# Stack Size Analysis for Interrupt-driven Programs [1]

Krishnendu Chatterjee [a] Di Ma [c] Rupak Majumdar [e]
Tian Zhao [d] Thomas A. Henzinger [a,b] Jens Palsberg [e,*]

[a] *Department of Electrical Engineering and Computer Sciences*
*University of California, Berkeley, CA 94720, USA*
*{c_krish,tah}@eecs.berkeley.edu*

[b] *School of Computer and Communication Sciences*
*École Polytechnique Fédérale de Lausanne, Switzerland*

[c] *Department of Computer Science*
*Purdue University, West Lafayette, IN 47907, USA*
*madi@cs.purdue.edu*

[d] *Department of Computer Science*
*University of Wisconsin, Milwaukee, WI 53211, USA*
*tzhao@cs.uwm.edu*

[e] *Department of Computer Science*
*University of California, Los Angeles, CA 90095, USA*
*{rupak,palsberg}@cs.ucla.edu*

**Abstract**

We study the problem of determining stack boundedness and the exact maximum stack size for three classes of interrupt-driven programs. Interrupt-driven programs are used in many real-time applications that require responsive interrupt handling. In order to ensure responsiveness, programmers often enable interrupt processing in the body of lower-priority interrupt handlers. In such programs a programming error can allow interrupt handlers to be interrupted in a cyclic fashion to lead to an unbounded stack, causing the system to crash. For a restricted class of interrupt-driven programs, we show that there is a polynomial-time procedure to check stack boundedness, while determining the exact maximum stack size is PSPACE-complete. For a larger class of programs, the two problems are both PSPACE-complete, and for the largest class of programs we consider, the two problems are PSPACE-hard and can be solved in exponential time. While the complexities are high, our algorithms are exponential only in the number of handlers, and polynomial in the size of the program.

*Key words:* Program analysis, stack bounds, interrupt programs.

## 1 Introduction

Most embedded software runs on resource-constrained processors, often for economic reasons. Once the processor, RAM, etc. have been chosen for an embedded system, the programmer has to fit everything into the available space. For example, on a Z86 processor, the stack exists in the 256 bytes of register space, and it is crucial that the program does not overflow the stack, corrupting other data. Estimating the stack size used by a program is therefore of paramount interest to the correct operation of these systems. A tight upper bound is necessary to check if the program fits into the available memory, and to prevent precious system resources (e.g., registers) from being allocated unnecessarily.

Stack size analysis is particularly challenging for *interrupt-driven software*. Interrupt-driven software is often used in embedded real-time applications that require fast response to external events. Such programs usually have a fixed number of external interrupt sources, and for each interrupt source, a handler that services the interrupt. When an external interrupt occurs, control is transferred automatically to the corresponding handler if interrupt processing is enabled. To maintain fast response, interrupts should be enabled most of the time, in particular, higher-priority interrupts are enabled in lower-priority handlers. Interrupt handling uses stack space: when a handler is called, a return address is placed on the stack, and if the handler itself gets interrupted, then another return address is placed on the stack, and so on. A programming error occurs when the interrupt handlers can interrupt each other indefinitely, leading to an unbounded stack. Moreover, since stack boundedness violations may occur only for particular interrupt sequences, these errors are difficult to replicate and debug, and standard testing is often inadequate. Therefore, algorithms that statically check for stack boundedness and automatically provide precise bounds on the maximum stack size will be important development tools for interrupt-driven systems.

In this paper, we provide algorithms for the following two problems (defined formally in Section 2.3) for a large class of interrupt-driven programs:

- **Stack boundedness problem.** Given an interrupt-driven program, the stack boundedness problem asks if the stack size is bounded by a finite constant. More precisely, given a program $p$, the stack boundedness problem returns "yes" if there exists a finite integer $K$ such that on all executions of

[*] A preliminary version of this paper appeared in the *Proceedings of the Static Analysis Symposium (SAS 2003), Lecture Notes in Computer Science 2694, Springer-Verlag, pages 109–126, 2003.*

[*] Corresponding Author

the program $p$, the stack size never grows beyond $K$, and "no" if no such $K$ exists.

- **Exact maximum stack size problem.** Given an interrupt-driven program, the exact maximum stack size problem asks for the maximum possible stack size. More precisely, given a program $p$, the exact maximum stack size problem returns an integer $K$ such that for all executions of the program $p$, the stack size never grows beyond $K$, and such that there is a possible schedule of interrupts and an execution of the program $p$ such that the stack size becomes $K$; the problem returns $\infty$ if there is an execution where the stack can grow unbounded.

We model interrupt-driven programs in the untyped *interrupt calculus* of Palsberg and Ma [4]. The interrupt calculus contains essential constructs for programming interrupt-driven systems. For example, we have found that the calculus can express the core aspects of seven commercial micro-controllers from Greenhill Manufacturing Ltd. A program in the calculus consists of a main part and some interrupt handlers. In the spirit of such processors as the Intel MCS-51 family (8051, etc.), Motorola Dragonball (68000 family), and Zilog Z86, the interrupt calculus supports an interrupt mask register (imr). An imr value consists of a master bit and one bit for each interrupt source. For example, the Motorola Dragonball processor can handle 22 interrupt sources. An interrupt handler is enabled, if *both* the master bit and the bit for that interrupt handler is set. When an interrupt handler is called, a return address is stored on the stack, and the master bit is automatically turned off. At the time of return, the master bit is turned back on (however, the handler can turn the master bit on at any point). A program execution has access to:

- the interrupt mask register, which can be updated during computation,
- a stack for storing return addresses, and
- a memory of integer variables; output is done via memory-mapped I/O.

Each element on the stack is a return address. When we measure the size of the stack, we simply count the number of elements on the stack. Our analysis is approximate: when doing the analysis, we ignore the memory of integer variables and the program statements that manipulate this memory. In particular, we assume that both branches of a conditional depending on the memory state can be taken. Of course, all the problems analyzed in this paper become undecidable if integer variables are considered in the analysis, since we can then easily encode two-counter machines.

We consider three versions of Palsberg and Ma's interrupt calculus, here presented in increasing order of generality:

- **Monotonic programs.** These are interrupt calculus programs that satisfy the following monotonicity restriction: when a handler is called with an imr

| **Calculus** | **Problem** | **Complexity** | **Reference** |
|---|---|---|---|
| Monotonic | Stack boundedness | NLOGSPACE-complete | Theorem 7 |
| | Exact maximum stack size | PSPACE-complete | Theorems 13,25 |
| Monotonic | Stack boundedness | PSPACE-complete | Theorems 22,25 |
| (enriched) | Exact maximum stack size | PSPACE-complete | Theorems 13,25 |
| Enriched | Stack boundedness | PSPACE-hard, EXPTIME | Theorems 22,30 |
| | Exact maximum stack size | PSPACE-hard, EXPTIME | Theorems 13,30 |

Table 1
Complexity results

value $imr_b$, then it returns with an imr value $imr_r$ such that $imr_r \leq imr_b$, where $\leq$ is the logical bitwise implication ordering. In other words, every interrupt that is enabled upon return of a handler must have been enabled when the handler was called (but could have possibly been disabled during the execution of the handler).

- **Monotonic enriched programs.** This calculus enriches Palsberg and Ma's calculus with conditionals on the interrupt mask register. The monotonicity restriction from above is retained.
- **Enriched programs.** These are programs in the enriched calculus, without the monotonicity restriction.

We summarize our results in Table 1. We have determined the complexity of stack boundedness and exact maximum stack size both for monotonic programs and for monotonic programs enriched with tests. For general programs enriched with tests, we have a PSPACE lower bound and an EXPTIME upper bound for both problems; tightening this gap remains an open problem. While the complexities are high, our algorithms are *polynomial* (linear or cubic) in the size of the program, and exponential only in the number of interrupts. In other words, our algorithms are polynomial if the number of interrupts is fixed. Since most real systems have a fixed small number of interrupts (for example Motorola Dragonball processor handles 22 interrupt sources), and the size of programs is the limiting factor, we believe the algorithms should be tractable in practice. Experiments are needed to settle this.

We reduce the stack boundedness and exact stack size problems to state space exploration problems over certain graphs constructed from the interrupt-driven program. We then use the structure of the graph to provide algorithms for the two problems. Our first insight is that for monotonic programs, the maximum stack bounds are attained without any intermediate handler return. The polynomial-time algorithm for monotonic programs is reduced to searching for cycles in a graph; the polynomial-space algorithm for determining the exact maximum stack size of monotonic enriched programs is based on finding the longest path in a (possibly exponential) acyclic graph. Finally, we can reduce the stack boundedness problem and exact maximum stack size problem for enriched programs to finding context-free cycles and context-free longest paths in graphs. Our EXPTIME algorithm for enriched programs is based on a novel technique to find the longest context-free path in a DAG. Our lower bounds are obtained by reductions from reachability in a DAG (which is NLOGSPACE-complete), satisfiability of quantified boolean formulas (which is PSPACE-complete), and reachability for polynomial-space Turing Machines (which is PSPACE-complete). We also provide algorithms that determine, given an interrupt-driven program, whether it is monotonic. In the nonenriched case, monotonicity can be checked in polynomial time (NLOGSPACE); in the enriched case, in co-NP. In Section 2, we recall the interrupt calculus of Palsberg and Ma [4]. In Section 3, we consider monotonic programs, in Section 4, we consider monotonic enriched programs, and in Section 5, we consider enriched programs without the monotonicity restriction.

### *1.1 Related Work*

Brylow, Damgaard, and Palsberg [1] do stack size analysis of a suite of microcontroller programs by running a context-free reachability algorithm for model checking. They use, essentially, the same abstraction that our EXPTIME algorithm uses for enriched programs. Our paper gives more algorithmic details and clarifies that the complexity is exponential in the number of handlers.

Palsberg and Ma [4] present a type system and a type checking algorithm for the interrupt calculus that guarantees stack boundedness and certifies that the stack size is within a given bound. Each type contains information about the stack size and serves as documentation of the program. However, this requires extensive annotations from the programmer (especially since the types can be exponential in the number of handlers), and the required type information is absent in legacy programs. Our work can be seen as related to *type inference* for the interrupt calculus. In particular, we check stack properties of programs without annotations. From our algorithms, we should be able to infer the types of [4]. It remains to be seen whether our algorithms can be as successful on legacy programs as the algorithm of Brylow, Damgaard, and Palsberg [1].

Regehr, Reid, and Webb [8] integrated a stack size analysis in the style of [1] with aggressive abstract interpretation of ALU operations and conditional branches, and they showed how global function inlining can significantly decrease the stack space requirements.

Hughes, Pareto, and Sabry [3,7] use *sized types* to reason about liveness, termination, and space boundedness of reactive systems. However, they require types with explicit space information, and do not address interrupt handling.

Wan, Taha, and Hudak [11] present event-driven Functional Reactive Programming (FRP), which is designed such that the time and space behavior of a program are necessarily bounded. However, the event-driven FRP programs are written in continuation-style, and therefore do not need a stack. Hence stack boundedness is not among the resource issues considered by Wan et al.

Context free reachability has been used in interprocedural program analysis [9]. Recently, Reps, Schwoon, and Jha [10] consider context-free reachability on weighted pushdown graphs to check security properties.

Hillebrand, Kanellakis, Mairson, Vardi [2] studied a boundedness problem that is related to ours, namely whether the depth of recursion of a given Datalog program is *independent* of the input. They showed that several variations of the problem are undecidable. Our boundedness problem focuses on whether the "depth of recursion" (that is, stack size) is finite under *all* possible inputs. The two boundedness problems are different. The problem studied by Hillebrand et al. is to decide whether the depth of recursion is always the same. Our problem is about whether the "depth of recursion" is bounded across all inputs. Note that our problem will allow the "depth of recursion" to vary with the input, as long as there is a common bound that works for all inputs. A further difference is that the *input* to a Datalog program is a finite database, while the "input" to an interrupt-calculus program is an infinite stream of interrupts.

## 2 The Interrupt Calculus

### *2.1 Syntax*

We recall the (abstract) syntax of the interrupt calculus of [4]. We use $x$ to range over a set of program variables, we use *imr* to range over bit strings, and we use $c$ to range over integer constants.

$$\begin{array}{lll}
\text{(program)} & p & ::= (m, \bar{h}) \\
\text{(main)} & m & ::= \mathsf{loop}\ s \mid s\ ;\ m \\
\text{(handler)} & h & ::= \mathsf{iret} \mid s\ ;\ h \\
\text{(statement)} & s & ::= x = e \mid \mathsf{imr} = \mathsf{imr} \wedge imr \mid \mathsf{imr} = \mathsf{imr} \vee imr \mid \\
 & & \quad \mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2 \mid s_1\ ;\ s_2 \mid \mathsf{skip} \\
\text{(expression)} & e & ::= c \mid x \mid x + c \mid x_1 + x_2
\end{array}$$

The pair $p = (m, \bar{h})$ is an *interrupt program* with main program $m$ and interrupt handlers $\bar{h}$. The over-bar notation $\bar{h}$ denotes a sequence $h_1 \ldots h_n$ of handlers. We use the notation $\bar{h}(i) = h_i$. We use $a$ to range over $m$ and $h$.

### *2.2 Semantics*

We use $R$ to denote a *store*, that is, a partial function mapping program variables to integers. We use $\sigma$ to denote a *stack* generated by the grammar $\sigma ::= \mathsf{nil} \mid a :: \sigma$. We define the size of a stack as $|\mathsf{nil}| = 0$ and $|a :: \sigma| = 1 + |\sigma|$.

