Bounded Model Checking of Pointer Programs

W. Charatonik, L. Georgieva, P. Maier[*]

MPI–I–2005–2–002 June 2005

[*]Corresponding author

Authors' Addresses

Witold Charatonik
Instytut Informatyki, Uniwersytet Wrocławski
Wrocław, Poland
`http://www.ii.uni.wroc.pl/~wch/`
`Witold.Charatonik@ii.uni.wroc.pl`

Lilia Georgieva
School of Mathematics and Computer Science, Heriot-Watt-University
Edinburgh, United Kingdom
`http://www.macs.hw.ac.uk/~lilia/`
`lilia@macs.hw.ac.uk`

Patrick Maier
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
`http://www.mpi-inf.mpg.de/~maier/`
`maier@mpi-inf.mpg.de`

Publication Notes

This report is the extended version of a paper presented at the *Annual Conference of the European Association for Computer Science Logic* (CSL '05), Oxford, UK, August 22 to 25, 2005.

Abstract

We propose a bounded model checking procedure for programs manipulating dynamically allocated pointer structures. Our procedure checks whether a program execution of length n ends in an error (e. g., a NULL dereference) by testing if the weakest precondition of the error condition together with the initial condition of the program (e. g., program variable x points to a circular list) is satisfiable. We express error conditions as formulas in the 2-variable fragment of the Bernays-Schönfinkel class with equality. We show that this fragment is closed under computing weakest preconditions. We express the initial conditions by unary relations which are defined by monadic Datalog programs.

Our main contribution is a small model theorem for the 2-variable fragment of the Bernays-Schönfinkel class extended with least fixed points expressible by certain monadic Datalog programs. The decidability of this extension of first-order logic gives us a bounded model checking procedure for programs manipulating dynamically allocated pointer structures. In contrast to SAT-based bounded model checking, we do not bound the size of the heap a priori, but allow for pointer structures of arbitrary size. Thus, we are doing bounded model checking of infinite state transition systems.

Keywords

Decidable fragments, pointer verification, model checking.

1 Introduction

Automatic verification of programs that can manipulate pointers into dynamically allocated memory is a challenging task, even for simple safety properties such as "there is no NULL dereference". In general, the problem is undecidable as the reachable state space of programs with dynamic memory allocation is infinite. Decidability can be traded off for precision by over- or under-approximating the reachable state space. Over-approximation is used by techniques based on abstraction, e. g., the shape analysis framework [23]. In bounded model checking (BMC), the set of reachable states is under-approximated by limiting the runtime or the memory of a program to an a priori chosen bound [6, 13]. Thus, BMC cuts off all states that require more than the chosen amount of time or space to be reached. As a consequence of under-approximation, BMC can not prove safety properties (unless the diameter of the state space is less than the time bound), it can only detect their violation. Progressively increasing the bound actually yields a semi-algorithm for detecting errors, provided the BMC problem is decidable.

The decidability of the BMC problem may seem trivial at first view. Note, however, that although we assume an explicit bound on the length of the program execution, we do not have any bound on the size of the initial data structure. This implies that the explored model is an infinite and infinitely branching transition system: given an initial condition like "the variable x points to a structure of type T" we have to consider all transitions from the initial state to a state where x points to a structure of type T and size n, for infinitely many numbers n. Although it is obvious that in a finite execution a program may explore only a finite fragment of the initial data structure and it is relatively easy to compute a bound on the size of this explored part, this observation alone still does not give any bound on the size of the initial data structure as a whole. In other words, even if we are in a bounded-model-checking setup, we still have to deal with an infinite set of reachable states due to the infinite branching in the underlying transition system. To overcome this problem we prove a kind of pumping lemma that implies that it is enough to consider initial structures up to a certain size.

The worst-case complexity of our method (2-NEXPTIME) may also look discouraging. Again one has to note that the doubly-exponential blowup comes from the specification of the initial data structure. As we show in section 3.3, in common cases the complexity is doubly exponential in the specification, but not in the length of the execution. In particular, for non-branching pointer structures like lists, and for fixed programs, the complexity boils down to NPTIME. Note also that it is not appropriate to directly compare the general complexity of our method with other approaches where the initial structure is given explicitly or its size is a priori bounded — one should not expect that any method could explore a data structure of doubly exponential size in less than doubly exponential time.

Our method to decide the BMC problem relies on the following observations.

- Error conditions like "x is a dangling pointer" are expressible in the 2-

variable fragment of the Bernays-Schönfinkel class with equality.

- The fragment is closed under weakest preconditions w. r. t. finite paths.
- Data structures like trees, singly or doubly linked lists and even circular lists are expressible in a fragment of monadic Datalog.
- The combination of the Datalog fragment and the above fragment of the Bernays-Schönfinkel class is decidable.

In section 2, we reduce the BMC problem to satisfiability in the Bernays-Schönfinkel class with Datalog. Section 3 presents our main technical contribution: decidability of satisfiability of the 2-variable fragment of the Bernays-Schönfinkel class with equality extended by a certain class of monadic Datalog programs. This result follows from a small model property of the logic, which we prove by a kind of pumping technique.

2 Pointer Programs

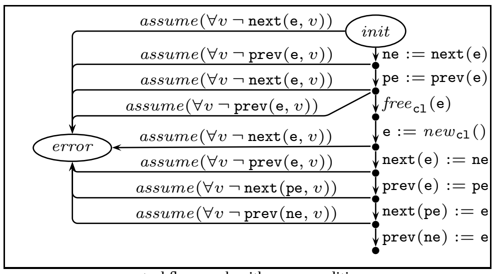
We investigate imperative programs that manipulate dynamic data structures on the heap. Given a program and a specification of its input, i. e., the heap contents upon start-up, we want to check safety properties, e. g., whether the program can crash by dereferencing a dangling pointer.

2.1 Syntax

A program P consists of two parts, a struct declaration specifying templates of heap cells and a control flow graph specifying the possible program executions.

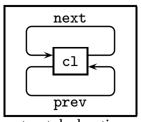
A *struct declaration* is finite directed graph with uniquely labeled edges, i. e., there are no two edges with the same label. We call the vertices of this graph *templates*, the edge labels we call *fields*. An edge label r is a field of a template T, denoted by $r \in T$, if r labels one of T's outgoing edges; note that due to unique labeling, each r is a field of exactly one template T.

A *control flow graph (CFG)* is a finite directed graph whose edges are labeled by actions. We call the vertices of a control flow graph *(control) locations*, and we assume that there is a distinguished location $init$, which does not have incoming edges. The set of *actions* Act is defined by the following grammar, where T is a template, s is a field, x and y are program variables, e is a program variable or a constant (including $NULL$), and γ is a formula built from program variables and



$assume(\forall v\, \neg\, \texttt{next}(\texttt{e}, v))$
$init$
$assume(\forall v\, \neg\, \texttt{prev}(\texttt{e}, v))$
$\texttt{ne} := \texttt{next(e)}$
$assume(\forall v\, \neg\, \texttt{next}(\texttt{e}, v))$
$\texttt{pe} := \texttt{prev(e)}$
$assume(\forall v\, \neg\, \texttt{prev}(\texttt{e}, v))$
$free_{\texttt{cl}}(\texttt{e})$
$assume(\forall v\, \neg\, \texttt{next}(\texttt{e}, v))$
$\texttt{e} := new_{\texttt{cl}}()$
$error$
$assume(\forall v\, \neg\, \texttt{prev}(\texttt{e}, v))$
$\texttt{next(e)} := \texttt{ne}$
$assume(\forall v\, \neg\, \texttt{next}(\texttt{pe}, v))$
$\texttt{prev(e)} := \texttt{pe}$
$assume(\forall v\, \neg\, \texttt{prev}(\texttt{ne}, v))$
$\texttt{next(pe)} := \texttt{e}$
$\texttt{prev(ne)} := \texttt{e}$

control flow graph with error conditions



struct declaration

$\texttt{cl}(u) \leftarrow u \approx \texttt{e}, \texttt{next}(u, v), \texttt{cl}'(v).$
$\texttt{cl}'(u) \leftarrow u \approx \texttt{e}.$
$\texttt{cl}'(u) \leftarrow \texttt{next}(u, v), \texttt{cl}'(v).$

Datalog program $\mathcal{P}$

$\forall u, v(\texttt{prev}(u, v) \Leftrightarrow \texttt{next}(v, u)) \wedge$
$\forall v\, \neg\, \texttt{next}(NULL, v) \wedge$
$\forall v\, \neg\, \texttt{prev}(NULL, v)$

axiom ϕ

Figure 1: Replacing an element in a doubly linked circular list by a new one; the initial condition that `e` points to a doubly linked circular list is expressed by the formula $\phi \wedge \mathcal{P} \wedge \texttt{cl}(\texttt{e})$.

constants (including $NULL$), the equality symbol $\approx$ and the Boolean connectives.

$Act ::= assume(\gamma)$	*Assume condition γ.*
$\mid\ y := e$	*Assign the value e to the variable y.*
$\mid\ y := s(x)$	*Read the s-field of the cell pointed to by x into y.*
$\mid\ s(x) := e$	*Write e to the s-field of the cell pointed to by x.*
$\mid\ free_T(x)$	*Deallocate the T-cell pointed to by x.*
$\mid\ y := new_T()$	*Allocate a new T-cell and assign its address to y.*

A *path* π (of length n) in the CFG is a sequence $\langle \ell_0, \alpha_1, \ell_1, \ldots, \alpha_n, \ell_n \rangle$ alternating between locations ℓ_i and actions α_j. Note that there is no action for procedure calls, yet in bounded model checking, calls can be handled by inlining procedure bodies.

Figure 1 shows a sample program. Upon start it expects that the variable `e` points to a doubly linked circular list (realized by `next`- and `prev`-pointers). The program deallocates the cell pointed to by `e`, allocates a new cell and inserts it in place of the old one (using the temporary variables `ne` and `pe`).

2.2 Semantics

Given a program, we will provide a transition system semantics, i. e., a directed graph whose vertices are states and whose edges are transitions. Informally, a state is the contents of the program variables, i. e., an assignment of the program variables to values, and the contents of the heap, i. e., an assignment of heap addresses to values; hereby, a value may again be a heap address. We represent both assignments by means of a relational first-order structure with constants. The interpretations of constants make up the contents of the program variables, whereas the interpretations of the relations make up the contents of the heap. More precisely, each field r of a template T is interpreted by a functional binary relation, so $r(u, v)$ means that an instance of T lives at address u in the heap, and v is the uniquely

$$\begin{aligned}
\llbracket \mathit{assume}(\gamma) \rrbracket &\equiv \gamma \wedge \textstyle\bigwedge_{c\in\bar{c}} c' \approx c \wedge \bigwedge_{r\in\bar{r}} r' = r\\
\llbracket y := e \rrbracket &\equiv y' \approx e \wedge \textstyle\bigwedge_{c\in\bar{c}\setminus\{y\}} c' \approx c \wedge \bigwedge_{r\in\bar{r}} r' = r\\
\llbracket y := s(x) \rrbracket &\equiv s(x,y') \wedge \textstyle\bigwedge_{c\in\bar{c}\setminus\{y\}} c' \approx c \wedge \bigwedge_{r\in\bar{r}} r' = r\\
\llbracket s(x) := e \rrbracket &\equiv \forall u,v\big(s'(u,v) \Leftrightarrow u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v)\big) \wedge\\
&\quad\ \textstyle\bigwedge_{c\in\bar{c}} c' \approx c \wedge \bigwedge_{r\in\bar{r}\setminus\{s\}} r' = r\\
\llbracket \mathit{free}_T(x) \rrbracket &\equiv \textstyle\bigwedge_{s\in T} \forall u,v\big(s'(u,v) \Leftrightarrow u \not\approx x \wedge s(u,v)\big) \wedge\\
&\quad\ \textstyle\bigwedge_{c\in\bar{c}} c' \approx c \wedge \bigwedge_{r\in\bar{r}\setminus T} r' = r\\
\llbracket y := \mathit{new}_T() \rrbracket &\equiv y' \not\approx \mathit{NULL} \wedge \textstyle\bigwedge_{s\in T} \forall u,v(s(u,v) \Rightarrow u \not\approx y') \wedge\\
&\quad\ \textstyle\bigwedge_{s\in T} \forall u,v\big(s'(u,v) \Leftrightarrow u \approx y' \wedge v \approx \mathit{NULL} \vee s(u,v)\big) \wedge\\
&\quad\ \textstyle\bigwedge_{c\in\bar{c}\setminus\{y\}} c' \approx c \wedge \bigwedge_{r\in\bar{r}\setminus T} r' = r
\end{aligned}$$

Figure 2: Semantics of actions. Note that the second-order equalities $r' = r$ are to be understood as abbreviations for first-order formulas $\forall u, v\big(r'(u,v) \Leftrightarrow r(u,v)\big)$.

defined value of the r-field of that instance. Note that the universe of the first-order structures can be assumed finite since the number of heap addresses is finite (yet unbounded).

