. Furthermore, a 2NPDA can be made to manipulate the abstract syntax represented by this string. More specifically, a 2NPDA can use the stack to keep track of how many subexpressions it has descended into as it walks from left to right across a string. By walking across a term left to right, and then walking back, a 2NPDA can test whether the head is currently pointing at a well formed expression of a given type (i.e., an application, a variable, or a λ-expression). It can also deterministically move from a position representing an application to the position representing the operand of that application. Operations such as "find a λ-expression", "find an application expression", "move to the body of the current λ-expression", or "move to the operand of the current application" can be implemented within a 2NPDA. We can also implement the operation "move from this position to the position of the innermost expression properly containing this position". By writing the bit string for a given variable on the stack the 2NPDA can implement the operation "move to a λ-expression which binds the current variable". By writing the label of the current expression on the stack, the automaton can implement the operation "move to any other occurrence of the current term". This last operation appears to require labels.

The 2NPDA algorithm can be described by a set of recursively defined nondeterministic operations each of which takes a single argument which is a position on the input tape. The operation EVAL takes a position on the tape at which a term u is written and returns a position on the tape at which a term v is written such that $u \rightarrow v$ (as defined by the above inference rules). Intuitively the operation EVAL should be viewed as an "evaluator". The operation $\mathtt{EVAL}^{-1}$ takes a position at which a term u is written and returns (the position of) a term v such that $v \rightarrow u$. The operations are defined by rewrite rules defining a rewrite relation $\Rightarrow$ (not to be confused with the reduction relation $\rightarrow$ used in the definition of control-flow reachability). Often more than one rewrite rule applies. In this case either rule can be selected — the operations are nondeterministic. Nondeterminism is, of course, easily implemented in a 2NPDA. All terms mentioned in these rules must be explicitly written on the input tape.

$$\mathtt{EVAL}(u) \Rightarrow u$$
$$\mathtt{EVAL}((f\ a)) \Rightarrow \mathtt{EVAL}(\mathtt{BODY}(\mathtt{EVAL}(f)))$$
$$\mathtt{EVAL}(x) \Rightarrow \mathtt{EVAL}(\mathtt{ARG}(\mathtt{EVAL}^{-1}(\lambda x.e)))$$

$$\mathtt{BODY}(\lambda x.e) \Rightarrow e$$
$$\mathtt{ARG}(f) \Rightarrow a \quad \text{where the tape contains } (f\ a)$$

$$\mathtt{EVAL}^{-1}(u) \Rightarrow u$$
$$\mathtt{EVAL}^{-1}(u) \Rightarrow \mathtt{EVAL}^{-1}(\mathtt{APP}(\mathtt{EVAL}^{-1}(\lambda x.u)))$$
$$\mathtt{EVAL}^{-1}(u) \Rightarrow \mathtt{EVAL}^{-1}(\mathtt{VAR}(\mathtt{EVAL}(f))))$$
$$\text{where the tape contains } (f\ u)$$

$$\mathtt{APP}(f) \Rightarrow (f\ a)$$
$$\mathtt{VAR}(\lambda x.e) \Rightarrow x$$

These rewrite rules can be implemented in a 2NPDA by using the stack to store the sequence of pending operations. We need to check that all the head movements implicit in the above rules can be implemented in a 2NPDA. For example, the head movement from u to $\lambda x.u$ in the third EVAL rule, or the head movement from u to f in the third $\mathtt{EVAL}^{-1}$ rule. We leave this as an exercise for the reader.

The above rules immediately satisfy the specifications that if $\mathtt{EVAL}(u) \Rightarrow v$ then $u \rightarrow v$ and if $\mathtt{EVAL}^{-1}(u) \Rightarrow v$ then $v \rightarrow u$. The converse is proved by induction on length of the proof of arcs of the form $u \rightarrow v$ using the inference rules defining the control-flow reachability problem. We want to prove that if $u \rightarrow v$ then $\mathtt{EVAL}(u) \Rightarrow v$ and $\mathtt{EVAL}^{-1}(v) \Rightarrow u$. This is done by assuming the result for the antecedents of the inference rules and then examining all the ways that arcs of the form $u \rightarrow v$ can be derived. The reflexivity and transitivity rules are essentially immediate. In the β-reduction rule we have that if the input contains $(f\ a)$ and $f \rightarrow \lambda x.e$ then $(f\ a) \rightarrow e$. From the antecedent $f \rightarrow \lambda x.e$ we can assume that $\mathtt{EVAL}(f) \rightarrow \lambda x.e$ This implies