We represent the imr as a bit sequence $imr = b_0 b_1 \ldots b_n$, where $b_i \in \{0, 1\}$. The 0th bit $b_0$ is the master bit, and for $i > 0$, the $i$th bit $b_i$ is the bit for interrupts from source $i$, which are handled by handler $i$. Notice that the master bit is the most significant bit, the bit for handler 1 is the second-most significant bit, and so on. This layout is different from some processors, but it simplifies the notation used later. For example, the imr value `101b` means that the master bit is set, the bit for handler 1 is not set, and the bit for handler 2 is set. We use the notation $imr(i)$ for bit $b_i$. The predicate *enabled* is defined as

$$\mathit{enabled}(imr, i) = (imr(0) = 1) \wedge (imr(i) = 1), \qquad i \in 1..n.$$

We use 0 to denote the imr value where all bits are 0. We use $\mathrm{t}_i$ to denote the imr value where all bits are 0's except that the $i$th bit is set to 1. We will use $\wedge$ to denote bitwise logical conjunction, $\vee$ to denote bitwise logical disjunction, $\leq$ to denote bitwise logical implication, and $\neg$ to denote bitwise logical negation. Notice that $\mathit{enabled}(\mathrm{t}_0 \vee \mathrm{t}_i, j)$ is true if $i = j$, and false otherwise. The imr values, ordered by $\leq$, form a lattice with bottom element 0.

A *program state* is a tuple $\langle \bar{h}, R, imr, \sigma, a \rangle$ consisting of interrupt handlers $\bar{h}$, a store $R$, an interrupt mask register $imr$, a stack $\sigma$ of return addresses, and a program counter $a$. We refer to $a$ as *the current statement*; it models the instruction pointer of a CPU. The interrupt handlers $\bar{h}$ do not change during computation; they are part of the state to ensure that all names are defined locally in the semantics below. We use $P$ to range over program states. If $P = \langle \bar{h}, R, imr, \sigma, a \rangle$, then we use the notation $P.stk = \sigma$. For $p = (m, \bar{h})$, the initial program state for executing $p$ is $P_p = \langle \bar{h}, \lambda x.0, 0, \mathsf{nil}, m \rangle$, where the function $\lambda x.0$ is defined on the variables that are used in the program $p$.

A small-step operational semantics for the language is given by the reflexive, transitive closure of the relation → on program states:

$$
\begin{aligned}
\langle \bar{h}, R, imr, \sigma, a \rangle &\rightarrow \langle \bar{h}, R, imr \wedge \neg \mathrm{t}_0, a :: \sigma, \bar{h}(i) \rangle && (1)\\
&\quad \text{if } enabled(imr, i) && \\
\langle \bar{h}, R, imr, a :: \sigma', \mathsf{iret} \rangle &\rightarrow \langle \bar{h}, R, imr \vee \mathrm{t}_0, \sigma', a \rangle && (2)\\
\langle \bar{h}, R, imr, \sigma, \mathsf{loop}\ s \rangle &\rightarrow \langle \bar{h}, R, imr, \sigma, s; \mathsf{loop}\ s \rangle && (3)\\
\langle \bar{h}, R, imr, \sigma, x = e; a \rangle &\rightarrow \langle \bar{h}, R\{x \mapsto eval_R(e)\}, imr, \sigma, a \rangle && (4)\\
\langle \bar{h}, R, imr, \sigma, \mathsf{imr} = \mathsf{imr} \wedge imr'; a \rangle &\rightarrow \langle \bar{h}, R, imr \wedge imr', \sigma, a \rangle && (5)\\
\langle \bar{h}, R, imr, \sigma, \mathsf{imr} = \mathsf{imr} \vee imr'; a \rangle &\rightarrow \langle \bar{h}, R, imr \vee imr', \sigma, a \rangle && (6)\\
\langle \bar{h}, R, imr, \sigma, (\mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2); a \rangle &\rightarrow \langle \bar{h}, R, imr, \sigma, s_1; a \rangle \text{ if } R(x) = 0 && (7)\\
\langle \bar{h}, R, imr, \sigma, (\mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2); a \rangle &\rightarrow \langle \bar{h}, R, imr, \sigma, s_2; a \rangle \text{ if } R(x) \neq 0 && (8)\\
\langle \bar{h}, R, imr, \sigma, \mathsf{skip}; a \rangle &\rightarrow \langle \bar{h}, R, imr, \sigma, a \rangle && (9)
\end{aligned}
$$

where the function $eval_R(e)$ is defined as:

$$
\begin{aligned}
eval_R(c) &= c & eval_R(x + c) &= R(x) + c\\
eval_R(x) &= R(x) & eval_R(x_1 + x_2) &= R(x_1) + R(x_2).
\end{aligned}
$$

Rule (1) models that if an interrupt is enabled, then it may occur. The rule says that if $enabled(imr, i)$, then it is a possible transition to push the current statement on the stack, make $\bar{h}(i)$ the current statement, and turn off the master bit in the imr. Notice that we make no assumptions about the interrupt arrivals; any enabled interrupt can occur at any time, and conversely, no interrupt has to occur. Rule (2) models interrupt return. The rule says that to return from an interrupt, remove the top element of the stack, make the removed top element the current statement, and turn on the master bit. Rule (3) is an unfolding rule for loops. It implies that interrupt calculus programs do not terminate, a feature common in reactive systems. Rules (4)–(9) are standard rules for statements. Let $\rightarrow^*$ denote the reflexive transitive closure of $\rightarrow$.

A *program execution* is a sequence $P_p \rightarrow P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_k$ of program states. Consider a program execution $\gamma$ of the form $P_p \rightarrow^* P_i \rightarrow P_{i+1} \rightarrow^* P_j \rightarrow P_{j+1}$ with $P_i = \langle \bar{h}, R, imr_b, \sigma, a \rangle$ and $P_j = \langle \bar{h}, R', imr', \sigma', a' \rangle$. The handler $\bar{h}(i)$ is *called in $\gamma$ with $imr_b$* from state $P_i$ and *returns with $imr_r$* from state $P_j$ if

$$
\begin{aligned}
&P_i \rightarrow P_{i+1} = \langle \bar{h}, R, imr_b \wedge \neg \mathrm{t}_0, a :: \sigma, \bar{h}(i) \rangle \quad \text{and } enabled(imr_b, i),\\
&P_j \rightarrow P_{j+1} = \langle \bar{h}, R', imr_r, \sigma, a \rangle \quad \text{and } \sigma' = a :: \sigma,
\end{aligned}
$$

```
imr = imr or 111b
loop { imr = imr or 111b }
handler 1 {
   imr = imr and 101b
   imr = imr or  100b
   iret
}
```

```
handler 2 {
   imr = imr and 110b
   imr = imr or  010b
   imr = imr or  100b
   imr = imr and 101b
   iret
}
```

Fig. 1. A program in the interrupt calculus

and $P_k.stk \neq \sigma$ for all $i < k \leq j$. We say that there is no handler call in $\gamma$ between $P_i$ and $P_j$ if for all $i \leq k < j$, the transition $P_k \rightarrow P_{k+1}$ is not a transition of the form (1). Similarly, given an execution $P_p \rightarrow^* P_i \rightarrow^* P_j$, there is no handler return between $P_i$ and $P_j$ if for all $i \leq k < j$, the transition $P_k \rightarrow P_{k+1}$ is not a transition of the form (2).

### *2.3 Stack Size Analysis*

We consider the following problems of stack size analysis.

- **Stack boundedness problem** Given an interrupt program $p$, the stack boundedness problem returns “yes” if there exists a finite integer $K$ such that for all program states $P'$, if $P_p \rightarrow^* P'$, then $|P'.stk| \leq K$; and returns “no” if there is no such $K$.

- **Exact maximum stack size problem** For a program state $P$ we define $maxStackSize(P)$ as the least $K \geq 0$ such that for all $P'$, if $P \rightarrow^* P'$, then $|P'.stk| \leq K$; and “infinite” in case no such $K$ exists. The exact maximum stack size problem is given an interrupt program $p$ and returns $maxStackSize(P_p)$.

Figure 1 shows an example of a program in the real interrupt calculus syntax, where “$\wedge$” and “$\vee$” are represented by “and” and ”or” respectively. The bit sequences such as 111b are imr constants. Notice that each of the two handlers can be called from different program points with different imr values. The bodies of the two handlers manipulate the imr, and both are at some point during the execution open to the possibility of being interrupted by the other handler. However, the maximum stack size is 3. This stack size happens if handler 1 is first called with 111b, then handler 2 with 101b, and then handler 1 again with 110b, at which time there are three return addresses on the stack.

We shall analyze interrupt programs under the usual program analysis assumption that all paths in the program are executable. More precisely, our analysis assumes that each data assignment statement $x = e$ in the program has been replaced by skip, and each conditional if0 $(x)$ $s_1$ else $s_2$ has been replaced by $\mathsf{if0}\ (*)\ s_1\ \mathsf{else}\ s_2$, where $*$ denotes nondeterministic choice. While this is an overapproximation of the actual set of executable paths, we avoid trivial undecidability results for deciding if a program path is actually executable. In the following, we assume that the relation $\rightarrow$ is defined on this abstract program.

## 3 Monotonic Interrupt Programs

We first define monotonic interrupt programs and then analyze the stack boundedness and exact maximum stack size problems for such programs. A handler $h_i$ of program $p$ is *monotonic* if for every execution $\gamma$ of $p$, if $h_i$ is called in $\gamma$ with an imr value $imr_b$ and returns with an imr value $imr_r$, then $imr_r \leq imr_b$. The program $p$ is *monotonic* if all handlers $h_1 \ldots h_n$ of $p$ are monotonic. The handler $h_i$ of $p$ is *monotonic in isolation* if for every execution $\gamma$ of $p$, if $h_i$ is called in $\gamma$ with an imr value $imr_b$ from a state $P_i$ and returns with an imr value $imr_r$ from a state $P_j$ such that there is no handler call between $P_i$ and $P_j$, then $imr_r \leq imr_b$.

We first show that a program $p = (m, \bar{h})$ is monotonic iff every handler $h_i \in \bar{h}$ is monotonic in isolation. Moreover, a handler $h_i$ is monotonic in isolation iff, whenever $h_i$ is called with imr value $\mathrm{t}_0 \vee \mathrm{t}_i$ from state $P_i$ and returns with $imr_r$ from state $P_j$, with no handler calls between $P_i$ and $P_j$, then $imr_r \leq \mathrm{t}_0 \vee \mathrm{t}_i$. These observations can be used to efficiently check if an interrupt program is monotonic: for each handler, we check that the return value $imr_r$ of the imr when called with $\mathrm{t}_0 \vee \mathrm{t}_i$ satisfies $imr_r \leq \mathrm{t}_0 \vee \mathrm{t}_i$.

**Lemma 1** *A program $p = (m, \bar{h})$ is monotonic iff every handler $h_i \in \bar{h}$ is monotonic in isolation.*

*Proof.* If there is a handler $h$ which violates monotonicity in isolation then $h$ is not monotonic and hence the program $p$ is not monotonic. For the converse, suppose that all handlers are monotonic in isolation, but the program $p$ is not monotonic. Consider an execution sequence $\gamma$ which violates the monotonicy condition. In $\gamma$, we can choose a handler $h$ which is called with an imr value $imr_b$ and returns with an imr value $imr_r$ such that

$$imr_b \not\geq imr_r \tag{10}$$

but any handler $h'$ which was called from within $h$ with an imr value $imr_{b_{h'}}$ returned with an imr value $imr_{r_{h'}}$ satisfying $imr_{r_{h'}} \leq imr_{b_{h'}}$. From $\gamma$ we now construct a simpler execution sequence $\gamma'$ which also violates the monotonicity condition. We construct $\gamma'$ by omitting from $\gamma$ all calls from within $h$. In $\gamma'$ there are no calls between the call to $h$ and the return of $h$. Each of the omitted calls are monotonic, so in $\gamma'$ $h$ will return with an imr value $imr'_r$ such that

$$imr_r \leq imr'_r \tag{11}$$

Since in this sequence no handler is called from $h$ and $h$ is monotonic in isolation it follows that:

$$imr'_r \leq imr_b \tag{12}$$

From (10), (11), and (12), we have a contradiction. Hence $p$ is monotonic. ∎

**Lemma 2** *Given a program $p = (m, \bar{h})$, a handler $h_i \in \bar{h}$ is monotonic in isolation iff when $h_i$ is called with imr value $t_0 \vee t_i$ from program state $P_i$, and returns with imr value $imr_r$ from program state $P_j$, with no handler calls between $P_i$ and $P_j$, then $imr_r \leq t_0 \vee t_i$.*

*Proof.* If the right-hand side of the "iff" is not satisfied, then $h_i$ is not monotonic in isolation. Conversely, suppose the right-hand side of the "iff" is satisfied but $h_i$ is not monotonic in isolation. Suppose further that $h_i$ is called with the imr value $imr_b$ and it follows some sequence of execution in the handler $h_i$ to return $imr'_r$, with $imr_b \not\geq imr'_r$. Hence there is a bit $j$ such that the $j$-th bit is on in $imr'_r$ but the $j$-th bit is off in $imr_b$. Since the conditionals do not depend on $imr$, the same sequence of execution can be followed when the handler is called with $t_0 \vee t_i$. In this case, the return value $imr_r$ will have the $j$-th bit on, and hence $t_0 \vee t_i \not\geq imr_r$. This is a contradiction. ∎

**Proposition 3** *It can be checked in linear time (NLOGSPACE) if an interrupt program is monotonic.*

*Proof.* It follows from Lemma 1 that checking monotonicity of a program $p$ can be achieved by checking monotonicity of the handlers in isolation. It follows from Lemma 2 that checking monotonicity in isolation for a handler $h_i$ can be achieved by checking if $h_i$ is monotonic when called with $t_i$. Thus checking monotonicity is just checking the return value of the imr when called with $t_i$. This can be achieved in polynomial time by a standard bitvector dataflow analysis. Since the conditionals do not test the value of $imr$, we can join the dataflow information (i.e., bits of the $imr$) at merge points. It is clear that finding bit by bit the return value when called with $t_i$ can be achieved in NLOGSPACE. ∎

### *3.1 Stack Boundedness*

We now analyze the complexity of stack boundedness of monotonic programs. Our main insight is that the maximum stack size is achieved without any intermediate handler returns. First observe that if handler $h$ is enabled when the imr is $imr_1$, then it is enabled for all imr $imr_2 \geq imr_1$. We argue the case where the maximum stack size is finite, the same argument can be formalized in case the maximum stack size is infinite. Fix an execution sequence that achieves the maximum stack size. Let $h$ be the last handler that returned in this sequence (if there is no such $h$ then we are done). Let the sequence of statements executed be $s_0, s_1, \ldots s_{i-1}, s_i, \ldots s_j, s_{j+1}, \ldots$ where $s_i$ was the starting statement of $h$ and $s_j$ the $\mathsf{iret}$ statement of $h$. Suppose $h$ was called with $imr_b$ and returned with $imr_r$ such that $imr_r \leq imr_b$. Consider the execution sequence of statements $s_0, s_1, \ldots s_{i-1}, s_{j+1}, \ldots$ with the execution of handler $h$ being omitted. In the first execution sequence the imr value while executing statement $s_{j+1}$ is $imr_r$ and in the second sequence the imr value is $imr_b$. Since $imr_r \leq imr_b$ then repeating the same sequence of statements and same sequence of calls to handlers with $h$ omitted gives the same stack size. Following a similar argument, we can show that all handlers that return intermediately can be omitted without changing the maximum stack size attained.