Formally, we associate a *vocabulary* σ to a program P, where $\sigma = \langle \bar{r}, \bar{c} \rangle$ declares a set of binary relation symbols $\bar{r}$ and a set of constants $\bar{c}$. Hereby, $\bar{c}$ is the set of program variables occurring in the control flow graph of P, and $\bar{r}$ is the set of fields occurring in the struct declaration of P. A *σ-structure* $\mathbf{A}$ is a tuple $\langle A, \bar{r}^{\mathbf{A}}, \bar{c}^{\mathbf{A}} \rangle$, where A is a non-empty universe, $\bar{r}^{\mathbf{A}} = \{r^{\mathbf{A}} \mid r \in \bar{r}\}$ is a set of binary relations on A interpreting the symbols in $\bar{r}$, and $\bar{c}^{\mathbf{A}} = \{c^{\mathbf{A}} \mid c \in \bar{c}\}$ is a set of elements of A interpreting the constants in $\bar{c}$. A *state* of the program P is a pair $\langle \ell, \mathbf{A} \rangle$ consisting of a location ℓ and a σ-structure $\mathbf{A}$. We require that the universe A is finite and that all relations $r^{\mathbf{A}}$ are functional, i. e., for all $a, b, b' \in A$, if $r^{\mathbf{A}}(a,b)$ and $r^{\mathbf{A}}(a,b')$ then $b = b'$. Additionally, we require that the interpretation of NULL is a dangling pointer, i. e., $r^{\mathbf{A}}(\mathit{NULL}^{\mathbf{A}}, b)$ is false for all $b \in A$ and all relations $r^{\mathbf{A}}$.

Transitions are certain pairs of states. For specifying which pairs, we have to extend the vocabulary σ. We define $\sigma' = \langle \bar{r}', \bar{c}' \rangle$ to be a copy of σ, where $\bar{r}' = \{r' \mid r \in \bar{r}\}$ and $\bar{c}' = \{c' \mid c \in \bar{c}\}$. By $\sigma + \sigma'$, we denote the union of the vocabularies σ and σ'. Given a σ-structure $\mathbf{A}$, we denote the corresponding σ'-structure (with universe A) by $\mathbf{A}'$. Thus, given σ-structures $\mathbf{A}$ and $\mathbf{B}$ with $A = B$, $\mathbf{A} + \mathbf{B}'$ can be viewed as a $(\sigma + \sigma')$-structure. Likewise, a $(\sigma + \sigma')$-formula can be viewed as defining a binary relation on σ-structures. Figure 2 specifies the semantics of actions α as $(\sigma + \sigma')$-formulas $\llbracket \alpha \rrbracket$.

A *transition* of the program P is pair $\big\langle \langle \ell, \mathbf{A} \rangle, \langle \ell', \mathbf{B} \rangle \big\rangle$ of states such that $A = B$ and $\mathbf{A} + \mathbf{B}' \models \llbracket \alpha \rrbracket$, where the action α is the label of some edge from ℓ to ℓ' in the CFG of P. We call $\langle \ell, \mathbf{A} \rangle$ the *pre-state* and $\langle \ell', \mathbf{B} \rangle$ the *post-state* of the transition. Note that $\llbracket y := s(x) \rrbracket$ is false if x is dangling, i. e., the semantics models read dereferences of dangling pointers by deadlock. On the other hand, $\llbracket y := \mathit{new}_T() \rrbracket$

defines a total relation, so allocation never fails due to lack of memory.[1]

Given a path $\pi = \langle \ell_0, \alpha_1, \ell_1, \ldots, \alpha_n, \ell_n \rangle$ in the CFG, we call a sequence of states $\big\langle \langle \ell_0, \mathbf{A}_0 \rangle, \langle \ell_1, \mathbf{A}_1 \rangle, \ldots, \langle \ell_n, \mathbf{A}_n \rangle \big\rangle$ a π-*execution* (of length n) if for $1 \leq i \leq n$, $\big\langle \langle \ell_{i-1}, \mathbf{A}_{i-1} \rangle, \langle \ell_i, \mathbf{A}_i \rangle \big\rangle$ is a transition. We call ℓ_n π-*reachable* from $\langle \ell_0, \mathbf{A}_0 \rangle$ if there is a π-execution $\big\langle \langle \ell_0, \mathbf{A}_0 \rangle, \ldots, \langle \ell_n, \mathbf{A}_n \rangle \big\rangle$. In general, we call a location ℓ' *reachable* from a state $\langle \ell, \mathbf{A} \rangle$ if there is a path π such that ℓ' is π-reachable from $\langle \ell, \mathbf{A} \rangle$.

2.3 Error Conditions

Run time errors in pointer programs occur when dereferencing dangling pointers or NULL pointers, or when freeing memory that is not allocated. We can check for such errors by introducing error conditions into the control flow graph just before the dangerous actions read, write and deallocate. As expressing the error conditions requires quantifiers, we have to allow more complex conditions in assume-actions.

Let P be a program and let σ be its associated vocabulary. We say that a σ-formula φ (of first-order logic with equality) is in the *Bernays-Schönfinkel* class [3, 20] with n variables[2], denoted by $\varphi \in BS_n$, if φ is equivalent to a sentence $\exists u_1, \ldots, u_m \forall v_1, \ldots, v_n \, \psi$, where ψ is quantifier-free. For expressing error conditions, we admit actions $assume(\gamma)$ where $\gamma \in BS_n$ for $n \geq 0$.

We extend the CFG of the program P to a *CFG with error conditions (ECFG)* in the following way. We introduce a new distinguished location $error$. For every edge from ℓ to ℓ' that is labeled by $y := s(x)$ or $s(x) := e$, we add an edge from ℓ to $error$ labeled by $assume(\forall v \neg s(x, v))$. And for every edge from ℓ to ℓ' labeled by $free_T(x)$ and every $r \in T$, we add an edge from ℓ to $error$ labeled by $assume(\forall v \neg r(x, v))$. Note that the error condition $\forall v \neg s(x, v)$ for the read- and write-actions is true if and only if the pointer x is dangling, i. e., there is no value for the s-field at address x. This condition also captures $NULL$ dereferences since we assume the special address $NULL$ to be a dangling pointer. For deallocation, the error conditions $\forall v \neg r(x, v)$ are true if and only if there is no instance of T at address x, e. g., because it has been deallocated earlier. Note that all error conditions are in BS_1.

2.4 Weakest Preconditions

Let P be a program and let σ be the associated vocabulary. Given an action α and a σ-formula φ, informally the weakest precondition of φ w. r. t. α captures those states which upon execution of α may lead to a state satisfying φ. Formally, the *weakest precondition* $pre(\alpha; \varphi)$ is defined as

$$pre(\alpha; \varphi) \equiv \exists \bar{r}' \exists \bar{c}' \big([\![\alpha]\!] \wedge \varphi[\bar{r}'/\bar{r}][\bar{c}'/\bar{c}] \big) \ .$$

[1]A slightly modified version of $[\![y := new_T()]\!]$ models failed allocation by returning $NULL$, see appendix B.

[2]We count only universally quantified variables; the other variables can be viewed as constants.

Here, $\varphi[\bar{r}'/\bar{r}][\bar{c}'/\bar{c}]$ is an abbreviation for $\varphi[r_1'/r_1,\dots,r_m'/r_m][c_1'/c_1,\dots,c_n'/c_n]$ where $\bar{r}=\{r_1,\dots,r_m\}$ and $\bar{c}=\{c_1,\dots,c_n\}$, i. e., $\varphi[\bar{r}'/\bar{r}][\bar{c}'/\bar{c}]$ is the formula obtained from φ by replacing every relation symbol r by r' and every constant c by c'. Further, $\exists\bar{r}'\exists\bar{c}'$ denotes the existential quantification of all relation symbols r' and all constants[3] c'. Note that $\mathit{pre}(\alpha;\varphi)$ is a second-order formula due to quantification over relations. Given a path π in the CFG of P and a σ-formula φ, we define the weakest precondition $\mathit{pre}(\pi;\varphi)$ of φ w. r. t. π in the usual way by induction on the length of π.

Depending on the actions α, we can rewrite the second-order formula $\mathit{pre}(\alpha;\varphi)$ to an equivalent first-order formula. We need to extended our notion of substitution to allow for the substitution of atomic formulas. Given two formulas φ and ψ and an atomic formula $r(u_1,u_2)$ with free variables u_i, we write $\varphi[\psi/r(u_1,u_2)]$ for the formula obtained from φ by replacing every atomic formula $r(t_1,t_2)$ by the formula $\psi[t_1/u_1,t_2/u_2]$, where the t_i are arbitrary terms. Note that the variables u_i just function as parameters for the terms t_i.

Lemma 1. *Given a σ-formula φ and an action α, we have the following characterization of $\mathit{pre}(\alpha;\varphi)$, where T is a template, $\bar{s}=\{s_1,\dots,s_n\}$ are the fields of T, s is an arbitrary field, x and y are program variables, e is a program variable or constant (including NULL), and γ is a σ-formula.*

$$\begin{aligned}
\mathit{pre}(\mathit{assume}(\gamma);\varphi) &\equiv \gamma\wedge\varphi\\
\mathit{pre}(y:=e;\varphi) &\equiv \varphi[e/y]\\
\mathit{pre}(y:=s(x);\varphi) &\equiv \exists y'\big(s(x,y')\wedge\varphi[y'/y]\big)\\
\mathit{pre}(s(x):=e;\varphi) &\equiv \varphi[u\approx x\wedge v\approx e\vee u\not\approx x\wedge s(u,v)/s(u,v)]\\
\mathit{pre}(\mathit{free}_T(x);\varphi) &\equiv \varphi[u\not\approx x\wedge \bar{s}(u,v)/\bar{s}(u,v)]^4\\
\mathit{pre}(y:=\mathit{new}_T();\varphi) &\equiv \exists y'\big(y'\not\approx \mathit{NULL}\wedge \textstyle\bigwedge_{s\in\bar{s}}\forall u,v(s(u,v)\Rightarrow u\not\approx y')\wedge\\
&\qquad \varphi[y'/y][u\approx y'\wedge v\approx \mathit{NULL}\vee\bar{s}(u,v)/\bar{s}(u,v)]\big)
\end{aligned}$$

Proof. See appendix A. □

Lemma 2. *Let P be a program, π a path in the ECFG and φ a σ-formula. If $\varphi\in BS_n$, $n\geq 2$, then $\mathit{pre}(\pi;\varphi)\in BS_n$. Moreover, the size of $\mathit{pre}(\pi;\varphi)$ is in $O(|\pi|^2\cdot|\varphi|)$, and the length of the quantifier prefix of $\mathit{pre}(\pi;\varphi)$ is in $O(|\pi|+|\varphi|)$.*

Proof. Follows from Lemma 1 once we note that all conditions are in BS_1 and the class BS_n is closed under outermost existential quantification, under conjunction, under substitution of terms for variables and under substitution of quantifier-free formulas for atomic ones. Note that the restriction $n\geq 2$ is enforced by the weakest preconditions for allocation actions.

It is easy to see the quantifier prefix grows linearly in the length of π. Note that the number of non-equality atoms (which is increased only by read access and

[3]We quantify over constants as if they were free variables.

[4]We consider a formula like $\varphi[u\not\approx x\wedge\bar{s}(u,v)/\bar{s}(u,v)]$ to be an abbreviation for the formula $\varphi[u\not\approx x\wedge s_1(u,v)/s_1(u,v)]\dots[u\not\approx x\wedge s_n(u,v)/s_n(u,v)]$.

allocation) also grows linearly. However, in each step, substituting $O(|\pi|)$ non-equality atoms may increase the number of equality atoms by $O(|\pi|)$, so the total growth is quadratic. $\square$

2.5 Initial Conditions

Using weakest preconditions, we can propagate an error condition backwards along a given error path $\pi = \langle \mathit{init}, \ldots, \mathit{error} \rangle$ and obtain a condition $\mathit{pre}(\pi; \top)$ expressing precisely when the error location is π-reachable from init. Due to Lemma 2 and the decidability of the Bernays-Schönfinkel class, we can thus decide whether there is some state $\langle \mathit{init}, \mathbf{A} \rangle$ from which error is π-reachable. However, there may be many such states which are irrelevant since they do not satisfy certain initial conditions that we want to impose, e. g., that program variable `e` points to a doubly linked circular list. Such properties are not expressible in first-order logic, however, list or tree structures, even certain circular ones, can be specified by logic programs. We will express initial conditions as conjunctions $\phi \wedge \mathcal{P} \wedge Q$, where ϕ is a σ-formula in the Bernays-Schönfinkel class with 2 variables, $\mathcal{P}$ is a (restricted) monadic Datalog program and Q is a (ground) query. The use of Datalog allows us, without losing decidability, to express initial data structures of arbitrary size (expressing reachability of all memory cells in such structures usually requires the use of some kind of transitive closure, which often leads to undecidability.) The Datalog program $\mathcal{P}$ will be interpreted over models of ϕ and will extend them with unary relations, hence the *extensional database (EDB)* vocabulary is $\sigma = \langle \bar{r}, \bar{c} \rangle$, and the *intensional database (IDB)* vocabulary $\sigma_I = \langle \bar{p} \rangle$ declares a set of unary predicates $\bar{p}$.