$$\mathtt{EVAL}(\mathtt{BODY}(\mathtt{EVAL}(f))) \Rightarrow e$$

so we get that $\mathtt{EVAL}((f\ a)) \Rightarrow e$. From the antecedent we can also assume that $\mathtt{EVAL}^{-1}(\lambda x.e) \Rightarrow f$. This implies that

$$\mathtt{EVAL}^{-1}(\mathtt{APP}(\mathtt{EVAL}^{-1}(\lambda x.u))) \Rightarrow (f\ a)$$

from which we can infer that $\mathtt{EVAL}^{-1}(e) \Rightarrow (f\ a)$. Finally, the β-reduction rule also states that if $f \rightarrow \lambda x.e$ and the input contains $(f\ a)$ then $x \rightarrow a$. We again use the induction hypothesis to show that in this case $\mathtt{EVAL}(x) \Rightarrow a$ and $\mathtt{EVAL}^{-1}(a) \Rightarrow x$.

Given that the above rules can be implemented in a 2NPDA and that they correctly capture the notion of control-flow reachability, it is easy to design a 2NPDA which accepts a term provided that $start \rightarrow finish$ where $start$ and $finish$ are distinguished variables. So control flow reachability is in the class 2NPDA.

6. Bimonotone Reachability

Our proof that control-flow reachability is 2NPDA-hard uses a preliminary lemma stating that another problem — bimonotone reachability — is 2NPDA-hard. Before discussing bimonotone reachability we first define the problem of monotone reachability. An instance of monotone reachability is a set of inequalities $s \leq t$ where s and t are terms built from constants and monadic functions. An instance of monotone reachability is solvable if the inequality $start \leq finish$ is derivable from the given inequalities using the inference rules of reflexivity ($x \rightarrow x$), transitivity (if $x \leq y$ and $y \leq z$, then $x \leq z$), and monotonicity (if $x \leq y$ then $f(x) \leq f(y)$). It is not difficult to see that monotone reachability is actually just a syntactic variant of GMR-reachability where the inequalities can be viewed as rewrite rules.

An instance of bimonotone reachability is a set of inequalities of the form $s \leq t$ where s and t are terms constructed from constants and the two functions f and g. An instance E of the bimonotone reachability problem is solvable if the inequality $start \leq finish$ can be derived from the given inequalities in E using the inference rules of transitivity plus the monotonicity of f (that if $x \leq y$ then $f(x) \leq f(y)$) and the anti-monotonicity of g (that if $x \leq y$ then $g(y) \leq g(x)$). Although we do not give the proof here, it is possible to show that bimonotone reachability is in the class 2NPDA. We now give an explicit proof that bimonotone reachability is 2NPDA-hard.

We prove that bimonotone reachability is 2NPDA-hard by a reduction of monotone reachability with two function symbols. Since monotone reachability is a syntactic variant of GMR-reachability, monotone reachability with two function symbols is 2NPDA-hard. Consider an instance E of monotone reachability with the two function symbols f and g. To reduce this problem to bimonotone reachability we first introduce a new constant c' for each constant c in E. Then we map each inequality $s \leq t$ in E to the pair of inequalities $tran(s) \leq tran(t)$ and $tran'(t) \leq tran'(s)$ where the operations $tran$ and $tran'$ are defined as follows.

$$tran(c) = c, \;\; tran(f(s)) = f(tran(s)), \;\; tran(g(s)) = g(tran'(s))$$
$$tran'(c) = c', \;\; tran'(f(s)) = f(tran'(s)), \;\; tran'(g(s)) = g(tran(s))$$