**Lemma 4** *For a monotonic program $p$, let $P_{\max}$ be a program state such that $P_p \rightarrow^* P_{\max}$ and for any state $P'$, if $P_p \rightarrow^* P'$ then $|P_{\max}.stk| \geq |P'.stk|$. Then there is a program state $P''$ such that $P_p \rightarrow^* P''$, $|P''.stk| = |P_{\max}.stk|$, and there is no handler return between $P_p$ and $P''$.*

We now give a polynomial-time algorithm for the stack boundedness problem for monotonic programs. The algorithm reduces the stack boundedness question to the presence of cycles in the *enabled graph* of a program. Let $h_1 \ldots h_n$ be the $n$ handlers of the program. Given the code of the handlers, we build the enabled graph $G = \langle V, E \rangle$ as follows.

- There is a node for each handler, i.e., $V = \{h_1, h_2, \ldots h_n\}$.
- Let the instructions of $h_i$ be $C_i = i_1, i_2, \ldots i_m$. There is an edge between $(h_i, h_j)$ if any of the following conditions holds.

(1) There is $l, k$ such that $l \leq k$, the instruction at $i_l$ is $\mathsf{imr} = \mathsf{imr} \vee imr$ with $t_0 \leq imr$, the instruction at $i_k$ is $\mathsf{imr} = \mathsf{imr} \vee imr$ with $t_j \leq imr$ and for all statements $i_m$ between $i_l$ and $i_k$, if $i_m$ is $\mathsf{imr} = \mathsf{imr} \wedge imr$ then $t_0 \leq imr$.

(2) There is $l, k$ such that $l \leq k$, the instruction at $i_l$ is $\mathsf{imr} = \mathsf{imr} \vee imr$ with $t_j \leq imr$, the instruction at $i_k$ is $\mathsf{imr} = \mathsf{imr} \vee imr$ with $t_0 \leq imr$ and for all statements $i_m$ between $i_l$ and $i_k$, if $i_m$ is $\mathsf{imr} = \mathsf{imr} \wedge imr$ then $t_j \leq imr$.

(3) We have $i = j$ and there is $l$ such that the instruction at $i_l$ is $\mathsf{imr} = \mathsf{imr} \vee imr$ with $t_0 \leq imr$ and for all statements $i_m$ between $i_1$ and $i_l$, if $i_m$ is $\mathsf{imr} = \mathsf{imr} \wedge imr$ then $t_i \leq imr$. This gives a self-loop $(h_i, h_i)$.

Since we do not model the program variables, we can analyze the code of $h_i$ and detect all outgoing edges $(h_i, h_j)$ in time linear in the length of $h_i$. We only need to check that there is an $\vee$ statement with an imr constant with $j$th bit 1 and then the master bit is turned on with no intermediate disabling of the $j$th bit or vice versa. Hence the enabled graph for program $p$ can be constructed in time $n^2 \times |p|$ (where $|p|$ denotes the length of $p$).

**Lemma 5** *Let $G_p$ be the enabled graph for a monotonic interrupt program p. If $G_p$ has a cycle, then the stack is unbounded, that is, for all positive integers $K$, there is a program state $P'$ such that $P_p \rightarrow^* P'$ and $|P'.stk| > K$.*

*Proof.* Consider a cycle $C = \langle h_{i_1}, h_{i_2}, \ldots, h_{i_k}, h_{i_1} \rangle$ such that for any two consecutive nodes in the cycle there is an edge between them in $G_p$. Consider the following execution sequence. When $h_{i_1}$ is executed, it turns $h_{i_2}$ and the master bit on. Then, an interrupt of type $h_{i_2}$ occurs. When $h_{i_2}$ is executed, it turns on $h_{i_3}$ and the master bit. Then, an interrupt of type $h_{i_3}$ occurs, and so on. Hence $h_{i_1}$ can be called with $h_{i_1}$ on stack and the sequence of calls can be repeated. If there is a self-loop at the node $h_i$, then $h_i$ can occur infinitely many times. This is because handler $h_i$ can turn the master bit on without disabling itself, so an infinite sequence of interrupts of type $h_i$ will make the stack grow unbounded. ∎

Since cycles in the enabled graph can be found in NLOGSPACE, the stack boundedness problem for monotonic programs is in NLOGSPACE. Note that the enabled graph of a program can be generated on the fly in logarithmic space. Hardness for NLOGSPACE follows from the hardness of DAG reachability.

**Lemma 6** *Stack Boundedness for monotonic interrupt programs is NLOGSPACE-hard.*

*Proof.* We reduce reachability in a DAG to the Stack Boundedness checking problem. Given a DAG $G = (V, E)$ where $V = \{1, 2, ..., n\}$ we write a program $p$ with $n$ handlers $h_1, h_2, ..., h_n$ as follows:

- The code of handler $h_i$ disables all handlers and then enables all its successors in the DAG and the master bit.
- the handler $h_n$ disables all the other handlers and enables itself and the master bit and then disables itself.

Hence the enabled graph of the program will be a DAG with only the node $n$ with a self-loop. So the stack size is bounded iff $n$ is not reachable. Hence stack boundedness checking is NLOGSPACE-hard. ∎

**Theorem 7** *Stack boundedness for monotonic interrupt programs can be checked in time linear in the size of the program and quadratic in the number of handlers. The complexity of stack boundedness for monotonic interrupt programs is NLOGSPACE-complete.*

In case the stack is bounded, we can get a simple upper bound on the stack size as follows. Let $G_p$ be the enabled graph for a monotonic interrupt program $p$. If $G_p$ is a DAG, and the node $h_i$ of $G_p$ has order $k$ in topological sorting order, then we can prove by induction that the corresponding handler $h_i$ of $p$

can occur at most $2^{(k-1)}$ times in the stack.

**Lemma 8** *Let $G_p$ be the enabled graph for a monotonic interrupt driven program $p$. If $G_p$ is a DAG, and the node $h_i$ of $G_p$ has order $k$ in topological sorting order, then the corresponding handler $h_i$ of $p$ can occur at most $2^{(k-1)}$ times in the stack.*

*Proof.* We prove this by induction. Let $h_i$ be the node with order 1. It has no predecessors in the enabled graph. No node in the enabled graph has a self-loop, since our assumption is that the enabled graph $G_p$ is a DAG. Hence $h_i$ must turn its bit off before turning the master bit on. Hence when $h_i$ occurs in the stack its bit is turned off. As no other handler turns it on when the master bit is on (since otherwise there would have been an edge to $h_i$) it cannot occur more than once in the stack. This proves the base case.

Consider a node $h$ with order $k$. By hypothesis, all nodes with order $j$ where $j \leq k-1$ can occur at most $2^{(j-1)}$ times in the stack. Now when the node $h$ occurs in the stack its bit is turned off. So before it occurs again in the stack, one of the predecessors of $h$ must occur in the stack. Hence the number of times $h$ can occur in the stack is given by

$$\begin{aligned}
&1+\textstyle\sum \text{ Number of times its predecessors can occur}\\
&\leq 1+\textstyle\sum_{i=1}^{k-1} 2^{(i-1)}\\
&= 2^{(k-1)}.
\end{aligned}$$

∎

We get the following bound as an immediate corollary of Lemma 8. Let $p = (m, \bar{h})$ be a monotonic interrupt driven program with $n$ handlers, and with enabled graph $G_p$. If $G_p$ is a DAG, then for any program state $P'$ such that $P_p \to^* P'$, we have $|P'.stk| \leq 2^n - 1$. This is because the maximum length of the stack is given by the sum of the number of times a individual handler can be in the stack. By Lemma 8 we know a node with order $j$ can occur at most $2^{j-1}$ times. Hence the maximum length of the stack is given by

$$\sum_{i=1}^{n} 2^{(i-1)} = 2^n - 1$$

In fact, this bound is tight: there is a program with $n$ handlers that achieves a maximum stack size of $2^n - 1$. We show that starting with an imr value of all 1's one can achieve the maximum stack length of $2^n - 1$ while keeping the stack bounded. We give an inductive strategy to achieve this. With one handler which does not turn itself on we can have a stack length 1 starting with imr value 11. By induction hypothesis, using $n-1$ handlers starting with

**Algorithm 1 Function MaxStackLengthBound**

```
Input: An interrupt program p
Output: If the stack size is unbounded then $\infty$,
    else an upper bound on the maximum stack size
1. Build the Enabled Graph $G$ from the Program p
2. If $G$ has a cycle then the Maximum Stack Length is $\infty$.
3. If $G$ is a DAG then topologically sort and order nodes of $G$
4.1 For $i \leftarrow$ 1to $|V[G]|$
    4.2 For a node $h$ with order $i$
        $N[h]= 1 + \sum N[h_j]$ where $h_j$ is a predecessor of $h$
5. Upper Bound on Maximum Length of Stack=$\sum N[h]$ for all handlers $h$
```

imr value all 1's we can achieve a stack length of $2^{n-1}-1$. Now we add the $n$th handler and modify the previous $n-1$ handlers such that they do not change the bit for the $n$th handler. The $n$-th handler turns on every bit except itself, and then turns on the master bit. The following sequence achieves a stack size of $2^n - 1$. First, the first $n-1$ handlers achieve a stack size of $2^{n-1} - 1$ using the inductive strategy. After this, the $n$th handler is called. It enables the $n-1$ handlers but disables itself. Hence the sequence of stack of $2^{n-1} - 1$ can be repeated twice and the $n$ the handler can occur once in the stack in between. The total length of stack is thus $1 + (2^{n-1} - 1) + (2^{n-1} - 1) = 2^n - 1$. Since none of the other handlers can turn the $n$th handler on, the stack size is in fact bounded.

We now give a polynomial time procedure to give an upper bound on the stack size if it is bounded. If the stack can possibly grow unbounded we report infinite. If the stack is bounded we compute an upper bound $N[h]$ on the number of times a handler $h$ can occur in the stack. The algorithm MaxStackLengthBound is shown in Algorithm 1.

**Lemma 9** *Function MaxStackLengthBound correctly checks the stack boundedness of interrupt driven programs, that is, if MaxStackLengthBound returns $\infty$ then there is some execution of the program that causes the stack to be unbounded. It also gives an upper bound on the number of times a handler can occur in the stack, that is, if MaxStackBound(p) is $N$, then the maximum stack size on any execution sequence of p is bounded above by $N$.*

*Proof.* It follows from Lemma 5 that if the enabled graph has a cycle then the stack can grow unbounded. This is achieved by Step 2 of MaxStackLengthBound. It follows from Lemma 8 that the maximum number of times a handler can occur in the stack is one plus the sum of the number of times its predecessors can occur. This is achieved in Step 4 of MaxStackLengthBound. ∎

**Lemma 10** *Function MaxStackLengthBound runs in time polynomial in size* *of the interrupt driven program.*

*Proof.* The enabled graph can be built in time $h^2 \times PC$ where $h$ is the number of handlers and $PC$ is the number of program statements. Steps 2,3 and 4 can be achieved in time linear in the size of the enabled graph, and hence in time linear in the size of the program. ∎

While MaxStackLengthBound is simple, one can construct simple examples to show that it may exponentially overestimate the upper bound on the stack size. We show next that this is no accident: we now prove that the exact maximum stack size problem problem is PSPACE-hard. There is a matching upper bound: the exact maximum stack size problem can be solved in PSPACE. We defer this algorithm to Section 4, where we solve the problem for a more general class of programs.