A *monadic Datalog program* $\mathcal{P}$ is a finite set of clauses $A_0 \leftarrow A_1, \ldots, A_k$, $k \geq 0$, where the *head* A_0 is a unary IDB atom and the *body* $A_1, \ldots, A_k$ is a conjunction of IDB atoms and EDB literals (i. e., possibly negated EDB atoms). A *query* Q is a conjunction $A_1, \ldots, A_k$, $k \geq 0$, of ground IDB atoms (i. e., IDB atoms of the form $p(c)$). Note that we consider the order of the atoms in queries and clause bodies irrelevant. In section 3, we will impose further restrictions on monadic Datalog programs in order to prove a decidability result.

Let $\mathcal{P}$ be a monadic Datalog program. Given an EDB (i. e., a σ-structure) $\mathbf{A}$, the *least model* of $\mathcal{P}$ over $\mathbf{A}$ is the least extension of $\mathbf{A}$ to a $(\sigma + \sigma_I)$-structure $\mathbf{B}$ such that $\mathbf{B}$ is a model of the clause set $\mathcal{P}$. Given a σ-formula ϕ and a query Q, we say that $\mathbf{A}$ *satisfies* $\phi \wedge \mathcal{P} \wedge Q$, denoted by $\mathbf{A} \models \phi \wedge \mathcal{P} \wedge Q$, if $\mathbf{A}$ is a model of ϕ and the least model of $\mathcal{P}$ over $\mathbf{A}$ is a model of Q. In section 3, we will show that satisfiability of $\phi \wedge \mathcal{P} \wedge Q$ is decidable, provided that ϕ is in BS_2 and $\mathcal{P}$ is a restricted monadic Datalog program.

Figure 1 shows the initial condition $\phi \wedge \mathcal{P} \wedge$ `cl(e)` for a program operating on a circular doubly linked list. The monadic Datalog program $\mathcal{P}$ together with the query `cl(e)` ensure that `e` points to a circular list linked via the `next`-fields. The axiom ϕ ensures that the binary relation `prev` is the converse of `next`, so the circular list is doubly linked. Furthermore, ϕ ensures that the address NULL

is a dangling pointer. Note that ϕ can not ensure functionality of `next` and `prev`, because functionality is not expressible in BS_2. Therefore, we need a semantic restriction on σ-structures, i. e., we will only consider σ-structures where all binary relations are functional.

2.6 The Bounded Model Checking Problem

Let P be a program. Informally, we call P pointer-safe if there is no initial state from which execution of P can result in a runtime error by dereferencing $NULL$ or a dangling pointer. To define pointer-safety formally, we assume that the initial conditions of P are given as a conjunction $\phi \wedge \mathcal{P} \wedge Q$, where ϕ is a BS_2-formula, $\mathcal{P}$ a monadic Datalog program and Q a query. We call P *pointer-safe* if for all states $\langle \mathit{init}, \mathbf{A}\rangle$ such that $\mathbf{A} \models \phi \wedge \mathcal{P} \wedge Q$, the location error is unreachable from $\langle \mathit{init}, \mathbf{A}\rangle$ in the ECFG. In bounded model checking, we do not solve the full reachability problem but restrict to paths of an a priori bounded length. Since there are only finitely many such bounded paths in the ECFG, for showing decidability of the bounded model checking problem it suffices to restrict to one path. We call an ECFG-path $\pi = \langle \mathit{init}, \ldots, \mathit{error}\rangle$ *pointer-safe* if for all $\mathbf{A}$ with $\mathbf{A} \models \phi \wedge \mathcal{P} \wedge Q$, error is not π-reachable from $\langle \mathit{init}, \mathbf{A}\rangle$.

Theorem 3. *Let P be a program with initial condition $\phi \wedge \mathcal{P} \wedge Q$, where ϕ is a BS_2-formula, $\mathcal{P}$ a Datalog program and Q a query. Let $\pi = \langle \mathit{init}, \ldots, \mathit{error}\rangle$ be an ECFG-path of P. It is decidable whether π is pointer-safe.*

Proof. The path π is pointer-safe if and only if $\mathit{pre}(\pi; \top) \wedge \phi \wedge \mathcal{P} \wedge Q$ is unsatisfiable. As $\mathit{pre}(\pi; \top) \wedge \phi \in BS_2$ by assumption and Lemma 2, the decidability follows from Theorem 8 in section 3. ☐

For the complexity of bounded model checking, we refer to section 3.3.

It turns out that the program from figure 1 is not pointer-safe. Consider the path $\pi = \langle \mathit{init}, \mathtt{ne} := \mathtt{next}(\mathtt{e}), \ell_1, \ldots, \ell_6, \mathit{assume}(\forall v\, \neg\, \mathtt{next}(\mathtt{pe}, v)), \mathit{error}\rangle$. Since the formula $\mathit{pre}(\pi; \top) \wedge \phi \wedge \mathcal{P} \wedge Q$ is satisfiable, the program can crash when executing the last but one action $\mathtt{next}(\mathtt{pe}) := \mathtt{e}$. An analysis of the models of $\mathit{pre}(\pi; \top) \wedge \phi \wedge \mathcal{P} \wedge Q$ reveals the reason: If `e` points to a circular list of length 1 then $\mathtt{pe} \approx \mathtt{e}$ after the second action, so `pe` is dangling after $\mathit{free}_{\mathtt{c1}}(\mathtt{e})$.

3 Deciding the Bernays-Schönfinkel Class with Datalog

In this section, we develop our main result, the decidability of the 2-variable fragment of the Bernays-Schönfinkel class extended by a class of monadic Datalog programs.

3.1 Syntax and Semantics of Bernays-Schönfinkel with Datalog

We are interested in satisfiability of formulas of the form $\phi \wedge \mathcal{P} \wedge Q$, where

- ϕ is a universal σ-formula in BS_2 (see section 2.3), i. e., ϕ is of the form $\forall u, v\, \psi$ with ψ quantifier-free,[5]

- $\mathcal{P}$ is a monadic Datalog program with EDB vocabulary σ and IDB vocabulary σ_I (see section 2.5), and

- Q a query (see section 2.5).

We are not interested in general satisfiability of $\phi \wedge \mathcal{P} \wedge Q$ but we impose two additional restrictions on models $\mathbf{A}$. One restriction is motivated by the fact that we model pointer structures (see section 2.2), the other is used in our decidability proof.

Functionality. We require that all binary relations are functional, i. e., for all EDB predicates r, $\mathbf{A}$ must be a model of $\forall u, v_1, v_2 \big(r(u, v_1) \wedge r(u, v_2) \Rightarrow v_1 {\approx} v_2\big)$. This ensures that $\mathbf{A}$ represents a pointer structure, which is functional graph since every pointer at any given moment points to at most one heap cell.

Non-Sharing. We require that the binary relations occurring in $\mathcal{P}$ represent pointers in data structures that do not share memory with other data structures defined by $\mathcal{P}$. That is, $\mathbf{A}$ must be a model of all sentences of the form $\forall u_1, u_2, v \big(s_1(u_1, v) \wedge s_1(u_2, v) \wedge u_1 \not\approx u_2 \Rightarrow \mathit{const}(v)\big)$ and $\forall u_1, u_2, v \big(s_1(u_1, v) \wedge s_2(u_2, v) \Rightarrow \mathit{const}(v)\big)$, where s_1 and s_2 are two distinct EDB predicates occurring in $\mathcal{P}$ and $\mathit{const}(v)$ is a shorthand for the disjunction $\bigvee_{c \in \bar{c}} v \approx c$. Note that the non-sharing restriction is not imposed on all binary predicates but just on the ones occurring in the Datalog program $\mathcal{P}$.

Obviously, the functionality and non-sharing restrictions are expressible in the Bernays-Schönfinkel class with equality. However, they require more than two variables, so we cannot add them to the formula ϕ but must deal with them on the semantic level.

Besides restrictions on the class of models, we need to impose two restrictions on monadic Datalog programs $\mathcal{P}$. We call $\mathcal{P}$ *tree-automaton-like* (*TA-like* for short) if all clauses are of the form

$$p(u) \leftarrow B_1, \ldots, B_l, r_1(u, v_1), q_1(v_1), \ldots, r_k(u, v_k), q_k(v_k) \tag{1}$$

for some $k, l \geq 0$, where $u, v_1, \ldots, v_k$ are $k + 1$ distinct variables, $r_1, \ldots, r_k$ are k distinct relation symbols, and the B_i are EDB literals containing at most the variable u. We define the *degree* of a clause of the form (1) to be the natural number k, i. e., the number of IDB atoms in the body. The *degree* of a TA-like monadic Datalog program is the maximal degree of its clauses. We call $\mathcal{P}$ *intersection-free* if for all EDBs $\mathbf{A}$, the least model of $\mathcal{P}$ over $\mathbf{A}$ satisfies all sentences of the form $\forall v \big(p(v) \wedge q(v) \Rightarrow \mathit{const}(v)\big)$, where p and q are two distinct IDB predicate symbols. Note that the EDBs are σ-structures satisfying the above functionality and non-sharing restrictions. Viewing the IDB predicates as shape types for heap

[5] We handle non-universal formulas $\phi = \exists z_1, \ldots, z_n \forall x, y\, \psi \in BS_2$ by extending the vocabulary $\sigma = \langle \bar{r}, \bar{c} \rangle$, adding the variables z_i as constants.

$$\begin{aligned}
\mathtt{list}(u) &\leftarrow u \approx \mathit{NULL}.\\
\mathtt{list}(u) &\leftarrow \mathtt{next}(u,v), \mathtt{list}(v).\\[4pt]
\mathtt{dll}(u) &\leftarrow u \approx \mathtt{lst}, \mathtt{next}(u, \mathit{NULL}).\\
\mathtt{dll}(u) &\leftarrow \mathtt{next}(u,v), \mathtt{dll}(v).\\[4pt]
\mathtt{ring}(u) &\leftarrow u \approx \mathtt{p}, \mathtt{next}(u,v), \mathtt{ring}'(v).\\
\mathtt{ring}'(u) &\leftarrow u \approx \mathtt{p}.\\
\mathtt{ring}'(u) &\leftarrow \mathtt{next}(u,v), \mathtt{ring}'(v).
\end{aligned}
\qquad
\begin{aligned}
\mathtt{tree}(u) &\leftarrow u \approx \mathit{NULL}.\\
\mathtt{tree}(u) &\leftarrow \mathtt{left}(u,v), \mathtt{right}(u,w),\\
&\qquad \mathtt{tree}(v), \mathtt{tree}(w).\\[4pt]
\mathtt{gtree}(u) &\leftarrow u \approx \mathit{NULL}.\\
\mathtt{gtree}(u) &\leftarrow \mathtt{sons}(u,v), \mathtt{gtrees}(v).\\
\mathtt{gtrees}(u) &\leftarrow u \approx \mathit{NULL}.\\
\mathtt{gtrees}(u) &\leftarrow \mathtt{tree}(u,v), \mathtt{gtree}(v),\\
&\qquad \mathtt{next}(u,w), \mathtt{gtrees}(w).
\end{aligned}$$

$$\mathtt{prev}(\mathtt{fst}, \mathit{NULL}) \wedge \forall u,v\big(u \approx \mathit{NULL} \vee v \approx \mathit{NULL} \vee (\mathtt{prev}(u,v) \Leftrightarrow \mathtt{next}(v,u))\big)$$
$$\forall u,v\big(u \approx \mathit{NULL} \vee v \approx \mathit{NULL} \vee (\mathtt{up}(u,v) \Leftrightarrow \mathtt{left}(v,u) \vee \mathtt{right}(v,u))\big)$$

Figure 3: Datalog programs $\mathcal{P}$ and axioms ϕ representing initial conditions.

cells, an intersection-free Datalog program associates to most heap cells only one shape type; i. e., there is no intersection of shape types (except for cells pointed to by program variables).