Let E' be this translation of E into bimonotone reachability. Intuitively, c' is the negative of c so that $c' \leq b'$ if and only if $b \leq c$. We say that a term is monotone if it is a constant of the form c or a term of the form $f(s)$ where s is monotone or a term of the form $g(s)$ where s is anti-monotone — a term is anti-monotone if it is a constant of the form c', or a term of the form $f(s)$ where s is anti-monotone, or a term of the form $g(s)$ where s is monotone. Intuitively, monotone terms represent monotone functions of c and anti-monotone terms represent anti-montone functions of c. Note that *tran* always produces a monotone term and *tran'* always produces an anti-monotone term. An inequality $s \leq t$ will be called well formed if either both s and t are monotone or both s and t are anti-monotone. All inequalities in E' are well formed. Furthermore the inference rules defining bimonotone reachability all preserve well-formedness. An inequality derivable from E' will be called monotone if both terms in that inequality are monotone and anti-monotone if both terms are anti-monotone. Now for any term s we define $|s|$ (the absolute value of s) to be the result of replacing each constant of the form c' by the constant c. One can prove by induction on the inference rules that if $s \leq t$ is a monotone inequality derivable from E' then $|s| \leq |t|$ is derivable from E and that if $s \leq t$ is a anti-monotone inequality derivable from E' then $|t| \leq |s|$ is derivable from E. This implies that if $start \leq finish$ is derivable from E' then it is also derivable from E. Conversely, by induction on the inference rules for monotone reachability we can show that if $s \leq t$ is derivable from E then both $tran(s) \leq tran(t)$ and $tran'(t) \leq tran'(s)$ are derivable from E'. Hence if $start \leq finish$ is derivable from E then it is also derivable from E'.

It is not difficult to see that bimonotone reachability remains 2NPDA-hard if we include the inference rule of reflexivity, i.e., $x \leq x$ for any x.

7. Control-Flow Reachability is 2NPDA-Hard

Now we prove that control-flow reachability is 2NPDA-hard by giving a linear time reduction of bimonotone reachability. Let E be the set of inequalities of a bimonotone reachability problem where the monotone operations is called ran and the anti-monotone operation is called dom (for range and domain respectively). The reduction builds a "machine" which uses the control-flow rules to implement the bimonotone rules. Unfortunately the machine is somewhat involved.

We will use $\perp$ as an abbreviation for the λ-term $(\lambda h.(h\ h)\ \lambda h.(h\ h))$. Note that the control-flow inference rules derive $\perp \rightarrow (h\ h)$ but no other arcs from $\perp$. α-variants of the identity function $\lambda x.x$ can be used to "store" sets of terms. Consider a term of the form $((\lambda s.b)\ \lambda x.x)$. This will generate the arc $s \rightarrow \lambda x.x$. Inside the term b the variable s can be used as a set into which items can be inserted and out of which items can be extracted. To insert an item v into the set s we include in b the application $(s\ v)$. Elements can be extracted from the set represented by s using the expression $(s\ \perp)$. More precisely, the rules derive $(s\ \perp) \rightarrow v$ for each v that is inserted into the set s.

Note that the term $((\lambda s.b)\ \lambda x.x)$ can be viewed as a definition of the variable s where s can then be used in b. More generally we will use

$$\textbf{let } x_1 = e_1; \ldots; x_k = e_k \textbf{ in } e$$

as an abbreviation for the following.

$$((\lambda x_1.((\lambda x_2.\ldots((\lambda x_k.e)\ e_k)\ \ldots)\ e_2))\ e_1)$$

Often we want to construct a term containing a certain set of subterms without regard for how those subterms are included except that "side effects" of the inclusion should be avoided. For terms $e_1, \ldots, e_n$ we take the expression $\{e_1, \ldots, e_n\}$ to be an abbreviation for the *term*

$$\textbf{let } y_1 = e_1; \ldots; y_n = e_n \textbf{ in } \perp$$

where $y_1, \ldots, y_n$ are fresh variables.