### *3.2 Maximum Stack Size*

We now prove that the exact maximum stack size problem is PSPACE-hard. We start with a little warm-up: first we show that the problem is both NP-hard and co-NP hard. We show this by showing that the problem is DP-hard, where DP is the class of all languages $L$ such that $L = L_1 \cap L_2$ for some language $L_1$ in NP and some language $L_2$ in co-NP [6] (note that DP is not the class NP ∩ co-NP [5,6]). We reduce the problem of EXACT-MAX Independent Set of a Graph and its complement to the problem of finding the exact maximum size of the stack of programs in interrupt calculus. The EXACT-MAX IND problem is the following:

EXACT-MAX IND $= \{\langle G, k\rangle$ the size of the maximum independent set is $k\}$

EXACT-MAX IND is DP-complete [5]. Given an undirected graph $G = \langle V, E\rangle$ where $V = \{1, 2, ..., n\}$, we construct an interrupt-driven program as follows. We create a handler $h_i$ for every node $i$. Let $N_i = \{j : (i, j) \in E\}$ be the neighbors of node $i$ in $G$. The code of $h_i$ disables itself and all the handlers of $N_i$ and then turns the master bit on. The main program enables all handlers and then enters an empty loop. Consider the maximum stack size and the handlers in it. First observe that once a handler is disabled, it is never re-enabled. Hence, no handler can occur twice in the stack, as every handler disables itself and no other handler turns it on. Let $h_i$ and $h_j$ be two handlers in the stack such that $h_i$ occurs before $h_j$. Then $(i, j) \notin E$, since if $(i, j) \in E$ then $h_i$ would have turned $h_j$ off, and thus $h_j$ could not have occurred in the stack (since $h_j$ is never re-enabled). Hence if we take all the handlers that occur in the stack the corresponding nodes in the graph form an independent set. Consider an independent set $I$ in $G$. All the handlers corresponding to the nodes in $I$ can occur in the stack as none of these handlers is disabled by any other. We have thus proved given a stack with handlers we can construct an independent set of size equal to the size of the stack. Conversely, given an independent set we can construct a stack size equal to the size of the independent set. Hence the EXACT-MAX IND problem can be reduced to the problem of finding the exact maximum size of the stack. Hence the problem of finding exact stack size in DP-hard. It follows that it is NP-hard and co-NP hard.

We now give the proof of PSPACE-hardness, which is considerably more technical. We define a subclass of monotonic interrupt calculus which we call simple interrupt calculus and show the exact maximum stack size problem is already PSPACE-hard for this class. It follows that exact maximum stack size is PSPACE-hard for monotonic interrupt-driven programs.

For $imr', imr''$ where $imr'(0) = 0$ and $imr''(0) = 0$, define $\mathcal{H}(imr';\ imr'')$ to be the interrupt handler

```
$\mathsf{imr} = \mathsf{imr} \wedge \neg imr'$;
$\mathsf{imr} = \mathsf{imr} \vee (t_0 \vee imr'')$;
$\mathsf{imr} = \mathsf{imr} \wedge \neg(t_0 \vee imr'')$;
iret.
```

A *simple interrupt calculus program* is an interrupt calculus program where the main program is of the form

```
$\mathsf{imr} = \mathsf{imr} \vee (imr_S \vee t_0)$;
loop skip
```

where $imr_S(0) = 0$ and every interrupt handler is of the form $\mathcal{H}(imr';\ imr'')$. Intuitively, a handler of a simple interrupt calculus program first disables some handlers, then enables other handlers and enables interrupt handling. This opens the door to the handler being interrupted by other handlers. After that, it disables interrupt handling, and makes sure that the handlers that are enabled on exit are a subset of those that were enabled on entry to the handler.

For a handler $h_i$ of the form $\mathcal{H}(imr';\ imr'')$, we define function $f_i(imr) = imr \wedge (\neg imr') \vee imr''$. Given a simple interrupt calculus program $p$, we define a directed graph $G(p) = (V, E)$ such that

- $V = \{\ imr \ \mid\ imr(0) = 0\ \}$,
- $E = \{\ (imr, f_i(imr), i)\ \mid\ \mathsf{t_i} \leq imr\}$ is a set of labeled edges from $imr$ to $f_i(imr)$ with label $i \in \{1..n\}$.

The edge $(imr, f_i(imr), i)$ in $G(p)$ represents the call to the interrupt handler

$h(i)$ when imr value is $imr$. We define $imr_S$ as the start node of $G(p)$ and we define $\mathcal{M}(imr)$ as the longest path in $G(p)$ from node $imr$. The notation $\mathcal{M}(imr)$ is ambiguous because it leaves the graph unspecified; however, in all cases below, the graph in question can be inferred from the context.

**Lemma 11** *For a simple interrupt calculus program p, we have that*

$$maxStackSize(P_p) = |\mathcal{M}(imr_S)|.$$

*Proof.* By definition, the state of a simple interrupt program $p = (m, \bar{h})$ is of the form $\langle \bar{h}, 0, imr, \sigma, a\rangle$ and stack size of $p$ increases whenever an interrupt is handled and we have state transition of the form

$$\langle \bar{h}, 0, imr, \sigma, a\rangle \rightarrow \langle \bar{h}, 0, imr \wedge \neg t_0, a :: \sigma, \bar{h}(i)\rangle \text{ if } imr \geq t_i \vee t_0.$$

Let $a_i,\ i \in \{1..4\}$ represent the four statements in the body of an interrupt handler such that any handler is of the form $a_1; a_2; a_3; a_4$. By definition of simple interrupt program., the master bit is enabled only between $a_2$ and $a_3$, where calls to other handlers may occur. Also, after a call to a interrupt handler returns, the imr value is always less than or equal to the imr value before the call. Thus, during a call to handler $h_i$ with initial imr value equal to $imr$, the only possible states where interrupts maybe be handled are of the form $\langle \bar{h}, 0, imr', \sigma, a_3; a_4\rangle$, where $imr' \leq f_i(imr)$. Then, we only need to examine state transitions of the following form to compute $maxStackSize(P_p)$:

$$\langle \bar{h}, 0, imr, \sigma, a\rangle \rightarrow^* \langle \bar{h}, 0, imr', a :: \sigma, a_3; a_4\rangle,$$

where $imr' \leq f_i(imr)\ \forall i,$ such that $t_i \vee t_0 \leq imr$.

Let $P = \langle \bar{h}, 0, imr, \sigma, a\rangle$ and $P' = \langle \bar{h}, 0, imr', \sigma, a\rangle$. By an easy induction on execution sequences, we have that $maxStackSize(P) \leq maxStackSize(P')$ if $imr \leq imr'$. Therefore, it is sufficient to consider state transitions of the form

$$\langle \bar{h}, 0, imr, \sigma, a\rangle \rightarrow^* \langle \bar{h}, 0, f_i(imr), a :: \sigma, a_3; a_4\rangle,$$

where $imr \geq t_i \vee t_0$. In the main loop, the possible states where interrupts may be handled are of the form

$$\langle \bar{h}, 0, imr_S \vee t_0, \mathsf{nil}, \mathsf{loop\ skip}\rangle \text{ and } \langle \bar{h}, 0, imr_S \vee t_0, \mathsf{nil}, \mathsf{skip; loop\ skip}\rangle.$$

Let $a_0$ be the statements of the form loop skip or skip;loop skip. To compute $maxStackSize(P_p)$, we only need to consider transitions of the form

$$\langle \bar{h}, 0, imr_S \vee t_0, \sigma, a_0\rangle \rightarrow^* \langle \bar{h}, 0, f_i(imr_S) \vee t_0, a_0 :: \sigma, a_3; a_4\rangle,$$

where $imr_S \geq t_i$, and

$$\langle \bar{h}, 0, imr, \sigma, a_3; a_4 \rangle \rightarrow^* \langle \bar{h}, 0, f_j(imr), a_3; a_4 :: \sigma, a_3'; a_4' \rangle,$$

where $imr \geq t_j \vee t_0$.

It is now clear that we can just use $imr \wedge \neg t_0$ to represent states that we are interested in with starting states represented by $imr_S$. The above two kinds of transitions can be uniquely represented by edges of the form $(imr_S, f_i(imr_S), i), (imr \wedge \neg t_0, f_j(imr \wedge \neg t_0), j)$ in graph $G(p)$. Therefore, $\mathit{maxStackSize}(P_p)$ is equal to the length of the longest path in $G(p)$ from the start node $imr_S$. ∎

**Lemma 12** *For a simple interrupt calculus program p, and a subgraph of $G(p)$, we have that if $imr \leq imr_1 \vee imr_2$, then $|\mathcal{M}(imr)| \leq |\mathcal{M}(imr_1)| + |\mathcal{M}(imr_2)|$.*

*Proof.* The lemma follows from the following claim. If $imr \leq imr_1 \vee imr_2$, and $P$ is a path from node $imr$ to $imr'$, then we can find a path $P_1$ from $imr_1$ to $imr_1'$ and a path $P_2$ from node $imr_2$ to $imr_2'$ such that $|P| = |P_1| + |P_2|$ and $imr' \leq imr_1' \vee imr_2'$.

Given this claim, the lemma following from the following reasoning. We can apply the above claim to the situation with $\mathcal{M}(imr)$ as the path $P$ from $imr$ to 0. Since $|\mathcal{M}(imr_1)| \geq |P_1|$ and $|\mathcal{M}(imr_2)| \geq |P_2|$, we have $|\mathcal{M}(imr)| \leq |\mathcal{M}(imr_1)| + |\mathcal{M}(imr_2)|$.

We now prove the claim. We proceed by induction on the length of $P$. The base case of $|P| = 0$ is trivially true. Suppose the claim is true for $|P| = k$ and that $P'$ is $P$ appended with an edge to $imr''$. We need to prove the case of $P'$. Since $P$ ends at $imr'$, there exists $t_i \leq imr'$ such that $imr'' = f_i(imr')$. By the induction hypothesis, $t_i \leq imr' \leq imr_1' \vee imr_2'$. Thus, there exists $a \in \{1, 2\}$ such that $t_i \leq imr_a'$. Suppose that $t_i \leq imr_1'$ (the case of $t_i \leq imr_2'$ is similar and is omitted). We can let $P_1'$ be $P_1$ appended with an edge to $imr_1''$ where $imr_1'' = f_i(imr_1')$. By definition of $f_i$, we have $f_i(imr') \leq f_i(imr_1') \vee imr_2'$. Thus, we have $|P'| = |P| + 1 = |P_1| + 1 + |P_2| = |P_1'| + |P_2|$ and $imr'' \leq imr_1'' \vee imr_2'$. ∎

We now show PSPACE-hardness for simple interrupt calculus. Our proof is based on a polynomial-time reduction from the *quantified boolean satisfiability* (QSAT) problem [5].

We first illustrate our reduction by a small example. Suppose we are given a QSAT instance $\mathsf{S} = \exists \mathsf{x}_2 \forall \mathsf{x}_1 \ \phi$ with

$$\phi = (\mathsf{l}_{11} \vee \mathsf{l}_{12}) \wedge (\mathsf{l}_{21} \vee \mathsf{l}_{22}) = (\mathsf{x}_2 \vee \neg \mathsf{x}_1) \wedge (\mathsf{x}_2 \vee \mathsf{x}_1).$$

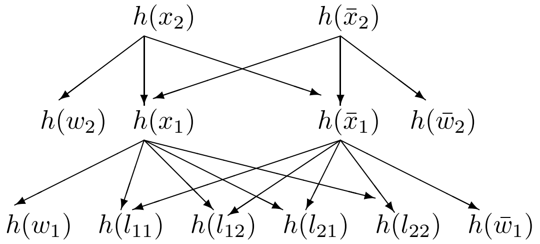

Fig. 2. Enable relation of interrupt handlers

We construct a simple interrupt program $p = (m, \bar{h})$ with an imr register, where $\bar{h} = \{h(x_i), h(\bar{x}_i), h(w_i), h(\bar{w}_i), h(l_{ij}) \mid i, j = 1, 2\}$ are 12 handlers. The imr contains 13 bits: a master bit, and each remaining bit 1-1 maps to each handler in $\bar{h}$. Let $\mathcal{D} = \{x_i, \bar{x}_i, w_i, \bar{w}_i, l_{ij} \mid i, j = 1, 2\}$. We use $t_x$, where $x \in \mathcal{D}$, to denote the imr value where all bits are 0's except the bit corresponding to handler $h(x)$ is set to 1. The initial imr value $imr_S$ is set to $imr_S = t_{x_2} \vee t_{\bar{x}_2}$.

We now construct $\bar{h}$. Let $E(h(x))$, $x \in \mathcal{D}$, be the set of handlers that $h(x)$ enables. This *enable* relation between the handlers of our example is illustrated in Figure 2, where there is an edge from $h(x_i)$ to $h(x_j)$ iff $h(x_i)$ enables $h(x_j)$. Let $D(h(x))$, $x \in \mathcal{D}$, be the set of handlers that $h(x)$ disables. Let $L = \{h(l_{ij}) \mid i, j = 1, 2\}$. The $D(h(x))$,$x \in \mathcal{D}$, are defined as follows:

$$D(h(x_2)) = D(h(\bar{x}_2)) = \{h(x_2), h(\bar{x}_2)\} \tag{13}$$
$$D(h(x_1)) = \{h(x_1)\} \quad D(h(\bar{x}_1)) = \{h(\bar{x}_1)\} \tag{14}$$
$$D(h(w_2)) = D(h(\bar{w}_2)) = \{h(x_1), h(\bar{x}_1)\} \cup \{h(w_i), h(\bar{w}_i) \mid i = 1, 2\} \cup L \tag{15}$$
$$D(h(w_1)) = D(h(\bar{w}_1)) = \{h(w_1), h(\bar{w}_1)\} \cup L \tag{16}$$
$$D(h(l_{ij})) = \{h(l_{i1}), h(l_{i2})\} \cup \{h(w_k) \mid l_{ij} = \neg x_k\} \cup \{h(\bar{w}_k) \mid l_{ij} = x_k\}. \tag{17}$$

If $h(x) = \mathcal{H}(imr'; imr'')$, then $imr' = \bigvee_{h(y) \in E(h(x))} t_y$ and $imr'' = \bigvee_{h(z) \in D(h(x))} t_z$, where $x, y, z \in \mathcal{D}$. We claim that the QSAT instance $\mathsf{S}$ is satisfiable iff $|\mathcal{M}(imr_S)| = 10$, where $imr_S = t_{x_2} \vee t_{\bar{x}_2}$. We sketch the proof as follows.