Besides the model-theoretic semantics from section 2.5 there is a proof theoretic semantics for Datalog programs. For simplicity, we will define the proof-theoretic semantics only for TA-like monadic Datalog programs $\mathcal{P}$. Let $\mathbf{A}$ be an EDB, i. e., a σ-structure. A *fact* $p(a)$ consists of an IDB predicate symbol p and an element $a \in A$. We say that a list of facts $q_1(a_1), \dots, q_k(a_k)$, $k \geq 0$, *produces* a fact $p(a)$ (w. r. t. $\mathcal{P}$ and $\mathbf{A}$) if $\mathcal{P}$ contains a clause

$$p(u) \leftarrow B_1, \dots, B_l, r_1(u, v_1), q_1(v_1), \dots, r_k(u, v_k), q_k(v_k)$$

such that $\mathbf{A}$ satisfies all literals $r_i(u, v_i)$ and B_j when interpreting the variables $u, v_1, \dots, v_k$ by $a, a_1, \dots, a_k$, respectively. A *proof tree* T for $\mathcal{P}$ w. r. t. $\mathbf{A}$ is an ordered tree where each node is labeled by a fact. Depending on the situation, we call a node n which is labeled by $p(a)$ an *a-node*, a *p-node* or a *$p(a)$-node*. For each $p(a)$-node n in T with k sons $n_1, \dots, n_k$, $k \geq 0$, labeled by $q_i(a_i)$, we require that the list of facts $q_1(a_1), \dots, q_k(a_k)$ produces the fact $p(a)$ w. r. t. $\mathcal{P}$ and $\mathbf{A}$. The proof tree T *proves* a ground IDB atom $p(c)$ if its root is labeled by the fact $p(c^{\mathbf{A}})$. The proof- resp. model-theoretic semantics are linked in that the least model of $\mathcal{P}$ over $\mathbf{A}$ satisfies a ground IDB atom $p(c)$ if and only if $p(c)$ is proven by some proof tree T. Note that for all facts $p(a)$, we can w. l. o. g. assume that all $p(a)$-subtrees of a proof tree (i. e., all subtrees rooted at $p(a)$-nodes) are isomorphic.

Figure 3 shows examples of monadic Datalog programs $\mathcal{P}$ and BS_2 axioms ϕ that represent several initial conditions. The initial condition "the program variable p points to a list" can be expressed by the formula $\mathcal{P} \wedge \mathtt{list}(\mathtt{p})$ where $\mathcal{P}$ is the first of the programs on figure 3. The condition "the program variables `fst` and `lst` point to the first and the last elements of a doubly linked list" can be expressed by the formula $\phi \wedge \mathcal{P} \wedge \mathtt{dll}(\mathtt{fst})$ where $\mathcal{P}$ is the second program and ϕ is the first of the axioms. Here, an atom $\mathtt{dll}(u)$ expresses that u points to a doubly linked list whose last element is pointed by `lst`; note that `lst` (unlike u) is a logical constant. The condition "p points to a binary tree" can be expressed by $\mathcal{P} \wedge \mathtt{tree}(\mathtt{p})$. In the same way as for doubly linked lists, one may add an axiom (the second one on figure 3) defining a predicate up to obtain a doubly linked binary tree. The condition "p points to a singly linked circular list" can be expressed by $\mathcal{P} \wedge \mathtt{ring}(\mathtt{p})$ where the Datalog program $\mathcal{P}$ defines two IDB predicates ring and $\mathtt{ring}'$. The corresponding doubly linked circular list can also be defined, as was shown in figure 1. In the last example, general (i. e., arbitrarily branching) singly linked trees can be handled by representing them as trees of lists, i. e., every tree node points (via sons) to a list of sons (singly linked via next), each node of which points to a tree node (via tree). The condition "p points to a arbitrarily branching tree" can be expressed by $\mathcal{P} \wedge \mathtt{gtree}(\mathtt{p})$.

Note that all these monadic Datalog programs are TA-like and intersection-free, even if they are appear together in one initial condition (provided that the predicate next is renamed to list_next, dll_next, ring_next and gtrees_next, respectively).

3.2 Decidability of Bernays-Schönfinkel with Datalog

In this section we prove that satisfiability of formulas of the form $\phi \wedge \mathcal{P} \wedge Q$ is decidable, where ϕ is a universal σ-formula in BS_2, $\mathcal{P}$ is a TA-like intersection-free monadic Datalog program and Q is a query. This is done by showing the small model property for these formulas: Every satisfiable formula has a model of size bounded by a function depending only on the formula. Before we prove this we recall some definitions and lemmas.

Finite model property. The proof of the following well-known lemma can be found in [7], for instance.

Lemma 4. *Let ϕ be a universal σ-formula in the Bernays-Schönfinkel class with equality. If $\mathbf{A} \models \phi$ and $\mathbf{B}$ is obtained from $\mathbf{A}$ by removing from its universe any number of elements not interpreting constants then $\mathbf{B} \models \phi$.*

The above lemma immediately gives us a finite model property for formulas of the form $\phi \wedge \mathcal{P} \wedge Q$ with $Q = q_1(c_1), \dots, q_k(c_k)$. It is enough to take any model $\mathbf{A}$ of ϕ and any k proof trees T_i proving $q_i(c_i)$ and restrict $\mathbf{A}$ to the interpretations of constants and to the elements that occur in the proof trees T_i.

Corollary 5. *Every satisfiable formula of the form $\phi \wedge \mathcal{P} \wedge Q$ has a finite model.*

Types. Recall the EDB vocabulary $\sigma = \langle \bar{r}, \bar{c} \rangle$. Let u and v be variables. A 1-atom $\alpha(u)$ is an atomic σ-formula containing at most the variable u, i. e., $\alpha(u)$ is a ground atom or it is of the form $u \approx u$, $u \approx c$, $r(u,u)$, $r(u,c)$ or $r(c,u)$ for $c \in \bar{c}$ and $r \in \bar{r}$; likewise we define 1-atoms $\alpha(v)$. A 2-atom $\alpha(u,v)$ is an atomic σ-formula containing at most the variables u and v, i. e., $\alpha(u,v)$ is a 1-atom or it is of the form $u \approx v$, $r(u,v)$ or $r(v,u)$ for $r \in \bar{r}$. A 1-literal (resp. 2-literal) is a possibly negated 1-atom (resp. 2-atom). A 1-type $\tau(u)$ (resp. $\tau(v)$) is a maximal propositionally consistent conjunction of 1-literals, i. e., all possible 1-atoms $\alpha(u)$

(resp. $\alpha(v)$) occur exactly once in the conjunction $\tau(u)$ (resp. $\tau(v)$). Similarly, a 2-type $\tau(u, v)$ is a maximal propositionally consistent conjunction of 2-literals. Note that there are only finitely many different types: the number of 1-types is $2^{(|\bar{c}|+1)^2(|\bar{r}|+1)}$ and the number of 2-types is $2^{(|\bar{c}|+2)^2(|\bar{r}|+1)}$.

In a given σ-structure $\mathbf{A}$, for every element $a \in A$ there is exactly one 1-type $\tau(u)$ that is true when one assigns a to the variable u; we denote this type $\tau(u)$ by $\tau_{\mathbf{A}}(a)$. Likewise, for every two elements $a, b \in A$ there is exactly one 2-type $\tau(u, v)$, denoted by $\tau_{\mathbf{A}}(a, b)$, that is true when one assigns a to u and b to v. Note that for $a, a', b, b' \in A$, the 2-type equality $\tau_{\mathbf{A}}(a, b) = \tau_{\mathbf{A}}(a', b')$ implies the 1-type equalities $\tau_{\mathbf{A}}(a) = \tau_{\mathbf{A}}(a')$ and $\tau_{\mathbf{A}}(b) = \tau_{\mathbf{A}}(b')$. A type $\tau(u)$ resp. $\tau(u, v)$ is *inhabited* in $\mathbf{A}$ if we have $\tau(u) = \tau_{\mathbf{A}}(a)$ resp. $\tau(u, v) = \tau_{\mathbf{A}}(a, b)$ for some $a, b \in A$. In general, not all types are inhabited in a fixed σ-structure $\mathbf{A}$, e. g., if $\mathbf{A}$ interprets the constants c_1 and c_2 by different elements, no type containing the conjuncts $u \approx c_1$ and $u \approx c_2$ can be inhabited. Therefore, $(|\bar{c}| + 1) \cdot 2^{(|\bar{c}|+1)^2|\bar{r}|}$ resp. $(|\bar{c}|^2 + 2|\bar{c}| + 2) \cdot 2^{(|\bar{c}|+2)^2|\bar{r}|}$ is an upper bound on the number of inhabited 1- resp. 2-types.

In bounded branching structures, we can get a tighter bound on the number of inhabited types. We say that a σ-structure $\mathbf{A}$ is *k-branching*, $k \geq 0$, if for all $r \in \bar{r}$, every $a \in A$ which is not interpreting a constant has at most k predecessors and k successors w. r. t. the relation $r^{\mathbf{A}}$. If $\mathbf{A}$ is k-branching then the number of inhabited 1-types is bounded by $|\bar{c}| + (((|\bar{c}| + 1)^{2k} + 1)^{|\bar{r}|}$, because there are $|\bar{c}|$ types inhabited by elements interpreting constants, and for the types inhabited by the other elements each binary predicate r contributes at most $(|\bar{c}|+1)^k$ possibilities due to predecessors, $(|\bar{c}| + 1)^k$ due to successors and one due to the self loop. Similarly, the number of inhabited 2-types is bounded by $|\bar{c}|^2 + 2|\bar{c}|(((|\bar{c}| + 1)^{2k} + 3)^{|\bar{r}|} + (((|\bar{c}| + 1)^{4k} + 4)^{|\bar{r}|}$.

Lemma 6. *The number of 2-types inhabited in a σ-structure $\mathbf{A}$ is bounded by a function singly exponential in the size of the vocabulary σ. If $\mathbf{A}$ is k-branching, $k \geq 0$, then the bound on the number of inhabited 2-types is exponential in k and in the number of binary relations but polynomial in the number of constants.*

Pumping Lemma. The core of our decidability result is the following technical lemma which "pumps down" big proof trees by compressing long paths, thus bounding the depth of proof trees. With the bound, the small model theorem follows easily. Recall the EDB vocabulary $\sigma = \langle \bar{r}, \bar{c} \rangle$ and the IDB vocabulary $\sigma_I = \langle \bar{p} \rangle$.

Lemma 7. *If a formula of the form $\phi \wedge \mathcal{P} \wedge Q$ is satisfiable then it is satisfiable by a σ-structure $\mathbf{A}$ such that all IDB atoms $q(c)$ in the query Q have proof trees of depth bounded by*

$$|\bar{p}||\bar{c}| + |\bar{p}||\bar{c}| \cdot (\theta \cdot |\bar{p}| \cdot \delta + 1), \tag{2}$$

where δ is the degree of $\mathcal{P}$, and θ is the maximal number of inhabited 2-types in models of $\phi \wedge \mathcal{P} \wedge Q$.

Proof. Let $\mathbf{A}$ be a model of the formula $\phi \wedge \mathcal{P} \wedge Q$, i. e., $\mathbf{A}$ is a model of ϕ and there is a finite set $\mathcal{T}$ of proof trees for $\mathcal{P}$ w. r. t. $\mathbf{A}$ such that each tree $T \in \mathcal{T}$ proves one query atom in Q. By Lemma 4 we can assume that every element in the universe A either interprets some constant $c \in \bar{c}$ or occurs in some proof tree $T \in Ts$. Note that for all facts $p(a)$, we can w. l. o. g. assume that all $p(a)$-*subtrees*, i. e., all subtrees rooted at $p(a)$-nodes, of proof trees in $\mathcal{T}$ are isomorphic.

Suppose there is a path of length greater than (2) in some proof tree $T \in \mathcal{T}$. The first node on that path, the root of T, is a *constant node*, i. e., it is a $p(a)$-node for some IDB predicate $p \in \bar{p}$ and some $a \in A$ interpreting a constant $c \in \bar{c}$. The constant nodes, of which there are at most $|\bar{p}||\bar{c}|$ on the path (recall that no fact $p(a)$ repeats on a path), divide the path into up to $|\bar{p}||\bar{c}|$ segments of non-constant nodes. Choose such a segment of non-constant nodes which is of maximal length, i. e., its length is greater than $\theta \cdot |\bar{p}| \cdot \delta + 1$. Drop the first node from the chosen segment and denote the remaining segment by π, so $|\pi| > \theta \cdot |\bar{p}| \cdot \delta$. To every node n of π, we associate a 2-type, written $\tau_{\mathbf{A}}(n)$, which is defined by $\tau_{\mathbf{A}}(n) = \tau_{\mathbf{A}}(a, b)$ where $a, b \in A$ such that n is a b-node and the father of n in T is an a-node. Since there are at most θ 2-types inhabited in $\mathbf{A}$, there exist a 2-type $\tau(u, v)$ and a set N_τ of nodes on π such $\tau_{\mathbf{A}}(n) = \tau(u, v)$ for all $n \in N_\tau$ and $|N_\tau| > |\bar{p}| \cdot \delta$. There exist an IDB predicate $q \in \bar{p}$ and a subset $N_{\tau,q} \subseteq N_\tau$ such that all $n \in N_{\tau,q}$ are q-nodes and $|N_{\tau,q}| > \delta$. Let $n \in N_{\tau,q}$ be the node which is closest to the root of T, and let m be the father of n in T. Since m has at most δ sons, there exists a node $n' \in N_{\tau,q}$, labeled by some fact $q(a)$, such that no son of m is an a-node. Let m' be the father of n' in T, and let $d, e, d', e' \in A$ be the elements labeling the nodes m, n, m' and n', respectively. Note that the four elements d, e, d' and e' are pairwise distinct because they are not interpreting constants, and in a proof tree of an intersection-free Datalog program such elements cannot repeat on a path.