The reduction can now be defined as follows. Consider a bimonotone reachability problem E where *ran* is the monotone function and *dom* is the anti-montone function. We can assume without loss of generality that every inequality has the form $b \leq t$ or $t \leq b$ where b is a constant symbol and t is of the form $dom(c)$ or $ran(c)$ where c is a constant symbol (we can replace any term of the form $f(s)$ by $f(c)$ where we add the inequalities $c \leq s$ and $s \leq c$). We will also assume that the reflexivity rule ($x \leq x$) is included in the rules for bimonotone reachability. We map the bimonotone reachability problem E to the following λ-term.

$$\begin{array}{ll}
\textbf{let} & Id_c = \lambda x.x,\ c \text{ in } E; \\
& Id_{dom(c)} = \lambda x.x,\ c \text{ in } E; \\
& Id_{ran(c)} = \lambda x.x,\ c \text{ in } E; \\
& \text{in } \{(Id_c\ \lambda y.\{(Id_{dom(c)}\ y)\},\ c \text{ in } E, \\
& \quad ((Id_c\ \perp)\ (Id_{dom(c)}\ \perp)),\ c \text{ in } E, \\
& \quad (Id_c\ \lambda y.(Id_{ran(c)}\ \perp)),\ c \text{ in } E, \\
& \quad (Id_{ran(c)}\ ((Id_c\ \perp)\ \perp))),\ c \text{ in } E, \\
& \quad (Id_s\ (Id_t\ \perp)),\ s \leq t \in E\}
\end{array}$$

We assume that distinct bound variables in the above term, such as the formal parameters to the various identity functions, are α-renamed so as to be distinct. We also assume that the bound variable of Id_{start} is $start$ and the the bound variable of Id_{finish} is $finish$. We let P be the above λ-term (program). We will use the notation $u \in P$ to mean that the λ-term u is a subterm of P. The control flow problem is to determine whether $start \rightarrow finish$ can be derived using the control-flow inference rules applied to this term.

We now prove the correctness of the reduction. For each expression s of the form c, $dom(c)$, or $ran(c)$ in E we have that P contains an identity function Id_s. We define the variable x_s to be the bound variable of the identity function Id_s. We now show that, for any terms s and t appearing in E, if $s \leq t$ is derivable from E then $x_s \to x_t$ is derivable from P. This implies that if $start \to finish$ is derivable from E then it is also derivable from P. Consider an inequality $s \leq t$ in E. The last line of the definition of P above ensures that

$$(Id_s\ (Id_t\ \bot)) \in P.$$

The flow rules then derive the following.

$$x_s \to (Id_t\ \bot) \to x_t$$

Hence we have that $x_s \to x_t$ is derivable from P. We now consider inequalities of the form $s \leq t$ which are derivable from E and where s and t appear in E (and hence x_s and x_t appear in P). It can be shown that if s and t are terms appearing in E and $s \leq t$ is derivable from E then there exists a derivation of $s \leq t$ such that every term in that derivation appears in E [12, 9]. We now prove by induction on these "local" derivations that if $s \leq t$ is derivable from E then $x_s \to x_t$ is derivable from P. We need to consider each inference rule in the definition of bimonotone reachability. In each case we can assume that terms appearing in the antecedent of the rule also appear in E and that the invariant holds for these antecedents. The reflexivity and transitivity rules are trivial since these rules are also rules of flow-analysis. We now consider the tonicity rules for dom and ran.

Now suppose that $s \leq t$ is derived by the rule expressing the anti-monotonicity of the function *dom* where s and t are in E. We have assumed that all terms in E are either constants or applications of functions to constants. Hence $s \leq t$ must be of the form $dom(b) \leq dom(c)$ where b and c are constants appearing in E and where, by the induction hypothesis, $x_c \to x_b$ is derivable from P. We need to show that $x_{dom(b)} \to x_{dom(c)}$ is derivable from P. We have that