Let $imr_L = \bigvee_{h(l) \in L} t_l$, where $l \in \mathcal{D}$. From (17) and Figure 2, it can be shown that $|\mathcal{M}(imr_L)| = 2$. From Figure 2, we have $E(h(x_1)) = \{h(w_1)\} \cup L$; and together with (16), and (17), it can be shown that

$$|\mathcal{M}(t_{x_1})| = 1 + |\mathcal{M}(t_{w_1} \vee imr_L)| \leq 2 + |\mathcal{M}(imr_L)| = 4$$

and the equality holds iff $\exists \mathsf{j}_1, \mathsf{j}_2 \in 1, 2$, such that $\mathsf{l}_{1\mathsf{j}_1}, \mathsf{l}_{2\mathsf{j}_2} \neq \neg \mathsf{x}_1$, because otherwise handler $h(w_1)$ would be surely disabled. Similarly, it can be shown that $|\mathcal{M}(t_{\bar{x}_1})| \leq 4$, and that

$$|\mathcal{M}(t_{x_1} \vee t_{\bar{x}_1})| \leq |\mathcal{M}(t_{x_1})| + |\mathcal{M}(t_{\bar{x}_1})| \leq 8,$$

where the equality holds iff $\exists \mathsf{j}_1, \mathsf{j}_2$, such that $\mathsf{l}_{1\mathsf{j}_1}, \mathsf{l}_{2\mathsf{j}_2} \neq \neg \mathsf{x}_1$ and $\exists \mathsf{j}'_1, \mathsf{j}'_2$, such that

$\mathsf{l}_{1\mathsf{j}_1'}, \mathsf{l}_{2\mathsf{j}_2'} \neq \mathsf{x}_1$. From Figure 2, we have $E(h(x_2)) = \{h(w_2), h(x_1), h(\bar{x}_1)\}$. Thus,

$$|\mathcal{M}(t_{x_2})| = 1 + |\mathcal{M}(t_{w_2} \vee t_{x_1} \vee t_{\bar{x}_1})| \leq 2 + |\mathcal{M}(t_{x_1} \vee t_{\bar{x}_1})| = 10,$$

and it can be shown from (15) and (17), that the equality holds iff $\exists \mathsf{j}_1, \mathsf{j}_2$ such that $\mathsf{l}_{\mathsf{ij}_1}, \mathsf{l}_{\mathsf{ij}_2} \neq \neg\mathsf{x}_2, \neg\mathsf{x}_1$ and $\exists \mathsf{j}_1', \mathsf{j}_2'$ such that $\mathsf{l}_{\mathsf{ij}_1'}, \mathsf{l}_{\mathsf{ij}_2'} \neq \neg\mathsf{x}_2, \mathsf{x}_1$, which implies that both $\mathsf{x}_2 = \mathsf{true}, \mathsf{x}_1 = \mathsf{true}$ and $\mathsf{x}_2 = \mathsf{true}, \mathsf{x}_1 = \mathsf{false}$ are satisfiable truth assignments to $\phi$. Similarly, it can be shown that $|\mathcal{M}(t_{\bar{x}_2})| = 10$ iff both $\mathsf{x}_2 = \mathsf{false}, \mathsf{x}_1 = \mathsf{true}$ and $\mathsf{x}_2 = \mathsf{false}, \mathsf{x}_1 = \mathsf{false}$ are satisfiable truth assignments to $\phi$.

From (13), we have $|\mathcal{M}(t_{x_2} \vee t_{\bar{x}_2})| = \max(|\mathcal{M}(t_{x_2})|, |\mathcal{M}(t_{\bar{x}_2})|)$. Therefore, $|\mathcal{M}(imr_S)| = 10$ iff there exists $\mathsf{x}_2$ such that for all $\mathsf{x}_1$, $\phi$ is satisfiable, or equivalently iff $\mathsf{S}$ is satisfiable. For our example, $\mathsf{S}$ is satisfiable since $\exists \mathsf{x}_2 = \mathsf{true}$ such that $\forall \mathsf{x}_1$, $\phi$ is satisfiable. Correspondingly, $|\mathcal{M}(imr_S)| = |\mathcal{M}(x_2)| = 10$.

**Theorem 13** *The exact maximum stack size problem for monotonic interrupt programs is PSPACE-hard.*

*Proof.* We will do a reduction from the QSAT problem. Suppose we are given an instance of QSAT problem

$$S = \exists x_n \forall x_{n-1} \ldots \exists x_2 \forall x_1 \quad \phi,$$

where $\phi$ is a 3SAT instance in conjunctive normal form of $n$ variables $x_n, \ldots, x_1$ and $L$ boolean clauses. Let $\phi_{ij}$ be the $j$th literal of the $i$th clause in $\phi$ and $\phi = \bigwedge_{i=1}^{L} \bigvee_{j=1}^{3} \phi_{ij}$. We construct a program $p = (m, \bar{h})$ and $\bar{h} = \{h(i) \mid i \in \{1 \ldots 3L + 4n\}\ \}$.

As before, we define a graph $G(p) = (V, E)$ such that $V = \{imr \mid imr(0) = 0\}$ and $E = \{(imr, f_i(imr), i) \mid t_i \leq imr\}$, where $f_i(imr) = imr \wedge \neg imr' \vee imr''$ iff $h(i) = \mathcal{H}(imr';\ imr'')$.

For clarity, we define three kinds of indices: $d_{ij} = 3(i-1) + j$, where $i \in \{1..L\}, j \in \{1..3\}$; $q_i^a = 3L + 4i - 3 + a$, and $w_i^a = 3L + 4i - 1 + a$, where $i \in \{1..n\}, a \in \{0, 1\}$.

Let

$$\begin{aligned}
D &= \{d_{i1}, d_{i2}, d_{i3} \mid \forall i \in \{1..L\}\} \\
D_{ij} &= \{d_{i1}, d_{i2}, d_{i3}\} \cup \{w_k^a \mid (a = 1 \wedge \phi_{ij} = x_k) \vee (a = 0 \wedge \phi_{ij} = \neg x_k)\} \\
W_i &= \{w_j^a \mid \forall j \in \{1..i\}, \forall a \in \{0, 1\}\} \\
Q_i &= \{q_j^a \mid \forall j \in \{1..i\}, \forall a \in \{0, 1\}\}.
\end{aligned}$$

We will use the abbreviation

$$imr_0 = \bigvee_{i \in D} t_i, \quad imr_k = t_{q_k^0} \vee t_{q_k^1} \; \forall k \in \{1..n\}.$$

Assume that $n$ is even. For all $a \in \{0,1\}$, let

$$\begin{aligned}
f_{q^a_{2k-1}}(x) &= x \wedge \neg t_{q^a_{2k-1}} \vee (imr_{2k-2} \vee t_{w^a_{2k-1}}), && \forall k \in \{1..n/2\} \\
f_{q^a_{2k}}(x) &= x \wedge \neg imr_{2k} \vee (imr_{2k-1} \vee t_{w^a_{2k}}), && \forall k \in \{1..n/2\} \\
f_{w^a_k}(x) &= x \wedge \neg \bigvee_{i \in D \cup Q_{k-1} \cup W_k} t_i, && \forall k \in \{1..n\} \\
f_{d_{ij}}(x) &= x \wedge \neg \bigvee_{k \in D_{ij}} t_k, && \forall i \in \{1..L\}, j \in \{1,2,3\}.
\end{aligned}$$

Given an imr value $r$, we define the graph $G_r(p)$ to be the subgraph of $G(p)$ such that any edge labeled $d_{ij}$ is removed for all $i, j$ such that $\phi_{ij} = \neg x_k$ and $t_{w^0_k} \leq r$, or $\phi_{ij} = x_k$ and $t_{w^1_k} \leq r$. We use $\mathcal{M}_r(imr)$ to denote the longest path in $G_r(p)$ from $imr$. We organize the proof as a sequence of claims.

**Claim 14** *$\forall k \in \{1..\frac{n}{2}\}$, $|\mathcal{M}_r(imr_{2k})| = \max_{a \in \{0,1\}} |\mathcal{M}_r(t_{q^a_{2k}})|$, and $|\mathcal{M}_r(imr_0)| \leq L$.*

*Proof of Claim 14.* By definition, we have that $\forall a \in \{0,1\}$, $f_{q^a_{2k}}(x) = x \wedge \neg imr_{2k} \vee (imr_{2k-1} \vee t_{w^a_{2k}})$, from which the claim follows.

By definition of $f_{d_{ij}}$, for each $i \in \{1..L\}$, $\mathcal{M}(imr_0)$ can contain at most one edge with label $d_{ij}$, where $j \in \{1,2,3\}$. Thus, $|\mathcal{M}(imr_0)| \leq L$. ∎

**Claim 15** *$|\mathcal{M}_r(imr_{2k-1})| = \sum_{b \in \{0,1\}} |\mathcal{M}_r(t_{q^b_{2k-1}})|$.*

*Proof of Claim 15.* From Lemma 12, we have $|\mathcal{M}_r(imr_{2k-1})| \leq \sum_{b \in \{0,1\}} |\mathcal{M}_r(t_{q^b_{2k-1}})|$.

Let $P$ be the path from $imr_{2k-1}$ to $t_{q^1_{2k-1}}$ constructed from $\mathcal{M}_r(t_{q^0_{2k-1}})$ by replacing any node $imr$ on $\mathcal{M}_r(t_{q^0_{2k-1}})$ with $imr \vee t_{q^1_{2k-1}}$. It is straightforward to show that if edge $(imr, imr', i)$ is on $\mathcal{M}_r(t_{q^0_{2k-1}})$, then $(imr \vee t_{q^1_{2k-1}}, imr' \vee t_{q^1_{2k-1}}, i)$ is on $P$. If we concatenate $P$ with $\mathcal{M}_r(t_{q^1_{2k-1}})$, then we have a path from $imr_{2k-1}$ of length $|\mathcal{M}_r(t_{q^0_{2k-1}})| + |\mathcal{M}_r(t_{q^1_{2k-1}})|$. ∎

**Claim 16** *$|\mathcal{M}(imr_n)| \leq 2^{n/2}(6+L) - 6$.*

*Proof of Claim 16.* It is sufficient to prove that $|\mathcal{M}(imr_{2k})| \leq 2^k(6+L) - 6$. For all $a \in \{0,1\}$ we have:

$$\begin{aligned}
|\mathcal{M}(t_{q^a_{2k}})| &= 1 + |\mathcal{M}(imr_{2k-1} \vee t_{w^a_{2k}})| \\
&\leq 2 + |\mathcal{M}(imr_{2k-1})| \\
&= 2 + \sum_{b \in \{0,1\}} |\mathcal{M}(t_{q^b_{2k-1}})| \\
&= 4 + \sum_{b \in \{0,1\}} |\mathcal{M}(imr_{2k-2} \vee t_{w^b_{2k-1}})| \\
&\leq 6 + 2|\mathcal{M}(imr_{2k-2})|
\end{aligned}$$

$$\begin{aligned}
|\mathcal{M}(imr_{2k})| &= \max_{a \in \{0,1\}} (|\mathcal{M}(t_{q^0_{2k}})|, |\mathcal{M}(t_{q^1_{2k}})|) \\
&\leq 6 + 2|\mathcal{M}(imr_{2k-2})|
\end{aligned}$$

From the last inequality and Claim 14, it is straightforward to show the claim by induction on $k$. ∎

**Claim 17** *For any $r$ and $a \in \{0,1\}$, $|\mathcal{M}_r(t_{q^a_{2k}})| = 2^k(6+L)-6$ iff $\forall b \in \{0,1\}$, $|\mathcal{M}_{r'}(t_{q^b_{2k-1}})| = 2^{k-1}(6+L)-4$, where $r' = r \vee t_{w^a_{2k}}$.*

*Proof of Claim 17.* Suppose that $|\mathcal{M}_r(t_{q^a_{2k}})| = 2^k(6+L)-6$. The path $\mathcal{M}_r(t_{q^a_{2k}})$ must contain the edge with label $w^a_{2k}$ because otherwise

$$\begin{aligned}
|\mathcal{M}_r(t_{q^a_{2k}})| &= 1 + |\mathcal{M}_r(imr_{2k-1} \vee t_{w^a_{2k}})| \\
&= 1 + |\mathcal{M}_r(imr_{2k-1})| \\
&\leq 1 + |\mathcal{M}(imr_{2k-1})| \leq 2^k(6+L) - 7.
\end{aligned}$$

By definition of $f_{q^a_{2k}}$, for any node $imr$ on the path $\mathcal{M}_r(imr_{2k-1} \vee t_{q^a_{2k}})$, we have $f_{w^a_{2k}}(imr) = 0$. Thus, the edge labeled $w^a_{2k}$ can only be the last edge on $\mathcal{M}_r(t_{q^a_{2k}})$. By definition of $f_{d_{ij}}$, the longest path from $imr_{2k-1} \vee t_{w^a_{2k}}$ containing edge labeled $w^a_{2k}$ does not contain any edge labeled $d_{ij}$ for all $i, j$ such that $\phi_{ij} = x_{2k}$ if $a = 1$, and $\phi_{ij} = \neg x_{2k}$ if $a = 0$. This path is the same path in $G_{r'}(p)$, where $r' = r \vee t_{w^a_{2k}}$. Therefore,

$$\begin{aligned}
|\mathcal{M}_r(t_{q^a_{2k}})| = 2^k(6+L) - 6 &= |\mathcal{M}_{r'}(t_{q^a_{2k}})| \\
&= 1 + |\mathcal{M}_{r'}(imr_{2k-1} \vee t_{w^a_{2k}})| \\
&\leq 1 + |\mathcal{M}_{r'}(imr_{2k-1})| + |\mathcal{M}_{r'}(t_{w^a_{2k}})| \\
&= 2 + \sum_{b \in \{0,1\}} |\mathcal{M}_{r'}(t_{q^b_{2k-1}})|, \text{ and} \\
\sum_{b \in \{0,1\}} |\mathcal{M}_{r'}(t_{q^b_{2k-1}})| &\geq 2^k(6+L) - 8.
\end{aligned}$$

Since

$$\begin{aligned}|\mathcal{M}_{r'}(t_{q^b_{2k-1}})| &= 1 + |\mathcal{M}_{r'}(imr_{2k-2} \vee t_{w^b_{2k-1}})| \\ &\leq 2 + |\mathcal{M}_{r'}(imr_{2k-2})| \\ &\leq 2 + |\mathcal{M}(imr_{2k-2})| = 2^{k-1}(6+L) - 4,\end{aligned}$$

we have $\forall b \in \{0,1\}$, $|\mathcal{M}_{r'}(t_{q^b_{2k-1}})| = 2^{k-1}(6+L) - 4$.