To summarize, the proof tree $T \in \mathcal{T}$ contains two father-son pairs $\langle m, n\rangle$ and $\langle m', n'\rangle$ on one path, with m being closer to the root than m', such that the following properties hold:

1. The nodes m, n, m' and n' are non-constant nodes and there is no constant node between them.
2. The elements $d, e, d', e' \in A$ labeling the respective nodes m, n, m' and n' are pairwise distinct.
3. The 2-types of the elements d, e and d', e' coincide, i. e., the type quality $\tau_{\mathbf{A}}(d, e) = \tau_{\mathbf{A}}(d', e')$ holds.
4. The nodes n and n' are labeled by the same IDB-predicate q.
5. The node m does not have an e'-son.

As m is the father of n (and assuming that n is the i-th son), there is a clause $C = p(u) \leftarrow \ldots, r_i(u, v_i), q_i(v_i), \ldots$ in $\mathcal{P}$ such that $q_i = q$ and $r_i^{\mathbf{A}}(d, e)$. In order to decrease the size (and eventually the depth) of the proof tree T, we remove from it the $q(e)$-subtree rooted at n and put the $q(e')$-subtree rooted at n' at its place. We do the same replacement in all proof trees in $\mathcal{T}$ at all $q(e)$-nodes, i. e., we replace every $q(e)$-subtree by its $q(e')$-subtree; recall that all $q(e)$-subtrees are isomorphic, so each has a $q(e')$-subtree. Call the resulting set of trees $\mathcal{T}'$. Note that these replacements necessarily happen below $p(d)$-nodes because every $q(e)$-node is a son of a $p(d)$-node. The reason for this is the non-sharing restriction, which implies that there is no predicate s occurring in $\mathcal{P}$ and no element $a \in A$ other than d such that $s^{\mathbf{A}}(a, e)$ holds, so every e-node must be the son a d-node. Below we will check that the new trees in $\mathcal{T}'$ are valid proof trees, which basically involves checking that the above clause C is satisfied at $p(d)$-nodes. However, for this to hold, we first have to update the model $\mathbf{A}$ in such a way that $r_i^{\mathbf{A}}(d, e')$ becomes true while the updated model stays a model of ϕ.

Call the updated model $\mathbf{B}$. As the universe of $\mathbf{B}$ we take a subset $B \subseteq A$ such that every element in B interprets a constant or labels a node in one of the trees in $\mathcal{T}$. The interpretation of the constants does not change, i. e., $c^{\mathbf{A}} = c^{\mathbf{B}}$ for all constants c. We define the interpretation of each binary predicate s in $\mathbf{B}$ such that $s^{\mathbf{B}}(d, e')$ iff $s^{\mathbf{A}}(d, e)$ and $s^{\mathbf{B}}(e', d)$ iff $s^{\mathbf{A}}(e, d)$, and for all $a, b \in B$ with $\{a, b\} \neq \{d, e'\}$, we define $s^{\mathbf{B}}(a, b)$ iff $s^{\mathbf{A}}(a, b)$. That is, $\mathbf{B}$ relates d and e' in the same way than $\mathbf{A}$ relates d and e, otherwise the interpretations do not differ.

Note that the element e is no longer in B since it does not interpret a constant and has been eliminated from all proof trees. The latter is case because all $q(e)$-nodes have been eliminated, and no tree in $\mathcal{T}$ contains any other e-node since $\mathcal{P}$ is intersection-free. Also, the element d' is no longer in B. To see this, let $a_0, a_1, \ldots, a_l$ denote the elements labeling the nodes from n to m' in the old proof tree T, i. e., $a_0 = e$ and $a_l = d'$. Let $r_1, \ldots, r_l$ be a sequence of EDB predicates occurring in the program $\mathcal{P}$ such that $r_j^{\mathbf{A}}(a_{j-1}, a_j)$ for $1 \leq j \leq l$; by construction of the proof tree T such a sequence of predicates exists. Because of the non-sharing restriction, for $1 \leq j \leq l$, there is no predicate s occurring in $\mathcal{P}$ and no element $a \in A$ other than a_{j-1} such that $s^{\mathbf{A}}(a, a_j)$ holds. Thus, each a_j can only occur in one of the new proof trees if a_{j-1} does occur. As $a_0 = e$ has been eliminated, all the other a_j are eliminated as well, in particular we have $d' = a_l \notin B$.

We have to show that the new model $\mathbf{B}$ satisfies the functionality restriction. Assume that s is a binary predicate. The critical cases for s to satisfy the functionality restriction $\forall u, v_1, v_2 \big(s(u, v_1) \wedge s(u, v_2) \Rightarrow v_1 \approx v_2\big)$ in $\mathbf{B}$ are those involving the new edges between d and e'. There are two such cases.

- Assume $s^{\mathbf{B}}(d, e')$ and $s^{\mathbf{B}}(d, b)$ for some $b \in B$. Towards a contradiction assume that $e' \neq b$. By construction of $\mathbf{B}$, we have $s^{\mathbf{A}}(d, e)$ and $s^{\mathbf{A}}(d, b)$. Due to functionality in $\mathbf{A}$, this implies $b = e$, which is a contradiction because $e \notin B$.

- Assume $s^{\mathbf{B}}(e', d)$ and $s^{\mathbf{B}}(e', b)$ for some $b \in B$. Towards a contradiction assume that $d \neq b$. By construction, we have $s^{\mathbf{A}}(e, d)$ and $s^{\mathbf{A}}(e', b)$. However, since $\langle d, e\rangle$ and $\langle d', e'\rangle$ inhabit the same 2-type, we have also $s^{\mathbf{A}}(e', d')$. Due to functionality in $\mathbf{A}$, this implies $b = d'$, which is a contradiction because $d' \notin B$.

Next, we have to show that $\mathbf{B}$ satisfies the non-sharing restriction. Assume that s is a binary predicate occurring in $\mathcal{P}$. The critical cases for s to satisfy the first non-sharing restriction $\forall u_1, u_2, v\big(s(u_1, v) \wedge s(u_2, v) \wedge u_1 \not\approx u_2 \Rightarrow \mathit{const}(v)\big)$ in $\mathbf{B}$ are those involving the new edges between d and e'. Potentially there are two such cases, but we will show that each one leads to a contradiction.

- Assume $s^{\mathbf{B}}(d, e')$ and $s^{\mathbf{B}}(b, e')$ and $d \neq b$ for some $b \in B$. By construction of $\mathbf{B}$, we have $s^{\mathbf{A}}(d, e)$ and $s^{\mathbf{A}}(b, e')$. However, since $\langle d, e\rangle$ and $\langle d', e'\rangle$ inhabit the same 2-type, we have also $s^{\mathbf{A}}(d', e')$ and $d' \neq b$ because $d' \notin B$. Due to non-sharing in $\mathbf{A}$, e' must interpret a constant, which contradicts n' being a non-constant node.

- Assume $s^{\mathbf{B}}(e', d)$ and $s^{\mathbf{B}}(b, d)$ and $e' \neq b$ for some $b \in B$. By construction, we have $s^{\mathbf{A}}(e, d)$ and $s^{\mathbf{A}}(b, d)$. Due to non-sharing in $\mathbf{A}$, d must interpret a constant, which contradicts m being a non-constant node.

Now assume that s_1, s_2 are two distinct binary predicates occurring in $\mathcal{P}$. The critical cases for s_1 and s_2 to satisfy the second non-sharing restriction $\forall u_1, u_2, v\big(s_1(u_1, v) \wedge s_2(u_2, v) \Rightarrow \mathit{const}(v)\big)$ in $\mathbf{B}$ are those involving the new edges between d and e'. As s_1 and s_2 are symmetric in the non-sharing restriction, there are two such cases.

- Assume $s_1^{\mathbf{B}}(d, e')$ and $s_2^{\mathbf{B}}(b, e')$ for some $b \in B$.
  - If $d = b$ then by construction of $\mathbf{B}$, we have $s_1^{\mathbf{A}}(d, e)$ and $s_2^{\mathbf{A}}(d, e)$, which due to non-sharing in $\mathbf{A}$ implies that e must interpret a constant. This contradicts n being a non-constant node.
  - If $d \neq b$ then we have $s_1^{\mathbf{A}}(d, e)$ and $s_2^{\mathbf{A}}(b, e')$. However, since $\langle d, e\rangle$ and $\langle d', e'\rangle$ inhabit the same 2-type, we have also $s_1^{\mathbf{A}}(d', e')$, which due to non-sharing in $\mathbf{A}$ implies that e' must interpret a constant. This contradicts n' being a non-constant node.

- Assume $s_1^{\mathbf{B}}(e', d)$ and $s_2^{\mathbf{B}}(b, d)$ for some $b \in B$. By construction of $\mathbf{B}$, we have $s_1^{\mathbf{A}}(e, d)$. Moreover, if $e' = b$ then we have $s_2^{\mathbf{A}}(e, d)$, otherwise we have $s_2^{\mathbf{A}}(b, d)$. In any case, d must interpret a constant due to non-sharing in $\mathbf{A}$. This contradicts m being a non-constant node.

Next, we have to show that the new model $\mathbf{B}$ still satisfies the universal BS_2-formula ϕ. Let $\phi = \forall u, v\, \psi$ for some quantifier-free formula ψ. W. l. o. g. we can assume ψ consists only of ground literals and literals of the form $[\neg]z \approx c$, $[\neg]u \approx v$, $[\neg]s(z, z)$, $[\neg]s(z, c)$, $[\neg]s(c, z)$, $[\neg]s(u, v)$ and $[\neg]s(v, u)$ for constants $c \in \bar{c}$, binary relations $s \in \bar{r}$ and variables $z \in \{u, v\}$. We have to show that ψ evaluates to true in $\mathbf{B}$ under all assignments β of the variables u and v to elements of the universe of $\mathbf{B}$. As the models $\mathbf{A}$ and $\mathbf{B}$ only differ in the relations between the elements d and e' (recall that e is no longer in B), ψ obviously evaluates to true as long as the image of β is not $\{d, e'\}$. Consider the critical assignment β with $\beta(u) = d$ and $\beta(v) = e'$. (Since u and v are symmetric in the literals, this also covers the other critical assignment γ with $\gamma(u) = e'$ and $\gamma(v) = d$.) We will show ψ still evaluates to true in $\mathbf{B}$ because each literal evaluates in $\mathbf{B}$ under β the same as in $\mathbf{A}$ under the assignment β' with $\beta'(u) = d$ and $\beta'(v) = e$.

- Ground literals evaluate the same in $\mathbf{B}$ and in $\mathbf{A}$ (regardless of the assignments) because $\mathbf{A}$ and $\mathbf{B}$ agree on the interpretation of the constants and the relations between them.

- Literals of the form $[\neg]u \approx c$, $[\neg]s(u, u)$, $[\neg]s(u, c)$ and $[\neg]s(c, u)$ evaluate the same in $\mathbf{B}$ under β and in $\mathbf{A}$ under β' because $\mathbf{A}$ and $\mathbf{B}$ agree on the relations between d and constants and between d and itself.

- Literals of the form $[\neg]v \approx c$, $[\neg]s(v, v)$, $[\neg]s(v, c)$ and $[\neg]s(c, v)$ evaluate the same in $\mathbf{B}$ under β and in $\mathbf{A}$ under β' because $\tau_{\mathbf{B}}(e') = \tau_{\mathbf{A}}(e') = \tau_{\mathbf{A}}(e)$. The first equality holds because $\mathbf{A}$ and $\mathbf{B}$ agree on the relations between e' and constants and between e' and itself. The second equality is implied by the 2-type equality $\tau_{\mathbf{A}}(d, e) = \tau_{\mathbf{A}}(d', e')$.

- Literals of the form $[\neg]s(u, v)$ and $[\neg]s(v, u)$ evaluate the same in $\mathbf{B}$ under β and in $\mathbf{A}$ under β' because we have $s^{\mathbf{B}}(d, e')$ iff $s^{\mathbf{A}}(d, e)$ and $s^{\mathbf{B}}(e', d)$ iff $s^{\mathbf{A}}(e, d)$ by construction of $\mathbf{B}$.