$$(Id_b\ \lambda y.\{(Id_{dom(b)}\ y)\}) \in P$$

from which it follows that

$$x_c \to x_b \to \lambda y.\{(Id_{dom(b)}\ y)\})$$

But we also have

$$((Id_c\ \bot)\ (Id_{dom(c)}\ \bot)) \in P$$

from which we have

$$(Id_c\ \bot) \to x_c \to \lambda y.\{(Id_{dom(b)}\ y)\}$$

and hence

$$y \to (Id_{dom(c)}\ \bot) \to x_{dom(c)}$$

Finally we have

$$(Id_{dom(b)}\ y) \in P$$

and hence

$$x_{dom(b)} \to y \to x_{dom(c)}$$

Suppose that $ran(c) \leq ran(b)$ is derived by an application of the monotonicity rule for ran where the terms $ran(c)$ and $ran(b)$ both appear in E. In this case the induction hypothesis gives $x_c \to x_b$. We need to show that the control-flow rules derive $x_{ran(c)} \to x_{ran(b)}$ from the program P. We have

$$(Id_b\ \lambda y.(Id_{ran(b)}\ \bot)) \in P$$

so we have

$$x_c \to x_b \to \lambda y.(Id_{ran(b)}\ \bot)$$

We also have

$$((Id_c\ \bot)\ \bot) \in P$$

from which we have

$$(Id_c\ \bot) \to x_c \to \lambda y.(Id_{ran(b)}\ \bot)$$

and hence

$$((Id_c\ \bot)\ \bot) \to (Id_{ran(b)}\ \bot) \to x_{ran(b)}.$$

But we also have

$$(Id_{ranc}\ ((Id_c\ \bot)\ \bot)) \in P$$

So finally we get

$$x_{ran(c)} \to x_{ran(b)}.$$

We have now proved that if $start \leq finish$ is derivable from E then it also derivable from P. We omit the proof of the converse.

8. Amadio-Cardelli Typability

Finally we show that typability in the Amadio-Cardelli system of recursive types is co2NPDA-complete (the set of untypable terms is 2NPDA-complete). In the Amadio-Cardelli system all pure λ-terms are typable. We therefore now consider a modified term grammar which includes the numerical constant 0.

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

By the recently established equivalence of control-flow analysis and recursive types [15] a term e is typable in the Amadio-Cardelli system if and only if there is no application $(f\ w)$ in e such that the control-flow rules of the previous section derive $f \to 0$, i.e., there is no attempt to apply the numerical constant 0 as a procedure. We can construct a 2NPDA which nondeterministically selects an application $(f\ a)$ and then accepts the overall input if $\mathtt{EVAL}(f) \Rightarrow 0$. This automaton will accept exactly the untypable terms. So the set of untypable terms is in the class 2NPDA. We now show that the set of untypable terms is 2NPDA-hard.

In order to make terms potentially untypable we need to include the non-procedure 0 in a variant of the reduction used in the previous section. This is done by using pairs of a procedure part and a "flag" part where the flag part is either the harmless term $\perp$ or the dangerous term 0. We use the notation $\langle u,\ w\rangle$, representing the pair of u and w, as an abbreviation for the term $\lambda z.((z\ u)\ w)$. We let $\mathit{first}(p)$ be an alternate notation for $(p\ \lambda xy.x)$ and we let $\mathit{second}(p)$ notate $(p\ \lambda xy.y)$.

Now given an instance E of the bimonotone reachability problem with target arc $n \to m$ we construct the following term.

$$
\begin{array}{ll}
\textbf{let} & Id_c = \lambda x.x,\ c \text{ in } E; \\
& Id_{dom(c)} = \lambda x.x,\ c \text{ in } E; \\
& Id_{ran(c)} = \lambda x.x,\ c \text{ in } E; \\
& \text{in } \{(Id_c\ \langle \lambda y.\{(Id_{dom(c)}\ y)\},\ \perp\rangle),\ \text{for } c \text{ in } E, \\
& \quad (\mathit{first}((Id_c\ \perp))\ (Id_{dom(c)}\ \perp)),\ c \text{ in } E, \\
& \quad (Id_c\ \langle \lambda y.(Id_{ran(c)}\ \perp),\ \perp\rangle),\ c \text{ in } E, \\
& \quad (Id_{ran(c)}\ (\mathit{first}((Id_c\ \perp))\ \perp)),\ c \text{ in } E, \\
& \quad (Id_s\ (Id_t\ \perp)),\ s \le t \in E, \\
& \quad (Id_{start}\ \langle \perp,\ 0\rangle), (\mathit{second}((Id_{finish}\ \perp))\ \perp)\}
\end{array}
$$