Conversely, assume that for all $b \in \{0,1\}$, $|\mathcal{M}_{r'}(t_{q^b_{2k-1}})| = 2^{k-1}(6+L) - 4$ where $r' = r \vee t_{w^a_{2k}}$. From Claim 2, we know that $|\mathcal{M}_{r'}(imr_{2k-1})| = \sum_{b\in\{0,1\}} |\mathcal{M}_{r'}(t_{q^b_{2k-1}})| = 2^k(6+L) - 8$.

Let $P$ be a path from $imr_{2k-1} \vee t_{w^a_{2k}}$ to $t_{w^a_{2k}}$ constructed from $\mathcal{M}_{r'}(imr_{2k-1})$ by replacing any node $imr$ on $\mathcal{M}_{r'}(imr_{2k-1})$ with $imr \vee t_{w^a_{2k}}$. It is straightforward to show that if edge $(imr, imr', i)$ is on $\mathcal{M}_{r'}(imr_{2k-1})$, then $(imr \vee t_{w^a_{2k}}, imr' \vee t_{w^a_{2k}}, i)$ is on $P$ as well.

If we concatenate $P$ with $\mathcal{M}_{r'}(t_{w^a_{2k}})$, then we have a path from $imr_{2k-1} \vee t_{w^a_{2k}}$ in graph $G_{r'}(p)$ of length $2^k(6+L) - 7$. Thus, $|\mathcal{M}_r(t_{q^a_{2k}})| = |\mathcal{M}_{r'}(t_{q^a_{2k}})| = 2^k(6+L) - 6$. ∎

**Claim 18** *For any $r'$ and $b \in \{0,1\}$, we have $|\mathcal{M}_{r'}(t_{q^b_{2k-1}})| = 2^{k-1}(6+L) - 4$ iff $|\mathcal{M}_{r''}(imr_{2k-2})| = 2^{k-1}(6+L) - 6$, where $r'' = r' \vee t_{w^b_{2k-1}}$.*

*Proof of Claim 18.* The proof is similar to the proof of Claim 17, we omit the details. ∎

**Claim 19** *$|\mathcal{M}(imr_n)| = 2^{n/2}(6+L) - 6$ iff $\exists a_n \forall a_{n-1} \ldots \exists a_2 \forall a_1 \in \{0,1\}$, such that for $r = \bigvee_{k\in\{1..n\}} t_{w^{a_k}_k}$, $|\mathcal{M}_r(imr_0)| = L$.*

*Proof of Claim 19.* From Claim 14, we know that $|\mathcal{M}_r(imr_{2k})| = 2^k(6+L) - 6$ iff $\exists a \in \{0,1\}$ such that $|\mathcal{M}_r(t_{q^a_{2k}})| = 2^k(6+L) - 6$. Together with Claim 17 and Claim 18, we have that for $k \in \{1..n/2\}$, $|\mathcal{M}_r(imr_{2k})| = 2^k(6+L) - 6$ iff there exists $a \in \{0,1\}$ such that forall $b \in \{0,1\}$, we have $|\mathcal{M}_{r''}(imr_{2k-2})| = 2^{k-1}(6+L) - 6$, where $r'' = r \vee t_{w^a_{2k}} \vee t_{w^b_{2k-1}}$.

It is straightforward to prove by induction from $k = n/2$ to 1, that

$$\begin{aligned}&|\mathcal{M}(imr_n)| = 2^{n/2}(6+L) - 6 \text{ iff } \exists a_n \forall a_{n-1} \ldots \exists a_{2k} \forall a_{2k-1}, \\ &\text{such that } |\mathcal{M}_r(imr_{2k-2})| = 2^{k-1}(6+L) - 6, \text{ where } r = \textstyle\bigvee_{i=2k-1}^{n} t_{w^{a_i}_i}.\end{aligned}$$

The claim follows when $k = 1$. ∎

**Claim 20** *$S$ is satisfiable iff $\exists a_n \forall a_{n-1} \ldots \exists a_2 \forall a_1 \in \{0,1\}$, we have $|\mathcal{M}_r(imr_0)| = L$, where $r = \bigvee_{k=1}^{n} t_{w^{a_k}_k}$.*

*Proof of Claim 20.* It is sufficient to prove that $\phi$ is satisfiable iff $\exists a_n, \dots, a_1 \in \{0,1\}$, such that for $r = \bigvee_{k=1}^{n} t_{w_k^{a_k}}$, we have $|\mathcal{M}_r(imr_0)| = L$.

Suppose we have $a_n, \dots, a_1$ such that $r = \bigvee_{k=1}^{n} t_{w_k^{a_k}}$, $|\mathcal{M}_r(imr_0)| = L$. We can construct a truth assignment $T$ by defining $T(x_k) = \mathit{true}$ if $a_k = 0$ and $T(x_k) = \mathit{false}$ if $a_k = 1$. By definition of $f_{d_{ij}}$, for each $i \in \{1..L\}$, there exists a $j$ such that the edge labeled $d_{ij}$ is on $\mathcal{M}_r(imr_0)$. By definition of $\mathcal{M}_r$, if an edge labeled $d_{ij}$ is on $\mathcal{M}_r(imr_0)$ and $\phi_{ij} = x_k$, then $a_k = 0$, and $T(x_k) = \mathit{true}$; and if $\phi_{ij} = \neg x_k$, then $a_k = 1$ and $T(x_k) = \mathit{false}$. $T(\phi_{ij}) = \mathit{true}$ in both cases. Therefore, $T$ satisfies $\phi$.

Conversely, suppose $T$ satisfies $\phi$. We can construct $r = \bigvee_{k=1}^{n} t_{w_k^{a_k}}$ from $T$ by defining $a_k = 0$ if $T(x_k) = \mathit{true}$ and $a_k = 1$ if $T(x_k) = \mathit{false}$. For each $i \in \{1..L\}$, there exists $j$ such that $T(\phi_{ij}) = \mathit{true}$, which means that the edge labeled $d_{ij}$ can be on the path $\mathcal{M}_r(imr_0)$. Therefore, $|\mathcal{M}_r(imr_0)| = L$. ∎

We now proceed with the proof of the theorem. We conclude

$$\begin{array}{lr}
S \text{ is satisfiable} & \\
\text{iff } \exists a_n \forall a_{n-1} \dots \exists a_2 \forall a_1\text{: for } r = \bigvee_{k=1}^{n} t_{w_k^{a_k}},\ |\mathcal{M}_r(imr_0)| = L & \text{(Claim 20)} \\
\text{iff } |\mathcal{M}(imr_S)| = 2^{n/2}(6+L) - 6, \text{ where } imr_S = imr_n & \text{(Claim 19)} \\
\text{iff } \mathit{maxStackSize}(P_p) = 2^{n/2}(6+L) - 6 & \text{(Lemma 11),}
\end{array}$$

so the exact maximum stack size problem is PSPACE-hard. ∎

Notice that we can combine the last part of the proof of Theorem 13 with Claim 16 to get that $S$ is not satisfiable iff $\mathit{maxStackSize}(P_p) < 2^{n/2}(6+L)-6$.

## 4 Monotonic Enriched Interrupt Programs

We now introduce an enriched version of the interrupt calculus, where we allow conditionals on the interrupt mask register. The conditional can test if some bit of the imr is on, and then take the bitwise or of the imr with a constant bit sequence; or it can test if some bit of the imr is off, and then take the bitwise and of the imr with a constant. The syntax for *enriched interrupt programs* is given by the syntax from Section 2 together with the following clauses:

$$\begin{array}{rcl}
\text{(statement) } s ::= \cdots & | & \mathsf{if(bit}\ i\ \mathsf{on)\ imr = imr} \vee imr \\
 & | & \mathsf{if(bit}\ i\ \mathsf{off)\ imr = imr} \wedge imr
\end{array}$$

The small-step operational semantics is given below:

$$\begin{array}{lr}
\langle \bar{h}, R, \mathit{imr}, \sigma, \mathsf{if}(\mathsf{bit}\ i\ \mathsf{on})\mathsf{imr} = \mathsf{imr} \vee \mathit{imr}'; a\rangle \to & \langle \bar{h}, R, \mathit{imr} \vee \mathit{imr}', \sigma, a\rangle \\
 & \text{if } \mathit{imr}(i) = 1 \\
\langle \bar{h}, R, \mathit{imr}, \sigma, \mathsf{if}(\mathsf{bit}\ i\ \mathsf{on})\mathsf{imr} = \mathsf{imr} \vee \mathit{imr}'; a\rangle \to & \langle \bar{h}, R, \mathit{imr}, \sigma, a\rangle \\
 & \text{if } \mathit{imr}(i) = 0 \\
\langle \bar{h}, R, \mathit{imr}, \sigma, \mathsf{if}(\mathsf{bit}\ i\ \mathsf{off})\mathsf{imr} = \mathsf{imr} \wedge \mathit{imr}'; a\rangle \to & \langle \bar{h}, R, \mathit{imr} \wedge \mathit{imr}', \sigma, a\rangle \\
 & \text{if } \mathit{imr}(i) = 0 \\
\langle \bar{h}, R, \mathit{imr}, \sigma, \mathsf{if}(\mathsf{bit}\ i\ \mathsf{off})\mathsf{imr} = \mathsf{imr} \vee \mathit{imr}'; a\rangle \to & \langle \bar{h}, R, \mathit{imr}, \sigma, a\rangle \\
 & \text{if } \mathit{imr}(i) = 1
\end{array}$$

Unlike the conditional statement $\mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2$ on data that has been overapproximated, our analysis will be path sensitive in the imr-conditional.

**Proposition 21** *Monotonicity of enriched interrupt programs can be checked in time exponential in the number of handlers (in co-NP).*

*Proof.* It follows from Lemma 1 that a program is monotonic iff each handler is monotonic in isolation. To check nonmonotonicity, we guess a handler and an imr value that shows it is nonmonotonic, and check in polynomial time that the handler is not monotonic for that imr. ∎

For monotonic enriched interrupt programs, both the stack boundedness problem and the exact maximum stack size problem are PSPACE-complete. To show this, we first show that the stack boundedness problem is PSPACE-hard by a generic reduction from polynomial-space Turing machines. We fix a PSPACE-complete Turing machine $M$. Given input $x$, we construct in polynomial time a program $p$ such that $M$ accepts $x$ iff $p$ has an unbounded stack. We have two handlers for each tape cell (one representing zero, and the other representing one), and a handler for each triple $(i, q, b)$ of head position $i$, control state $q$, and bit $b$. The handlers encode the working of the Turing machine in a standard way. The main program sets the bits corresponding to the initial state of the Turing machine, with $x$ written on the tape. Finally, we have an extra handler that enables itself (and so can cause an unbounded stack) which is set only when the machine reaches an accepting state. We provide the formal proof below.

**Theorem 22** *The stack boundedness problem for monotonic enriched interrupt programs is PSPACE-hard.*

*Proof.* Fix a PSPACE-complete Turing Machine $M$ which on any input $x$ uses at most $r(|x|)$ space to decide whether $M$ accepts $x$, where $r$ is a polynomial. Given any input $x$, the TM $M$ always halts and answers *accept* or *reject.* It is PSPACE-complete to decide given $M$ and $x$ whether $M(x)$ accepts or rejects

[5]. Let the states of $M$ be $Q = \{q_1, q_2, ..., q_t\}$ with $q_t$ as the accepting state. Given such a machine $M$ and an input $x$ we reduce the problem of whether $q_t$ is reachable to the stack boundedness analysis of a interrupt driven program such that the stack size is infinite iff $q_t$ is reachable. We construct, from $M$ and $x$ a monotone interrupt program $p(M, x)$ such that $M$ accepts $x$ iff the stack of $p(M, x)$ is unbounded.

Now we describe the *imr* and handlers. The total number of bits in the imr is $1 + 2 \times r(|x|) + 2 \times |Q| \times r(|x|)$. The first (0th) bit is the master bit. There are two bits for each position of the tape, so $2r(|x|)$ bits encode the tape of the TM $M$. Further, the imr has one bit for every tuple $(i, q, \sigma)$ for each tape position $i \in \{1, \ldots, r(|x|)\}$, TM state $q \in Q$, and symbol $\sigma \in \{0, 1\}$.

For $k \in \{1, \ldots, r(|x|)\}$, the $k$-th tape cell is stored in bits $2k - 1$ and $2k$. For all $1 \leq k \leq r(|x|)$, the $(2k - 1)$-th bit is 1 and $2k$-th bit is 0 if the tape cell in the $k$-th position of $M$ is 0. Similarly, or all $1 \leq k \leq r(|x|)$, the $(2k - 1)$-th bit is 0 and $2k$-th bit is 1 if the tape cell in the $k$-th position of $M$ is 1.

The bit for $(i, q, \sigma)$ bit is turned on when the head of $M$ is in the $i$-th cell, the control is in state $q$ and $\sigma$ is written in the $i$-th cell. Formally, the bit $(2r(|x|) + 2kr(|x|) + 2i - 1)$ is 1 if the head of TM $M$ is in position $i$, the TM is in state $q_k$, and the $i$th tape cell reads 0. The code for this handler implements the transition for the TM $M$ corresponding to $(q_k, 0)$. Similarly, the $(2r(|x|) + 2kr(|x|) + 2i)$-th bit is 1 if the head of TM $M$ is in position $i$, the TM is in state $q_k$, and the $i$th tape cell reads 1. The code for this handler implements the transition for the TM $M$ corresponding to $(q_k, 1)$.