- Literals of the form $[\neg]u \approx v$ evaluate the same in $\mathbf{B}$ under β and in $\mathbf{A}$ under β' because $d \neq e'$ in $\mathbf{B}$ and $d \neq e$ in $\mathbf{A}$.

Finally, we have to show that the new trees in $\mathcal{T}'$ are in fact valid proof trees proving the query atoms in Q. Let $T \in \mathcal{T}$ be an old proof tree for $\mathcal{P}$ w. r. t. $\mathbf{A}$ and let $T' \in \mathcal{T}'$ be its new counterpart. Note that the roots of T and T' are the same, so if T' is a valid proof tree it will prove the same query atom as T. Because $\mathbf{A}$ and $\mathbf{B}$ only differ in the relations between the elements d and e', in order to show that T' is a proof tree for $\mathcal{P}$ w. r. t. $\mathbf{B}$, it suffices to check those d-nodes in T' which have an e'-son. If T' does not contain a d-node there is nothing to check, so assume that T' contains a d-node. This implies that T contains a d-node, which is a $p(d)$-node because $\mathcal{P}$ is intersection-free. Let $q_1(a_1), \ldots, q_k(a_k)$ be the facts labeling the sons of such a $p(d)$-node; we know that $q_i(a_i) = q(e)$ because the i-th son is a $q(e)$-node. As T is a proof tree, the list of facts $q_1(a_1), \ldots, q_{i-1}(a_{i-1}), q(e), q_{i+1}(a_{i+1}), \ldots, q_k(a_k)$ produces the fact $p(d)$ (w. r. t. $\mathcal{P}$ and $\mathbf{A}$), i. e., $\mathcal{P}$ contains a clause of the form

$$p(u) \leftarrow B_1, \ldots, B_l, r_1(u, v_1), q_1(v_1), \ldots, r_k(u, v_k), q_k(v_k)$$

such that $\mathbf{A}$ satisfies all $r_j(u, v_j)$ atoms and all B_j literals when interpreting the variables $u, v_1, \ldots, v_k$ by $d, a_1, \ldots, a_{i-1}, e, a_{i+1}, \ldots, a_k$, respectively. Now in T, the i-th son of a $p(d)$-node is its only e-son, for if the j-th son, $i \neq j$, also was an e-son, we would have $r_i^{\mathbf{A}}(d,e)$ and $r_j^{\mathbf{A}}(d,e)$, which contradicts to the non-sharing restriction. Hence for showing that T' is a valid proof tree, we have to prove that the list of facts $q_1(a_1), \ldots, q_{i-1}(a_{i-1}), q(e'), q_{i+1}(a_{i+1}), \ldots, q_k(a_k)$, which label the sons of a $p(d)$-node in T', produces $p(d)$. This follows from three facts:

- For the atom $r_i(u, v_i)$, $r_i^{\mathbf{B}}(d, e')$ follows from $r_i^{\mathbf{A}}(d,e)$ by construction of $\mathbf{B}$.
- For all other $r_j(u, v_j)$ atoms, $j \neq i$, nothing changes. This is so because by choice of d (more precisely by choice of the node determining d), a $p(d)$-node in T does not have an e'-son, so $a_j \neq e'$. It follows that we inherit $r_j^{\mathbf{B}}(d, a_j)$ from $r_j^{\mathbf{A}}(d, a_j)$ because $\mathbf{A}$ and $\mathbf{B}$ agree on the relations between d and a_j.
- For all B_j literals, nothing changes because their only free variable is u and the models $\mathbf{A}$ and $\mathbf{B}$ agree on the relations between d and constants and between d and itself.

Obviously, the above modifications of the model $\mathbf{A}$ and the proof trees in $\mathcal{T}$ can be performed as long as there is a proof tree whose depth is greater than (2). This ends the proof of Lemma 7. $\square$

Theorem 8. *Let ϕ be a universal σ-formula with 2 variables, let $\mathcal{P}$ be an intersection-free TA-like monadic Datalog program and let Q be a query. If the formula $\phi \wedge \mathcal{P} \wedge Q$ is satisfiable then it has a model of cardinality at most doubly exponential in the size of the formula. Deciding satisfiability of such formulas is in 2-*NEXPTIME.

Proof. By the lemmas 7 and 6, a satisfiable formula has a model $\mathbf{A}$ where the proof trees for the query atoms are bounded by a function singly exponential in the size of the formula, so their size is at most doubly exponential. By the observation following Lemma 4, the model $\mathbf{A}$ can be reduced to a model $\mathbf{B}$ consisting only of interpretations of the constants and elements occurring in the proof trees. $\square$

3.3 Complexity of Bounded Model Checking

It follows from Theorem 8, that the bounded model checking problem from Theorem 3 (see section 2.6) is in 2-NEXPTIME, because the size of the formula $pre(\pi; \top) \wedge \phi \wedge \mathcal{P} \wedge Q$ is polynomial in $|\pi|$ by Lemma 2. The double exponential complexity originates from two sources, the exponential bound on the number of inhabited 2-types and the (linear) degree of the Datalog program, leading to proof trees of exponential depth and double exponential size. In common situations, however, the complexity of bounded model checking can be improved significantly.

In the following, we consider bounded model checking problems for a fixed program P with a fixed initial condition $\phi \wedge \mathcal{P} \wedge Q$. We say that the formula

$\phi \wedge \mathcal{P} \wedge Q$ is a *bounded branching formula* if there is $k \geq 0$ such that all models of $\phi \wedge \mathcal{P} \wedge Q$ are k-branching.

Theorem 9. *Let $\pi = \langle init, \ldots, error \rangle$ be an ECFG-path of a program P. If the initial condition $\phi \wedge \mathcal{P} \wedge Q$ is a bounded branching formula then (for fixed program and fixed initial condition) deciding whether π is pointer-safe is in* NEXPTIME. *If additionally the degree of $\mathcal{P}$ is* 1 *then the problem is in* NPTIME.

Proof. Assume that all models of $\phi \wedge \mathcal{P} \wedge Q$ are k-branching for some fixed $k \geq 0$. By Lemma 2, the size of the precondition formula $pre(\pi; \top)$ is polynomial in $|\pi|$, in particular the length of the existential quantifier prefix is linear in $|\pi|$. For checking satisfiability of $pre(\pi; \top) \wedge \phi \wedge \mathcal{P} \wedge Q$, we convert $pre(\pi; \top)$ into a universal formula, which requires extending the EDB vocabulary σ by adding a number (linear in $|\pi|$) of new constants. By Lemma 6, the number of inhabited 2-types in models of $pre(\pi; \top) \wedge \phi \wedge \mathcal{P} \wedge Q$ is polynomial in the number of constants,[6] hence polynomial in $|\pi|$. Thus, Lemma 7 yields a polynomial depth bound for proof trees, which implies a singly exponential bound on the size of the models. If the degree of $\mathcal{P}$ is 1, the polynomial depth bound implies a polynomial size bound. $\square$

Functionality and non-sharing restrictions ensure that all initial conditions in our examples are bounded branching formulas. The models of the initial conditions for lists (singly or doubly linked, circular or not) and singly linked trees (binary or general) are all 1-branching, whereas the models of the initial conditions for doubly linked binary trees are 2-branching. Thus for all these data structures, bounded model checking can be done in NEXPTIME, even if the program manipulates all these data structures simultaneously. Moreover, if a program works on list data structures only then bounded model checking can be done in NPTIME, which is the optimal worst-case complexity for BMC of list manipulating programs.

4 Related Work

Automatic verification of pointer programs has received quite some attention recently. Dynamically allocated heap memory and properties such as sharing, cyclicity, and reachability in the heap have been formalized in various logical languages.

Abstraction from possibly unbounded state space to a finite model has been studied in [15, 23, 25]. These approaches use the framework of abstract interpretation to over-approximate the set of reachable states. This is achieved by interpreting program statements and properties in a 3-valued first-order logic with transitive closure (TC). Recently there have been attempts to increase the precision of the approximation by incorporating automated theorem for classical 2-valued first-order logic into the 3-valued setting [25].

[6]That the number of 2-types is exponential in k and the number of binary relations is irrelevant because those parameters are fi xed.

Other approaches for shape analysis use decidable extensions of first-order fragments to reason about shape graphs. [14] proves decidable the $\exists\forall$-fragment with restricted occurrences of TC and deterministic TC. Unfortunately, without severe restrictions on transitive closure, most decidable fragments of first-order logic become undecidable [14]. In [24], the decidable guarded fixed-point logic μGF [10] is used for shape analysis. In μGF, one can express reachability from specified points along specified paths, but full transitive closure (i. e., reachability between a pair of variables) is inexpressible. Moreover, μGF lacks the finite model property [10] and becomes undecidable when functionality restrictions are added [8].

Special syntactically defined logics for expressing reachability have been designed. The reachability logic $\mathcal{RL}$ defined in [1] is a fragment of 2-variable first-order logic with transitive closure and additional Boolean variables. Expressive logics like PDL and CTL* can be embedded into it. Model checking for $\mathcal{RL}$ is efficient, but decidability of satisfiability has not been investigated.

In order to employ decision procedures for monadic second order logic over trees, Schwartzbach et al. model linked data structures using graph types [17, 19]. Graph types are logical representations of sets of graphs, where each graph has a tree backbone which uniquely defines the other (tree-violating) edges. The approach is similar to ours because the Datalog proof trees can be seen as tree backbones. However, the two approaches differ in how to specify the tree-violating edges. We can (but do not have to) specify global restrictions on the tree-violating edges in a fragment of first-order logic whereas graph types specify their tree-violating edges in a dynamic logic.

Graphs as models for software systems that contain pointers have been studied in [18, 21] where graph logics based on C^2, the 2-variable first-order logic with counting quantifiers have been defined. These logics can also be seen as variants of description logics [2] without fixed-points or transitive closure, hence they can model graphs but cannot express reachability. Via translations to C^2, the logics in [18] and [21] inherit decidability [9].

The functional modal fragment of first-order logic as defined by Herzig [12] is a target logic for mapping basic modal and description logics into the framework of first-order logic. The good computational properties of these logic, i. e., PSPACE-decidability and the finite model property, carry over to the functional modal fragment of first-order logic. In this fragment universal and existential quantification can be permuted [11] (i. e., $\forall\exists$ can be exchanged by $\exists\forall$), hence deciding satisfiability can be reduced to deciding the Bernays-Schönfinkel class.

Reynolds and O'Hearn introduced separation logic [22] and a Hoare-style proof system for local reasoning about pointer programs. In this approach, verification requires manual construction of proofs for Hoare-triplets because the logic is undecidable. Towards more automation, in [5] a decidable fragment of separation logic is studied. To obtain decidability expressiveness has to be sacrificed: the fragment can only specify singly linked lists. On the other hand, besides satisfiability also entailment is decidable, which is crucial for verification.

In hardware verification, bounded model checking using propositional SAT solvers was adopted as a standard technique almost immediately after its introduction by Biere et al. [4] in 1999. Jackson and Vaziri [16] extended SAT-based BMC from hardware circuits to Java-like imperative programs with heap references. Essentially, they translate the program specification and bounded executions of the program to a formula in first-order logic with transitive closure, which they check for satisfiability in small models using a SAT-solver. The use of first-order logic with TC as a specification language is very convenient, however, SAT-checking is only feasible on very small models, i. e., the size of the heap must be bounded a priori to only few cells. Similar approaches to SAT-based BMC of pointer programs are pursued in [6] and [13].

5 Conclusion and Future Work

We proposed a bounded model checking procedure for programs manipulating dynamically allocated pointer structures of arbitrary size. The worst-case complexity of our method is 2-NEXPTIME, but in common cases it goes down to NEXPTIME or even to NPTIME. Our approach is based on a combination of two logics, both of which are efficiently decidable in practice. Therefore, we hope that our algorithm can be implemented (e. g., by integrating a Datalog inference engine into a Bernays-Schönfinkel decision procedure) quite efficiently.

There are several possible directions for the future work. One of them is implementation via combination of decision procedures for Bernays-Schönfinkel class and Datalog. Another one is investigation of further applications of our method, e. g., in the analysis whether counterexamples generated by an abstraction-refinement model checker for pointer programs are spurious. Still another direction is to extend the applicability of the method, e. g., by releasing the non-sharing restriction (currently we are not able to express structures like DAG representation of trees). Further possibility is to extend our bounded model checking (which is a debugging method) to a verification method — this requires the ability to express the negation of initial conditions which leads to Datalog programs with greatest fixed point semantics (as opposed to the least fixed point semantics considered here). Finally, we consider the use of other decidable fragments of the first-order logic whose combination with Datalog could lead to a decidable logic expressive enough for reasoning about pointer structures. A good candidate is C^2, the 2-variable fragment of first-order logic with counting quantifiers, in which restrictions like functionality or non-sharing are easily expressible. As C^2 is closed under negation, a Datalog extension with greatest fixed points should also be well suited for verification of invariants.