This term is similar to the one used in the previous section except that pairs are used here where functions were used before. The first component of each pair is a function that plays the same role as the corresponding function in the previous term. However, using pairs allows the second component of the pair to carry a "bomb" which can explode if it reaches the appropriate place. Note that the only occurrence of second is in the very last line of the above term. The second of most terms is $\perp$. The only way to get 0 as a value, i.e., to generate arcs of the for $u \to 0$, is if second is applied to the pair $\langle \perp,\ 0\rangle$ which also appears in the last line. Now if the bimonotone inference procedure fails to derive the target arc then the flow analysis will fail to generate any arcs of the form $u \to 0$. So the term is typable. On the other hand, if the bimonotone reachability generates the target arc then the flow analysis generates $\mathit{second}((Id_c\ \perp)) \to 0$ and the term is untypable.

9. Conclusions

We have investigated the cubic bottleneck in data-flow, control-flow, and typability problems from the perspective of the problem class 2NPDA. We have provided evidence that a sub-cubic procedure for any of these problems will be difficult to find — such a procedure would imply a sub-cubic procedure for all 2NPDA problems and no such procedure has been discovered in the almost 30 that 2NPDA has been investigated. We have also shown that these problems are in the class 2NPDA. This shows that 2NPDA is a fairly rich class. At an intuitive level, the richness of the class 2NPDA provides evidence that 2NPDA-complete problems are intrinsically hard.

References

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

[2] A. Aho, J. Hopcroft, and J. Ullmann, "Time and tape complexity of pushdown automaton languages", *Information and Control*, vol. 13, no. 3, pp. 186-206, 1968.

[3] A. Bondorf and J. Jorgensen, "Efficient analysis for realistic off-line partial evaluation", *Journal of Functional Programming*, Vol. 3, No. 3, July 1993.

[4] N. Heintze and J. Jaffar. "A Finite Presentation Theorem for Approximating Logic Programs", POPL90, pp 197-209.

[5] N. Heintze, "Set-Based Analysis of ML Programs", *ACM Conference on Lisp and Functional Programming*, pp 306–317, 1994.

[6] N. Heintze, "Control-Flow Analysis and Type Systems", *Static Analysis Symposium*, 1995, pp 189–206.

[7] N. Jones, "Flow Analysis of Lambda Expressions", *Symp. on Functional Languages and Computer Architecture*, pp. 66-74, 1981.

[8] N. Jones, "Flow Analysis of Lazy Higher-Order Functional Programs", in *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin (Eds.), Ellis Horwood, 1987.

[9] Robert Givan and David McAllester. New results on local inference relations. In *Principles of Knowlwedge Representation and Reasoning: Proceedings of the Third International Conference*, pages 403–412. Morgan Kaufman Press, October 1992. internet file ftp.ai.mit.edu:/pub/users/dam/kr92.ps.

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

[11] David McAllester and Nevin Heintze, "On the compexity of set-based analysis", (http://www.ai.mit.edu/people/dam/setbased.ps)

[12] D. McAllester and R. Givan, "Taxonomic syntax for first order inference", *JACM*, 40(2):246–283, April 1993. (ftp.ai.mit.edu:/pub/users/dam/jacm1.ps)

[13] E. Melski and T. Reps, "Interconvertability of Set Constraints and context-Free Language Reachability", PEPM'97, to appear.

[14] R. Neal, "The computational complexity of taxonomic inference", 1989, (ftp://ftp.cs.utoronto.ca/pub/radford/taxc.ps.Z).

[15] J. Palsberg and P. O'Keefe, "A Type System Equivalent to Flow Analysis", *POPL-95*, pp. 367–378, 1995.

[16] J. Palsberg and M. Schwartzbach, "Safety Analysis versus Type Inference" *Information and Computation*, Vol. 118, No. 1, pp. 128-141, April 1995.

[17] O. Shivers, "Control Flow Analysis in Scheme", *Proc. 1988 ACM Conf. on Programming Language Design and Implementation*, Atlanta, pp. 164–174, June 1988.