The first $2 \times r(|x|)$ handlers which encode the tape cells do not change the status of the imr, that is, the body of the handler $\mathtt{H}_i$ contains only the statement `iret` for $i = 1, \ldots, 2r(|x|)$.

We show how to encode the transition of TM $M$ in the code for the handler. We first introduce some short-hand notation for readability.

- The operation write$(\sigma, i)$ writes $\sigma$ in the $i$-th cell tape of $M$. This is shorthand for the following code:

(1) if $\sigma = 0$, the code is $\mathsf{imr} = \mathsf{imr} \ \wedge \ \neg t_{2\times i}; \mathsf{imr} = \mathsf{imr} \ \vee \ t_{2\times i-1}$ (recall that $t_j$ denotes the imr with all bits 0's and only the $j$-th bit 1).

(2) if $\sigma = 1$, the code is $\mathsf{imr} = \mathsf{imr} \ \wedge \ \neg t_{2\times i-1}; \mathsf{imr} = \mathsf{imr} \ \vee \ t_{2\times i}$.

- The operation set($i$-th bit on) sets the $i$-th bit of the imr on. This is shorthand for $\mathsf{imr} = \mathsf{imr} \vee t_i$.
- The operation set($i$-th bit off) sets the $i$-th bit of the imr off. This is shorthand for $\mathsf{imr} = \mathsf{imr} \wedge \neg t_i$.

We now give the encoding of the transition relation of the TM.

```
Handler $H_j$ {
    1. set($j$-th bit off);
    2. write $(\sigma, i)$
    3. if(bit (2($i+1$)) on) { /* Check if the $(i+1)$-th TM cell is 1 */
        3.1 set ($l$-th bit on)
        3.2 set (0th bit on) /* set master bit */
    }
    4. if(bit $(2(i+1)-1)$ on) { /* Check if the $(i+1)$-th TM cell is 0 */
        4.1 set (($l-1$)-th bit on)
        4.2 set (0th bit on) /* set master bit */
    }
    5. imr = imr $\wedge$ 000...00..00
    6. iret
}
```

Fig. 3. Code for Turing Machine transition

```
Main {
    1. imr = imr $\vee$ $c$
        where $c$ is an imr constant which correctly encodes
        the starting configuration of $M$ on $x$.
    2. loop skip
}
```

Fig. 4. Code for the main program

(1) Consider Handler $H_j$ where $j = 2r(|x|) + 2kr(|x|) + 2i - 1$ and the transition for $(q_k, 0)$ is $(q_{k'}, \sigma, R)$. Let $l = 2r(|x|) + 2k'r(|x|) + 2(i+1)$. The code for handler $H_j$ is shown in Figure 3. Note that the two consecutive $imr_{or}$ statements in line 3.1 and 3.2 and 4.1 and 4.2 can be folded into a single $imr_{or}$ statement, we separate them for readability. We can encode the other transition types similarly.

(2) The code for a handler $H_j$ corresponding to an accepting state sets the master bit on, and returns.

(3) The main program initializes the imr with the initial configuration, and enters an empty loop. The code for the main program is shown in Figure 4.

**Lemma 23** *If $M$ accepts $x$ then the stack of $p(M, x)$ grows unbounded.*

*Proof.* We show that there is a sequence of interrupt occurrences such that a handler corresponding to the accepting state is called. Whenever the $l$-th or the $(l-1)$-th bit in turned on and the master bit is turned on in lines 3.1,3.2 or 4.1,4.2 of Figure 3, an interrupt of type $l$ or $(l-1)$ occurs. Hence following this sequence a handler corresponding to the accepting state is called and then the stack can grow unbounded as the handler sets the master bit on without disabling itself. ∎

**Lemma 24** *If $M$ halts without accepting $x$ then the stack of $p(M, x)$ is bounded.*

*Proof.* If any handler which encodes the transitions of the $M(x)$ returns it sets all the bits of *imr* to 0 (Statement 6 in Figure 3). Hence all the following checks in the statements 3 and 4 will fail and the master bit will not be set any further. Hence the stack would go empty. So if the stack is unbounded then no handler which encodes the configuration of the machine $M$ returns. If the stack is unbounded and the accepting state is not reached then there is a handler $h$ which encodes the transition of the machine $M$ and it occurs infinitely many times in the stack. This means one of the configurations of $M(x)$ can be repeated. This means there is a cycle in the configuration graph of $M(x)$ and hence it cannot halt. But this is a contradiction, since our TM always halts. This proves that if the accepting state is not reached then the stack is bounded. ∎

From Lemmas 23, 24 the theorem follows. ∎

We now give a PSPACE algorithm to check the exact maximum stack size. Since we restrict our programs to be monotonic it follows from Lemma 4 that the maximum length of the stack can be achieved with no handler returning in between. Given a program $p$ with $m$ statements and $n$ handlers, we label the statements as $pc_1, \ldots pc_m$. Let $PC$ denote the set of all statements, i.e., $PC = \{pc_1, \ldots pc_m\}$. Consider the graph $G_p$ where there is a node $v$ for every statement with all possible imr values (i.e., $v = \langle pc, imr \rangle$ for some value among $PC$ and some imr value). Let $v = \langle pc, imr \rangle$ and $v' = \langle pc', imr' \rangle$ be two nodes in the graph. There is an edge between $v, v'$ in $G$ if any of the following two conditions hold:

- on executing the statement at $pc$ with imr value $imr$ the control goes to $pc'$ and the value of imr is $imr'$. The weight of this edge is 0.
- $pc'$ is a starting address of a handler $h_i$ and $enabled(imr, i)$ and $imr' = imr \wedge \neg t_0$. The weight of this edge is 1.

We also have a special node in the graph called *target* and add edges to *target* of weight 0 from all those nodes which correspond to a $pc \in PC$ which is a *iret* statement. This graph is exponential in the size of the input as there are $O(|PC| \times 2^n)$ nodes in the graph. The starting node of the graph is the node with $pc_1$ and $imr = 0$. If there is a node in the graph which is the starting address of a handler $h$ and which is reachable from the start node and also self-reachable then the stack length would be infinite. This is because the sequence of calls from the starting statement to the handler $h$ is first executed and then the cycle of handler calls is repeated infinitely many times. As the handler $h$ is in stack when it is called again the stack would grow infinite. Since there is a sequence of interrupts which achieves the maximum stack length without any handler returning in between (follows from Lemma 4) if there is no cycle in $G_p$ we need to find the longest path in the DAG $G_p$.

**Theorem 25** *The exact maximum stack size for monotonic enriched interrupt programs can be found in time linear in the size of the program and exponential in the number of handlers. The complexity of exact maximum stack size for monotonic enriched interrupt programs is PSPACE.*

In polynomial space one can generate in lexicographic order all the nodes that have a *pc* value of the starting statement of a handler. If such a node is reachable from the start node, and also self-reachable, then the stack size is infinite. Since the graph is exponential, this can be checked in PSPACE. If no node has such a cycle, we find the longest path from the start node to the target. Again, since longest path in a DAG is in NLOGSPACE, this can be achieved in PSPACE. It follows that both the stack boundedness and exact maximum stack size problems for monotonic enriched interrupt programs are PSPACE-complete.

## 5 Nonmonotonic Enriched Interrupt Programs

In this section we consider interrupt programs with tests, but do not restrict handlers to be monotonic. We give an EXPTIME algorithm to check stack boundedness and find the exact maximum stack size for this class of programs. The algorithm involves computing longest context-free paths in context-free DAGs, a technique that may be of independent interest.

### *5.1 Longest Paths in Acyclic Context-free Graphs*

We define a context-free graph as in [9]. Let $\Sigma$ be a finite alphabet. A *context-free graph* is a tuple $G = (V, E, \Sigma)$ where $V$ is a set of nodes and $E \subseteq (V \times V \times (\Sigma \cup \{\tau\}))$ is a set of labeled edges (and $\tau$ is a special symbol not in $\Sigma$).

We shall particularly consider the context-free language of matched parentheses. Let $\Sigma = \{(^1, (^2, \ldots, (^k, )^1, )^2, \ldots, )^k\}$ be the alphabet of opening and closing parentheses. Let $\mathcal{L}$ be the language generated by the context-free grammar

$$M \to M(^i S \quad | \quad S \text{ for } 1 \leq i \leq k$$

$$S \to \epsilon \quad | \quad (^i S)^i S \text{ for } 1 \leq i \leq k$$

from the starting symbol $M$. Thus $\mathcal{L}$ defines words of matched parentheses with possibly some opening parentheses mismatched. From this point, we restrict our discussion to this $\Sigma$ and the language $\mathcal{L}$.

**Algorithm 2 Function LongestContextFreePath**

```
Input: A context-free DAG $G$, a vertex $v_1$ of $G$
Output: For each vertex $v$ of $G$, return the length of the longest
    context-free path from $v$ to $v_1$, and
    0 if there is no context-free path from $v$ to $v_1$
1. For each vertex $v_j \in V$:  $val[v_j] = 0$
2. Construct the transitive closure matrix $T$ such that
    $T[i,j] = 1$ iff there is a context-free path from $i$ to $j$
3. For $j = 1$ to $n$:
    3.1 For each immediate successor $v_i$ of $v_j$ such that
        the edge $e_{v_j,v_i}$ from $v_j$ to $v_i$ satisfies $wt(e_{v_j,v_i}) \geq 0$:
        $val[v_j] = \max\{val[v_j], val[v_i] + wt(e_{v_j,v_i})\}$
    3.2 For each vertex $v_i \in V$:
        3.2.1 if($T[i,j]$) ($v_j$ is context-free reachable from $v_i$)
            $val[v_i] = \max\{val[v_i], val[v_j]\}$
```

We associate with each edge of $G$ a *weight function* $wt : E \to \{0, +1, -1\}$ defined as follows:

- $wt(e) = 0$ if $e$ is of the form $(v, v', \tau)$,
- $wt(e) = -1$ if $e$ is of the form $(v, v', )^i)$ for some $i$,
- $wt(e) = 1$ if $e$ is of the form $(v, v', (^i)$ for some $i$.

A *context-free path* $\pi$ in a context-free graph $G$ is a sequence of vertices $v_1, v_2, \ldots v_k$ such that for all $i = 1 \ldots k-1$, there is an edge between $v_i$ and $v_{i+1}$, i.e., there is a letter $\sigma \in \Sigma \cup \{\tau\}$ such that $(v_i, v_{i+1}, \sigma) \in E$ and the projection of the labels along the edges of the path to $\Sigma$ is a word in $\mathcal{L}$. Given a context-free path $\pi$ with edges $e_1, e_2, \ldots e_k$ the *cost* of the path $Cost(\pi)$ is defined as $\sum_i wt(e_i)$. Note that $Cost(\pi) \geq 0$ for any context-free path $\pi$. A context-free graph $G$ is a *context-free DAG* iff there is no cycle $C$ of $G$ such that $\sum_{e \in C} wt(e) > 0$. Given a context-free DAG $G = (V, E, \Sigma)$ we define an ordering $order : V \to \mathbb{N}$ of the vertices satisfying the following condition: if there is a path $\pi$ in $G$ from vertex $v_i$ to $v_j$ and $Cost(\pi) > 0$ then $order(v_j) < order(v_i)$. This ordering is well defined for context-free DAGs. Let $G$ be a context-free DAG $G$, and let $V = \{v_1, v_2, \ldots v_n\}$ be the ordering of the vertex set consistent with *order* (i.e., $order(v_i) = i$). We give a polynomial-time procedure to find the longest context-free path from any node $v_i$ to $v_1$ in $G$.

The correctness proof of our algorithm uses a function *Num* from paths to $\mathbb{N}$. Given a path $\pi$ we define $Num(\pi)$ as $\max\{order(v) \mid v \text{ occurs in } \pi\}$. Given a node $v$ let $L_v = \{L_1, L_2, \ldots L_k\}$ be the set of longest paths from $v$ to $v_1$. Then we define $Num_{v_1}(v) = \min\{Num(L_i) \mid L_i \in L_v\}$. The correctness of the algorithm follows from the following set of observations.

**Lemma 26** *If there is a longest path $L$ from a node $v$ to $v_1$ such that $L$ starts with an opening parenthesis $(^i$ that is not matched along the path $L$ then $order(v) = Num_{v_1}(v)$.*

*Proof.* Consider any node $v'$ in the path $L$. Since the first opening parenthesis is never matched, the sub-path $L(v, v')$ of $L$ from $v$ to $v'$ satisfies $Cost(L(v, v')) > 0$. Hence it follows that for all nodes $v'$ in $L$, we have $order(v') < order(v)$. Thus $Num_{v_1}(v) = order(v)$. ∎

**Lemma 27** *A node $v$ in the DAG $G$ satisfies the following conditions.*

- *If $Num_{v_1}(v) = order(v) = j$ then within the execution of Statement 3.1 of the $j$-th iteration of Loop 3 of function LongestContextFreePath, $val[v]$ is equal to the cost of a longest path from $v$ to $v_1$.*
- *If $order(v) < Num_{v_1}(v) = j$ then by the $j$-th iteration of Loop 3 of function LongestContextFreePath $val[v]$ is equal to the cost of a longest path from $v$ to $v_1$.*

*Proof.* We prove by induction on $Num_{v_1}(v)$. The base case holds when $Num_{v_1}(v) = 1$, since $v = v_1$. We now prove the inductive case. If the value of the longest path is 0 then it was fixed initially and it cannot decrease. Otherwise, there is a positive cost longest path from $v$ to $v_1$. We consider the two cases when $order(v) = Num_{v_1}(v)$ and when $order(v) < Num_{v_1}(v)$.

Case $order(v) = Num_{v_1}(v) = j$. Let $L(v, v_1)$ be a longest path from $v$ to $v_1$ such that $Num(L(v, v_1)) = order(v)$. We consider the two possible cases.