References

[1] N. Alechina and N. Immerman. Reachability logic: An efficient fragment of transitive closure logic. *Logic Journal of IGPL*, 8:325–337, 2000.

[2] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[3] P. Bernays and M. Schönfinkel. Zum Entscheidungsproblem der Mathematischen Logik. *Math. Annalen*, 99:342–372, 1928.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDD. In *Proc. TACAS'99*, LNCS 1579, pages 193–207. Springer, 1999.

[5] R. Bornat, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *Proc. FSTTCS'04*, LNCS 3328, pages 97–109. Springer, 2004.

[6] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS'04*, LNCS 2988, pages 168–176. Springer, 2004.

[7] B. Dreben and W. D. Goldfarb. *The Decision Problem. Solvable Classes of Quantificational Formulas*. Addison-Wesley, 1979.

[8] E. Grädel. On the restraining power of guards. *J. Symbolic Logic*, 64:1719–1742, 1999.

[9] E. Grädel, M. Otto, and E. Rosen. Two-variable logic with counting is decidable. In *Proc. LICS'97*, pages 306–317. IEEE Computer Society Press, 1997.

[10] E. Grädel and I. Walukiewicz. Guarded fixed point logic. In *Proc. LICS'99*, pages 45–54. IEEE Computer Society Press, 1999.

[11] A. Herzig. *Raisonnement automatique en logique modale et algorithmes d'unification*. PhD thesis, Université Paul Sabatier, Toulouse, 1989.

[12] A. Herzig. A new decidable fragment of first order logic, June 1990. In Abstracts of the 3rd Logical Biennial, Summer School & Conference in honour of S. C. Kleene, Varna, Bulgaria.

[13] M. Huth and S. Pradhan. Consistent partial model checking. *Electronic Notes in Theoretical Computer Science*, 23, 2003.

[14] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Proc. CSL'04*, LNCS 3210, pages 160–174. Springer, 2004.

[15] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via structure simulation. In *Proc. CAV'04*, LNCS 3114, pages 281–294. Springer, 2004.

[16] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proc. ISSTA'00*, pages 14–25, 2000.

[17] N. Klarlund and M. I. Schwartzbach. Graph types. In *Proc. POPL'93*, pages 196–205, 1993.

[18] V. Kuncak and M. Rinard. On role logic. Technical Report 925, MIT Computer Science and Artificial Intelligence Laboratory, 2003.

[19] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proc. PLDI'01*, pages 221–231, 2001.

[20] F. Ramsey. On a problem of formal logic. In *Proc. London Mathematical Society*, pages 338–384, 1928.

[21] A. Rensink. Canonical graph shapes. In *Proc. ESOP'04*, LNCS 2986, pages 401–415. Springer, 2004.

[22] J. Reynolds. Intuitionistic reasoning about shared mutable data structure, 1999. Proc. of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare.

[23] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape-analysis problems via 3-valued logic. *ACM TOPLAS*, 24(2):217–298, 2002.

[24] T. Wies. Symbolic shape analysis. Master's thesis, MPI Informatik, Saarbrücken, Germany, 2004.

[25] G. Yorsh, T. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Proc. TACAS'04*, LNCS 2988, pages 530–545. Springer, 2004.

A Weakest Preconditions Expressed in First-Order Logic

This appendix is devoted to the proof of Lemma 1. For convenience, we state it again.

Lemma 1. *Given a σ-formula φ and an action α, we have the following characterization of $pre(\alpha;\varphi)$, where T is a template, $\bar{s} = \{s_1,\dots,s_n\}$ are the fields of T, s is an arbitrary field, x and y are program variables, e is a program variable or constant (including $NULL$), and γ is a σ-formula.*

$$\begin{aligned}
pre(assume(\gamma);\varphi) &\equiv \gamma \wedge \varphi\\
pre(y := e;\varphi) &\equiv \varphi[e/y]\\
pre(y := s(x);\varphi) &\equiv \exists y'\big(s(x,y') \wedge \varphi[y'/y]\big)\\
pre(s(x) := e;\varphi) &\equiv \varphi[u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v)/s(u,v)]\\
pre(free_T(x);\varphi) &\equiv \varphi[u \not\approx x \wedge \bar{s}(u,v)/\bar{s}(u,v)]\\
pre(y := new_T();\varphi) &\equiv \exists y'\big(y' \not\approx NULL \wedge \textstyle\bigwedge_{s\in\bar{s}} \forall u,v(s(u,v) \Rightarrow u \not\approx y') \wedge{}\\
&\qquad \varphi[y'/y][u \approx y' \wedge v \approx NULL \vee \bar{s}(u,v)/\bar{s}(u,v)]\big)
\end{aligned}$$

Proof. We will prove the lemma by a case distinction on α.

Assuming a condition. Let $\alpha = assume(\gamma)$. We prove this first case in all detail; later cases will be presented less verbose.

$$\begin{aligned}
& pre(assume(\gamma);\varphi)\\
\equiv\quad & \text{by definition}\\
& \exists\bar{r}'\exists\bar{c}'\big(\gamma \wedge \bigwedge_{c\in\bar{c}} c' \approx c \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[\bar{r}'/\bar{r}][\bar{c}'/\bar{c}]\big)\\
\equiv\quad & \text{substitute } \bar{c} \text{ for } \bar{c}' \text{ in } \varphi[\dots][\dots]\\
& \exists\bar{r}'\exists\bar{c}'\big(\gamma \wedge \bigwedge_{c\in\bar{c}} c' \approx c \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[\bar{r}'/\bar{r}][\bar{c}'/\bar{c}][\bar{c}/\bar{c}']\big)\\
\equiv\quad & \text{transitivity of substitution}\\
& \exists\bar{r}'\exists\bar{c}'\big(\gamma \wedge \bigwedge_{c\in\bar{c}} c' \approx c \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[\bar{r}'/\bar{r}]\big)\\
\equiv\quad & \text{push } \exists\bar{c}' \text{ inwards}\\
& \exists\bar{r}'\big(\gamma \wedge \bigwedge_{c\in\bar{c}} \exists c'(c' \approx c) \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[\bar{r}'/\bar{r}]\big)\\
\equiv\quad & \text{eliminate } \exists c' \dots\\
& \exists\bar{r}'\big(\gamma \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[\bar{r}'/\bar{r}]\big)\\
\equiv\quad & \text{substitute } \bar{r} \text{ for } \bar{r}' \text{ in } \varphi[\dots]\\
& \exists\bar{r}'\big(\gamma \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[\bar{r}'/\bar{r}][\bar{r}/\bar{r}']\big)
\end{aligned}$$

$\equiv$ transitivity of substitution

$$\exists\bar{r}'\big(\gamma \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi\big)$$

$\equiv$ push $\exists\bar{r}'$ inwards

$$\gamma \wedge \bigwedge_{r\in\bar{r}} \exists r'(r' = r) \wedge \varphi$$

$\equiv$ eliminate $\exists r' \ldots$

$$\gamma \wedge \varphi$$

Assignment. Let $\alpha = y := e$. Recall that y and e are constants in $\bar{c}$.

$$pre(y := e; \varphi)$$

$\equiv$ by definition

$$\exists\bar{r}'\exists\bar{c}'\big(y' := e \wedge \bigwedge_{c\in\bar{c}\setminus\{y\}} c' \approx c \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[\bar{r}'/\bar{r}][\bar{c}'/\bar{c}]\big)$$

$\equiv$ substitute e for y' and c for c', $c \in \bar{c} \setminus \{y\}$, in $\varphi[\ldots][\ldots]$; push $\exists\bar{c}'$ inwards

$$\exists\bar{r}'\big(\exists y'(y' := e) \wedge \bigwedge_{c\in\bar{c}\setminus\{y\}} \exists c'(c' \approx c) \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[\bar{r}'/\bar{r}][e/y]\big)$$

$\equiv$ eliminate $\exists y' \ldots$ and $\exists c' \ldots$; swap order of substitutions in $\varphi[\ldots][\ldots]$

$$\exists\bar{r}'\big(\bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[e/y][\bar{r}'/\bar{r}]\big)$$

$\equiv$ substitute $\bar{r}$ for $\bar{r}'$ in $\varphi[\ldots][\ldots]$; push $\exists\bar{r}'$ inwards and eliminate

$$\varphi[e/y]$$

Read access. Let $\alpha = y := s(x)$. Recall that x and y are constants in $\bar{c}$ and s is a relation symbol in $\bar{r}$.

$$pre(y := s(x); \varphi)$$

$\equiv$ by definition

$$\exists\bar{r}'\exists\bar{c}'\big(s(x, y') \wedge \bigwedge_{c\in\bar{c}\setminus\{y\}} c' \approx c \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[\bar{r}'/\bar{r}][\bar{c}'/\bar{c}]\big)$$

$\equiv$ substitute c for c', $c \in \bar{c} \setminus \{y\}$, in $\varphi[\ldots][\ldots]$; push $\exists c'$ inwards, $c' \in \bar{c}' \setminus \{y'\}$

$$\exists\bar{r}'\exists y'\big(s(x, y') \wedge \bigwedge_{c\in\bar{c}\setminus\{y\}} \exists c'(c' \approx c) \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[\bar{r}'/\bar{r}][y'/y]\big)$$

$\equiv$ eliminate $\exists c' \ldots$; swap order of substitutions in $\varphi[\ldots][\ldots]$

$$\exists\bar{r}'\exists y'\big(s(x, y') \wedge \bigwedge_{r\in\bar{r}} r' = r \wedge \varphi[y'/y][\bar{r}'/\bar{r}]\big)$$

$\equiv$ substitute $\bar{r}$ for $\bar{r}'$ in $\varphi[\ldots][\ldots]$; push $\exists\bar{r}'$ inwards and eliminate

$$\exists y'\big(s(x, y') \wedge \varphi[y'/y]\big)$$

Write access. Let $\alpha = s(x) := e$. Recall that x and e are constants in $\bar{c}$ and s is a relation symbol in $\bar{r}$.

$$
\begin{array}{ll}
& \mathit{pre}(s(x) := e; \varphi) \\
\equiv & \text{by definition} \\
& \exists \bar{r}' \exists \bar{c}' \big(\forall u, v \big(s'(u,v) \Leftrightarrow u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v) \big) \wedge \\
& \qquad \bigwedge_{c \in \bar{c}} c' \approx c \wedge \bigwedge_{r \in \bar{r} \setminus \{s\}} r' = r \wedge \\
& \qquad \varphi[\bar{r}'/\bar{r}][\bar{c}'/\bar{c}] \big) \\
\equiv & \text{substitute } \bar{c} \text{ for } \bar{c}' \text{ in } \varphi[\ldots][\ldots]\text{; push } \exists \bar{c}' \text{ inwards and eliminate} \\
& \exists \bar{r}' \big(\forall u, v \big(s'(u,v) \Leftrightarrow u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v) \big) \wedge \\
& \quad \bigwedge_{r \in \bar{r} \setminus \{s\}} r' = r \wedge \\
& \quad \varphi[\bar{r}'/\bar{r}] \big) \\
\equiv & \text{substitute } r \text{ for } r'\text{, } r \in \bar{r} \setminus \{s\}\text{, in } \varphi[\ldots]\text{; push } \exists r' \text{ inwards, } r' \in \bar{r}' \setminus \{s'\} \\
& \exists s' \big(\forall u, v \big(s'(u,v) \Leftrightarrow u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v) \big) \wedge \\
& \quad \bigwedge_{r \in \bar{r} \setminus \{s\}} \exists r' (r' = r) \wedge \\
& \quad \varphi[s'/s] \big) \\
\equiv & \text{eliminate } \exists r' \ldots\text{; apply equivalence for } s' \text{ to } \varphi[\ldots] \\
& \exists s' \big(\forall u, v \big(s'(u,v) \Leftrightarrow u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v) \big) \wedge \\
& \quad \varphi[s'/s][u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v) / s'(u,v)] \big) \\
\equiv & \text{transitivity of substitution} \\
& \exists s' \big(\forall u, v \big(s'(u,v) \Leftrightarrow u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v) \big) \wedge \\
& \quad \varphi[u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v) / s(u,v)] \big) \\
\equiv & \text{push } \exists s' \text{ inwards} \\
& \exists s' \forall u, v \big(s'(u,v) \Leftrightarrow u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v) \big) \wedge \\
& \varphi[u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v) / s(u,v)] \\
\equiv & \text{eliminate } \exists s' \ldots \\
& \varphi[u \approx x \wedge v \approx e \vee u \not\approx x \wedge s(u,v) / s(u,v)]
\end{array}
$$

Deallocation. Let $\alpha = \mathit{free}_T(x)$. Recall that x is constant in $\bar{c}$ and all $s \in \bar{s}$ are relation symbols in $\bar{r}$.

$$
\begin{array}{ll}
& \mathit{pre}(\mathit{free}_T(x); \varphi) \\
\equiv & \text{by definition} \\
& \exists \bar{r}' \exists \bar{c}' \big(\bigwedge_{s \in \bar{s}} \forall u, v \big(s'(u,v) \Leftrightarrow u \not\approx x \wedge s(u,v) \big) \wedge \\
& \qquad \bigwedge_{c \in \bar{c}} c' \approx c \wedge \bigwedge_{r \in \bar{r} \setminus \bar{s}} r' = r \wedge \\
& \qquad \varphi[\bar{r}'/\bar{r}][\bar{c}'/\bar{c}] \big) \\
\equiv & \text{substitute } \bar{c} \text{ for } \bar{c}' \text{ in } \varphi[\ldots][\ldots]\text{; push } \exists \bar{c}' \text{ inwards and eliminate}
\end{array}
$$

$$\exists\bar{r}'\big(\textstyle\bigwedge_{s\in\bar{s}}\forall u,v\big(s'(u,v)\Leftrightarrow u\not\approx x\wedge s(u,v)\big)\wedge \\ \textstyle\bigwedge_{r\in\bar{r}\setminus\bar{s}} r'=r\wedge \\ \varphi[\bar{r}'/\bar{r}]\big)$$

$\equiv$ substitute r for r', $r\in\bar{r}\setminus\bar{s}$, in $\varphi[\ldots]$; push $\exists r'$ inwards, $r'\in\bar{r}'\setminus\bar{s}'$

$$\exists\bar{s}'\big(\textstyle\bigwedge_{s\in\bar{s}}\forall u,v\big(s'(u,v)\Leftrightarrow u\not\approx x\wedge s(u,v)\big)\wedge \\ \textstyle\bigwedge_{r\in\bar{r}\setminus\bar{s}}\exists r'(r'=r)\wedge \\ \varphi[\bar{s}'/\bar{s}]\big)$$

$\equiv$ eliminate $\exists r'\ldots$; apply equivalence for s' to $\varphi[\ldots]$, $s'\in\bar{s}'$

$$\exists\bar{s}'\big(\textstyle\bigwedge_{s\in\bar{s}}\forall u,v\big(s'(u,v)\Leftrightarrow u\not\approx x\wedge s(u,v)\big)\wedge \\ \varphi[u\not\approx x\wedge\bar{s}(u,v)/\bar{s}(u,v)]\big)$$

$\equiv$ push $\exists\bar{s}'$ inwards

$$\textstyle\bigwedge_{s\in\bar{s}}\exists s'\forall u,v\big(s'(u,v)\Leftrightarrow u\not\approx x\wedge s(u,v)\big)\wedge \\ \varphi[u\not\approx x\wedge\bar{s}(u,v)/\bar{s}(u,v)]$$

$\equiv$ eliminate $\exists s'\ldots$

$$\varphi[u\not\approx x\wedge\bar{s}(u,v)/\bar{s}(u,v)]$$

Allocation. Let $\alpha = y := new_T()$. Recall that y is constant in $\bar{c}$ and all $s\in\bar{s}$ are relation symbols in $\bar{r}$.

$$pre(y:=new_T();\varphi)$$

$\equiv$ by definition

$$\exists\bar{r}'\exists\bar{c}'\big(y'\not\approx NULL\wedge\textstyle\bigwedge_{s\in\bar{s}}\forall u,v(s(u,v)\Rightarrow u\not\approx y')\wedge \\ \textstyle\bigwedge_{s\in\bar{s}}\forall u,v\big(s'(u,v)\Leftrightarrow u\approx y'\wedge v\approx NULL\vee s(u,v)\big)\wedge \\ \textstyle\bigwedge_{c\in\bar{c}\setminus\{y\}} c'\approx c\wedge\textstyle\bigwedge_{r\in\bar{r}\setminus\bar{s}} r'=r\wedge \\ \varphi[\bar{r}'/\bar{r}][\bar{c}'/\bar{c}]\big)$$

$\equiv$ substitute c for c', $c\in\bar{c}\setminus\{y\}$, in $\varphi[\ldots][\ldots]$; push $\exists c'$ inwards, $c'\in\bar{c}'\setminus\{y'\}$

$$\exists\bar{r}'\exists y'\big(y'\not\approx NULL\wedge\textstyle\bigwedge_{s\in\bar{s}}\forall u,v(s(u,v)\Rightarrow u\not\approx y')\wedge \\ \textstyle\bigwedge_{s\in\bar{s}}\forall u,v\big(s'(u,v)\Leftrightarrow u\approx y'\wedge v\approx NULL\vee s(u,v)\big)\wedge \\ \textstyle\bigwedge_{c\in\bar{c}\setminus\{y\}}\exists c'(c'\approx c)\wedge\textstyle\bigwedge_{r\in\bar{r}\setminus\bar{s}} r'=r\wedge \\ \varphi[\bar{r}'/\bar{r}][y'/y]\big)$$

$\equiv$ eliminate $\exists c'\ldots$; swap order of substitutions in $\varphi[\ldots][\ldots]$

$$\exists\bar{r}'\exists y'\big(y'\not\approx NULL\wedge\textstyle\bigwedge_{s\in\bar{s}}\forall u,v(s(u,v)\Rightarrow u\not\approx y')\wedge \\ \textstyle\bigwedge_{s\in\bar{s}}\forall u,v\big(s'(u,v)\Leftrightarrow u\approx y'\wedge v\approx NULL\vee s(u,v)\big)\wedge \\ \textstyle\bigwedge_{r\in\bar{r}\setminus\bar{s}} r'=r\wedge \\ \varphi[y'/y][\bar{r}'/\bar{r}]\big)$$

$\equiv$ substitute r for r', $r\in\bar{r}\setminus\bar{s}$, in $\varphi[\ldots][\ldots]$; push $\exists r'$ inwards, $r'\in\bar{r}'\setminus\bar{s}'$

$$\begin{aligned}
& \exists \bar{s}' \exists y' \big(y' \not\approx \mathit{NULL} \wedge \textstyle\bigwedge_{s \in \bar{s}} \forall u, v (s(u,v) \Rightarrow u \not\approx y') \wedge \\
& \qquad \textstyle\bigwedge_{s \in \bar{s}} \forall u, v \big(s'(u,v) \Leftrightarrow u \approx y' \wedge v \approx \mathit{NULL} \vee s(u,v) \big) \wedge \\
& \qquad \textstyle\bigwedge_{r \in \bar{r} \setminus \bar{s}} \exists r' (r' = r) \wedge \\
& \qquad \varphi[y'/y][\bar{s}'/\bar{s}] \big) \\
\equiv \quad & \text{eliminate } \exists r' \ldots \text{; apply equivalence for } s' \text{ to } \varphi[\ldots][\ldots],\ s' \in \bar{s}' \\
& \exists \bar{s}' \exists y' \big(y' \not\approx \mathit{NULL} \wedge \textstyle\bigwedge_{s \in \bar{s}} \forall u, v (s(u,v) \Rightarrow u \not\approx y') \wedge \\
& \qquad \textstyle\bigwedge_{s \in \bar{s}} \forall u, v \big(s'(u,v) \Leftrightarrow u \approx y' \wedge v \approx \mathit{NULL} \vee s(u,v) \big) \wedge \\
& \qquad \varphi[y'/y][u \approx y' \wedge v \approx \mathit{NULL} \vee \bar{s}(u,v)/\bar{s}(u,v)] \big) \\
\equiv \quad & \text{push } \exists \bar{s}' \text{ inwards} \\
& \exists y' \big(y' \not\approx \mathit{NULL} \wedge \textstyle\bigwedge_{s \in \bar{s}} \forall u, v (s(u,v) \Rightarrow u \not\approx y') \wedge \\
& \quad \textstyle\bigwedge_{s \in \bar{s}} \exists s' \forall u, v \big(s'(u,v) \Leftrightarrow u \approx y' \wedge v \approx \mathit{NULL} \vee s(u,v) \big) \wedge \\
& \quad \varphi[y'/y][u \approx y' \wedge v \approx \mathit{NULL} \vee \bar{s}(u,v)/\bar{s}(u,v)] \big) \\
\equiv \quad & \text{eliminate } \exists s' \ldots \\
& \exists y' \big(y' \not\approx \mathit{NULL} \wedge \textstyle\bigwedge_{s \in \bar{s}} \forall u, v (s(u,v) \Rightarrow u \not\approx y') \wedge \\
& \quad \varphi[y'/y][u \approx y' \wedge v \approx \mathit{NULL} \vee \bar{s}(u,v)/\bar{s}(u,v)] \big)
\end{aligned}$$

This ends the proof of Lemma 1. □

B Alternative Semantics of Allocation

In section 2, we have treated memory allocation as an unfailing operation, i. e., no out-of-memory errors can occur. This appendix presents an alternative semantics for memory allocation. It interprets $\mathit{new}_T()$ as a function which may non-deterministically return NULL to signal an out-of-memory error. Below, this new semantics $[\![\alpha]\!]^{\#}$ is defined for all actions α in terms of the old semantics $[\![\alpha]\!]$.

$$[\![\alpha]\!]^{\#} = \begin{cases} [\![y := \mathit{NULL}]\!] \vee [\![y := \mathit{new}_T()]\!] & \text{if } \alpha \text{ is of the form } y := \mathit{new}_T() \\ [\![\alpha]\!] & \text{otherwise} \end{cases}$$

Consequently, the weakest preconditions $\mathit{pre}^{\#}(\alpha; \varphi)$ corresponding to the new semantics can be stated in terms of the old weakest preconditions $\mathit{pre}(\alpha; \varphi)$. More precisely for all σ-formulas φ and all actions α, we have $\mathit{pre}^{\#}(\alpha; \varphi) = \mathit{pre}(\alpha; \varphi)$ except when α is of the form $y := \mathit{new}_T()$, where

$$\mathit{pre}^{\#}(y := \mathit{new}_T(); \varphi) = \mathit{pre}(y := \mathit{NULL}; \varphi) \vee \mathit{pre}(y := \mathit{new}_T(); \varphi) \ .$$

With the new semantics, Lemma 2 does not continue to hold. The reason is that the class BS_n, $n \geq 1$, is not closed under disjunction. Therefore, we introduce the class $BS_n^{\vee}$, $n \geq 0$, as the set of finite disjunctions $\varphi_1 \vee \ldots \vee \varphi_k$, $k \geq 1$, of formulas $\varphi_i \in BS_n$. As the following lemma shows, formulas in $BS_n^{\vee}$ are preserved under the new weakest preconditions.

Lemma 10. *Let P be a program, π a path in the ECFG and φ a σ-formula. If $\varphi \in BS_n$, $n \geq 2$, then $pre^{\#}(\pi; \varphi) \in BS_n^{\vee}$. The number of disjuncts in $pre^{\#}(\pi; \varphi)$ is in $O(2^{|\pi|} \cdot |\varphi|)$, but for each disjunct, the size is in $O(|\pi|^2 \cdot |\varphi|)$ and the length of the quantifier prefix is in $O(|\pi| + |\varphi|)$.*

Proof. That $pre^{\#}(\pi; \varphi)$ is in $BS_n^{\vee}$ follows from the above characterization of $pre^{\#}$ and the fact that pre distributes over disjunctions, i. e., $pre(\alpha; \varphi_1 \vee \varphi_2) = pre(\alpha; \varphi_1) \vee pre(\alpha; \varphi_2)$. The exponential blowup in the number of disjuncts is obvious. For the remaining claims, see the proof of Lemma 2. $\square$

The small model theorem (Theorem 8) can be extended to finite disjunctions, see below. Therefore, bounded model checking for pointer-safety is still decidable, even with the alternative semantics for modeling failed memory allocation.

Theorem 11. *Let $\phi_1, \ldots, \phi_k$ be k universal σ-formulas in BS_2, let $\mathcal{P}$ be an intersection-free TA-like monadic Datalog program and let Q be a query. If the formula $(\phi_1 \vee \ldots \vee \phi_k) \wedge \mathcal{P} \wedge Q$ is satisfiable then it has a model of cardinality at most doubly exponential in the size of the formula. Deciding satisfiability of such formulas is in 2-*NEXPTIME.

Proof. If $(\phi_1 \vee \ldots \vee \phi_k) \wedge \mathcal{P} \wedge Q$ is satisfiable then there exists i such that $\phi_i \wedge \mathcal{P} \wedge Q$ is satisfiable. Use Theorem 8. $\square$