(1) The longest path $L(v, v_1)$ is such that it starts with a opening parenthesis which is never matched. Let $v''$ be the successor of $v$ in $L(v, v_1)$. Hence $order(v'') < order(v) = Num_{v_1}(v) = j$. Also the sub-path $L(v'', v_1)$ of $L(v, v_1)$ is a longest path from $v''$ to $v_1$ (since otherwise we could have a greater cost path from $v$ to $v_1$ by following the path from $v$ to $v''$ and then the path from $v''$ to $v_1$). Hence $Num_{v_1}(v'') < Num_{v_1}(v) = j$. By the induction hypothesis, before the $j$-th iteration $val[v'']$ is equal to the cost of the longest path from $v''$ to $v_1$. Hence during the $j$-th iteration of Loop 3, when the loop of statement 3.1 is executed and $v''$ is chosen as $v$'s successor then $val[v]$ is set to the cost of the longest path from $v$ to $v_1$.

(2) The longest path $L(v, v_1)$ goes through a node $v'$ such that the cost of the subpath of $L(v, v')$ of $L(v, v_1)$ satisfies $Cost(L(v, v')) = 0$ and there is a opening parenthesis from $v'$ which is not matched in $L(v, v_1)$. Clearly the sub-path $L(v', v_1)$ must be a longest path from $v'$ to $v_1$ as otherwise $L(v, v_1)$ would not have been a longest path. It follows from Lemma 26 that $Num_{v_1}(v') = order(v') = k < j$. By the induction hypothesis, by the end of Statement 3.1 of $k$-th iteration

$val[v']$ is equal to the longest path from $v'$ to $v_1$. As $v$ can context-free reach $v'$ we have during the execution of statement 3.2 of the $k$-th iteration $val[v]$ is equal to the longest path from $v$ to $v_1$.

Case $order(v) < Num_{v_1}(v) = j$. Let $L(v, v_1)$ be a longest path from $v$ to $v_1$. The longest path $L(v, v_1)$ goes through a node $v'$ such that the cost of the subpath of $L(v, v')$ of $L(v, v_1)$ satisfies $Cost(L(v, v')) = 0$ and there is an opening parenthesis from $v'$ which is not matched in $L(v, v_1)$. Clearly the sub-path $L(v', v_1)$ must be a longest path from $v'$ to $v_1$ as otherwise $L(v, v_1)$ would not have been a longest path. It follows from Lemma 26 that $Num_{v_1}(v') = order(v') = k$. By hypothesis by the end of Statement 3.1 of $k$-th iteration $val[v']$ is equal to the cost of the longest path from $v'$ to $v_1$. As $v$ can context-free reach $v'$ we have during the execution of statement 3.2 of the $k$-th iteration $val[v]$ is equal to the longest path from $v$ to $v_1$. As $Num_{v_1}(v) \geq order(v')$ (since $v'$ occurs in the path) it follows by the end of $j$-th iteration $val[v]$ is equal to the cost of the longest path from $v$ to $v_1$.

Notice also that every time $val[v]$ is updated (to $c$, say), it is easy to construct a witness path that shows that the cost of the longest path is at least $c$. This concludes the proof. ∎

From the above two lemmas, we get the following.

**Corollary 28** *At the end of function* LongestContextFreePath$(G, v_1)$*, for each vertex $v$, the value of $val[v]$ is equal to the longest context-free path to $v_1$, and equal to zero if there is no context-free path to $v_1$.*

We now consider the time complexity of the function LongestContextFreePath. In the Function LongestContextFreePath the statement 3.2.1 gets executed at most $n^2$ times since the loop on line 3 gets executed $n$ times at most and the nested loop on line 3.2 also gets executed $n$ times at most. The context-free transitive closure can be constructed in $O(n^3)$ time [12]. Hence the complexity of our algorithm is polynomial and it runs in time $O(n^2 + n^3) = O(n^3)$.

**Theorem 29** *The longest context-free path of a context-free DAG can be found in time cubic in the size of the graph.*

To complete our description of the algorithm, we must check if a given context-free graph is a context-free DAG, and generate the topological ordering *order* for a context-free DAG. We give a polynomial-time procedure to check whether a given context-free graph is a DAG. Let $G = (V, E, \Sigma)$ be a given context-free graph, and let $V = \{1, 2, \ldots n\}$. For every node $k \in V$ the graph $G$ can be unrolled as a DAG for depth $|V|$, and it can be checked if there is a path $\pi$ from $k$ to $k$ such that $Cost(\pi) > 0$. Given the graph $G$ and a node $k$ we create a context-free DAG $G_k = (V_k, E_k, \Sigma)$ as follows:

1. $V_k = \{k_0\} \cup \{(i,j) \mid 1 \le i \le n-2, 1 \le j \le n\} \cup \{k_{n-1}\}$
2. $E_k = \{\langle k_0, (1,j), *\rangle \mid \langle k, j, *\rangle \in E\} \cup \{\langle (i,j), (i+1,j'), *\rangle \mid \langle j, j', *\rangle \in E\}$
   $\cup \{\langle (n-2,j), k_{n-1}, *\rangle \mid \langle j, k, *\rangle \in E\}$
   $\cup \{\langle k_0, (1,k), \tau\rangle\} \cup \{\langle (i,k), (i+1,k), \tau\rangle\}$

where $*$ can represent a opening parenthesis, closing parenthesis or can be $\tau$. Notice that the edges in the last line ensure that if there is a cycle of positive cost from $k$ to itself with length $t < n$ then it is possible to go from $k_0$ to $(n-t-1,k)$ and then to reach $k_{n-1}$ by a path of positive cost.

We can find the longest context-free path from $k_0$ to $k_n$ in $G_n$ (by the function LongestContextFreePath). If the length is positive, then there is a positive cycle in $G$ from $k$ to $k$. If for all nodes the length of the longest path in $G_n$ is 0, then $G$ is a context-free DAG and the longest context-free path can be computed in $G$. Given a context-free DAG $G$ we can define $\mathit{order}(v)$ in polynomial time. If a vertex $v$ can reach $v'$ and $v'$ can reach $v$ put them in the same group of vertices. Both the path from $v$ to $v'$ and $v'$ to $v$ must be cost 0 since there is no cycle of positive cost. Hence the ordering of vertices within a group can be arbitrary. We can topologically order the graph induced by the groups and then assign an order to the vertices where vertices in the same group are ordered arbitrarily.

### *5.2 Stack Size Analysis*

We present an algorithm to check for stack boundedness and exact maximum stack size. The idea is to perform context-free longest path analysis on the state space of the program. Given a program $p$ with $m$ statements and $n$ handlers, we label the statements as $pc_1, pc_2, \ldots pc_m$. Let $PC = \{pc_1, \ldots pc_m\}$ as before. We construct a context-free graph $G_p = \langle V, E, \Sigma\rangle$, called the *state graph of* $p$, where $\Sigma = \{(^1, (^2, \ldots, (^m, )^1, )^2, \ldots )^m\}$ as follows:

- $V = PC \times \mathit{IMR}$, where $\mathit{IMR}$ is the set of all $2^n$ possible imr values.
- $E \subseteq (V \times V \times (\Sigma \cup \{\tau\})$ consists of the following edges.

(1) Handler call: $(v, v', (_i) \in E$ iff $v = (pc_i, \mathit{imr}_1)$ and $v' = (pc_j, \mathit{imr}_2)$ and $pc_j$ is the starting address of some handler $h_j$ such that $\mathit{enabled}(\mathit{imr}_1, j)$ and $\mathit{imr}_2 = \mathit{imr}_1 \wedge \neg \mathrm{t}_0$.
(2) Handler return: $(v', v, )_i) \in E$ iff $v = (pc_i, \mathit{imr}_1)$ and $v' = (pc_j, \mathit{imr}_2)$ and $pc_j$ is the iret statement of some handler and $\mathit{imr}_1 = \mathit{imr}_2 \vee \mathrm{t}_0$.
(3) Statement execution: $(v, v', \tau) \in E$ iff $v = (pc_i, \mathit{imr}_1)$ and $v' = (pc_j, \mathit{imr}_2)$ and executing the statement at $pc_i$ with imr value $\mathit{imr}_1$ the control goes to $pc_j$ and the imr value is $\mathit{imr}_2$.

The vertex $(pc_1, 0)$ is the starting vertex of $G_p$. Let $G'_p$ be the induced subgraph of $G_p$ containing only nodes that are context-free reachable from the start

**Algorithm 3 Function StackSizeGeneral**

```
Input: Enriched interrupt program $p$
Output: $maxStackSize(P_p)$
1. Build the state graph $G_p = \langle V, E, \Sigma \rangle$ from the program $p$
2. Let $V' = \{v' \mid$ there is a context-free path from the starting vertex to $v'\}$
3. Let $G'_p$ be the subgraph of $G_p$ induced by the vertex set $V'$
4. If $G'_p$ is not a context-free DAG then return "infinite"
5. Else create $G''_p = (V'', E'', \Sigma)$ as follows :
    5.1 $V'' = V' \cup \{target\}$ and $E'' = E' \cup \{(v, target, \tau) \mid v \in V'\}$
6. Return the value of the longest context-free path
    from the starting vertex to $target$
```

node. If $G'_p$ is not a context-free DAG then we report that stack is unbounded. Otherwise, we create a new DAG $G''_p$ by adding a new vertex *target* and adding edges to *target* from all nodes of $G'_p$ of weight 0. Then, we find the value of a longest context-free path from the start vertex to *target* in the DAG $G''_p$.

From the construction of the state graph, it follows that there is a context-free path from a vertex $v = (pc, imr)$ to $v' = (pc', imr')$ in the state graph $G_p$ if there exists stores $R$, $R'$ and stacks $\sigma$, $\sigma'$ such that $\langle \bar{h}, R, imr, \sigma, pc \rangle \to^* \langle \bar{h}, R', imr', \sigma' pc' \rangle$. Moreover, if $G'_p$ is the reachable state graph then there exists $K$ such that for all $P'$ such that $P_p \to^* P'$ we have $|P'.stk| \leq K$ iff $G'_p$ is a context-free DAG. To see this, first notice that if $G'_p$ is not a context-free DAG then there is a cycle of positive cost. Traversing this cycle infinitely many times makes the stack grow unbounded. On the other hand, if the stack is unbounded then there is a program address that is visited infinitely many times with the same imr value and the stack grows between the successive visits. Hence there is a cycle of positive cost in $G'_p$. These observations, together with Theorem 29 show that function StackSizeGeneral correctly computes the exact maximum stack size of an interrupt program $p$.

**Theorem 30** *The exact maximum stack size of nonmonotonic enriched interrupt programs can be found in time cubic in the size of the program and exponential in the number of handlers.*

*Proof.* The number of vertices in $G_p$ is $m \times 2^n$, for $m$ program statements and $n$ interrupt handlers. It follows from Theorem 29 and the earlier discussion that the steps 1, 2, 3, 4, 5, and 6 of StackSizeGeneral can be computed in time polynomial in $G_p$. Since $G_p$ is linear in the size of the input program $p$, and exponential in the number of handlers, we have a procedure for determining the exact maximum stack size of nonmonotonic enriched interrupt programs that runs in $O(m^3 8^n)$. This gives an EXPTIME procedure for the exact maximum stack size problem. ∎

While our syntax ensures that all statements that modify the imr are monotonic, this is not a fundamental limitation for the above algorithm. Indeed, we can extend the syntax of the enriched calculus to allow any imr operations, and the above algorithm still solves the exact maximum stack size problem, with no change in complexity.

We leave open whether the exact maximum stack size problem for nonmonotonic interrupts programs, in the nonenriched and enriched cases, is EXPTIME-hard or PSPACE-complete (PSPACE-hardness follows from Theorem 22). One can note that the time to execute the algorithms grows exponentially with the *number* of interrupt handlers, which is typically small, and cubically with the *size* of the interrupt handler programs.

*Acknowledgments*

Palsberg, Ma, and Zhao were supported by the NSF ITR award 0112628. Henzinger, Chatterjee, and Majumdar were supported by the AFOSR grant F49620-00-1-0327, the DARPA grants F33615-C-98-3614 and F33615-00-C-1693, the MARCO grant 98-DT-660, and the NSF grants CCR-0208875 and CCR-0085949.

## References

[1] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *ICSE: International Conference on Software Engineering*, pp. 47–56. ACM/IEEE, 2001.

[2] G. G. Hillebrand, P. C. Kanellakis, H. G. Mairson, and M. Y. Vardi. Undecidable boundedness problems for datalog programs. Journal of Logic Programming 25(2):163–190, 1995.

[3] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL: Principles of Programming Languages*, pp. 410–423. ACM, 1996.

[4] J. Palsberg and D. Ma. A typed interrupt calculus. In *FTRTFT: Formal Techniques in Real-Time and Fault-tolerant Systems*, LNCS 2469, pp. 291–310. Springer, 2002.

[5] C.H. Papadimitriou. *Computational Complexity.* Addision-Wesley, 1994.

[6] C.H. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). Journal of Computer and System Sciences 28:244-259, 1984.

[7] L. Pareto. *Types for Crash Prevention.* PhD thesis, Chalmers University of Technology, 2000.

[8] J. Regehr, A. Reid, and K. Webb, Eliminating stack overflow by abstract interpretation, In *EMSOFT'03: Third International Workshop on Embedded Software*, 2003. To appear.

[9] T.W. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pp. 49–61. ACM, 1995.

[10] T.W. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SAS 03: Static Analysis Symposium*, LNCS 2694, pp. 189–213. Springer, 2003.

[11] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *PADL: Practical Aspects of Declarative Languages*, LNCS 2257, pp. 155–172. Springer, 2002.

[12] M. Yannakakis. Graph-theoretic methods in database theory. In *PODS: Principles of Database Systems*, pp. 203–242. ACM, 1990.