RV'09, LNCS 5779, pp 132-151. 2009

Runtime Verification of C Memory Safety

Grigore Roșu[1], Wolfram Schulte[2], and Traian Florin Șerbănuță[1]

[1] University of Illinois at Urbana-Champaign
[2] Microsoft Research
grosu@cs.uiuc.edu, schulte@microsoft.com, tserban2@cs.uiuc.edu

Abstract. C is the most widely used imperative system's implementation language. While C provides types and high-level abstractions, its design goal has been to provide highest performance which often requires low-level access to memory. As a consequence C supports arbitrary pointer arithmetic, casting, and explicit allocation and deallocation. These operations are difficult to use, resulting in programs that often have software bugs like buffer overflows and dangling pointers that cause security vulnerabilities. We say a C program is memory safe, if at runtime it never goes wrong with such a memory access error. Based on standards for writing "good" C code, this paper proposes *strong memory safety* as the least restrictive formal definition of memory safety amenable for runtime verification. We show that although verification of memory safety is in general undecidable, even when restricted to closed, terminating programs, runtime verification of strong memory safety is a decision procedure for this class of programs. We verify strong memory safety of a program by executing the program using a symbolic, deterministic definition of the dynamic semantics. A prototype implementation of these ideas shows the feasibility of this approach.

1 Introduction

Memory safety is an crucial and desirable property for any piece of software. Its absence is a major source for software bugs which can lead to abrupt termination of software execution, but also, and sometimes even more dangerous, can be turned into a malicious tool: most of the recent security vulnerabilities are due to memory safety violations. Nevertheless most existing software applications, and especially performance-critical applications, are written in low-level programming languages such as C, which offer performance at the expense of safety. Due to C's support of pointer arithmetic, casting, and explicit allocation and deallocation, C program executions can exhibit memory safety violations ranging from buffer overflows, to memory leaks, to dangling pointers.

An important research question is thus the following:

Given a program written in an unsafe programming language like C, how can one guarantee that any execution of this program is memory-safe?

Many different approaches and tools were developed to address this problem. For instance, CCured [1] uses pointer annotations and analyzes the source of

the program, trying to prove it memory safe, introducing runtime checks in the code to monitor at runtime the parts which cannot be proven; Purify [2] and Valgrind [3] execute the program in a "debugging" mode, adding metadata to the program pointers to guarantee a proper separation of the allocation zones, and use that metadata to monitor and detect anomalies at runtime; DieHard [4] and Exterminator [5] replace the standard allocation routines by ones using randomization, which enables the detection of errors with high probability, attempting to correct the errors on-the-fly. However, most of these tools arise from ad-hoc observations and practical experience, without formally defining what it means for a program to be memory safe.

This paper makes a first step towards bridging this gap, by introducing a formal definition of memory safety for programs written in a non-memory safe programming language and execution platform. The language and platform chosen to this aim is KernelC, a formal definition of the executable semantics of *a fragment of the C language including memory allocation/freeing routines*. KernelC only supports one type namely mathematical integers; and each KernelC location can hold exactly one integer. Nevertheless one can write many interesting pieces of C code in KernelC. Here are some that we will refer to in the paper: `ALLOCATE` allocates a linked list of 5 nodes, in reversed order, each node having two contiguous locations, one holding a value and the other a pointer to the next node; `REVERSE` reverses a list of nodes as above that starts at `p`; and `DEALLOCATE` frees a list starting with `p`.

```
ALLOCATE
n = 0;
p = null;
while(n != 5) {
 q = malloc(2);
 *q = n;
 *(q+1) = p;
 p = q;
 n = n+1;
}
REVERSE
if(p != null) {
 x = *(p+1);
 *(p+1) = null;
 while(x != null) {
  y = *(x+1);
  *(x+1) = p;
  p = x;
  x = y;
 }
}
DEALLOCATE
while(p != null) {
 q = *(p+1);
 free(p);
 p = q;
}
```

Informally, memory safety means that the program cannot access a memory location which it shouldn't (e.g., exceeding arrays boundaries, addressing unallocated memory, and so on). For example, consider the program `ALLOCATE'`, obtained from `ALLOCATE` by removing the second statement, i.e., `p = null;`. Then, any of the composed programs `ALLOCATE' DEALLOCATE`, or `ALLOCATE' REVERSE`, is not memory safe, since the list can potentially be non-null terminated, which would lead to an attempt of accessing non-allocated memory upon deallocating/reversing the list. On the other hand, a compiler might initialize all local variables with 0 (which in C corresponds to `null`); if so our example program would have no memory access error and would terminate.

The principal source of C's non-determinism comes from the under-specification of *C's memory allocator*, which implements the `malloc` and `free` functions. The C language specification guarantees that a call to `malloc(` n `)` will, if it succeeds, return a pointer to a region of n contiguous and previously unallocated locations. These locations now become allocated. When these locations are no longer needed, the pointer, which was returned from a `malloc` call, is passed to `free` which deallocates the mem-

ory. The C language specification provides no guarantees except for that fact that `malloc` returns unallocated locations; `free` might deallocate the memory or not. To cope with this non-determinism, memory safety of KernelC programs is defined as a global property on the entire set of executions of a program, derivable using the KernelC definition. We say: A KernelC program is *memory safe* if none of its possible executions gets stuck in a non-final state.

One might expect that verification of memory safety would be decidable for terminating programs – after all we have so many checkers addressing the problem. However, we show that *memory safety is undecidable* even *for closed, terminating programs*. The argument for undecidability comes from the rather unusual usage of memory allocation, that is, using memory allocation as a source of non-determinism in the execution, such as the examples in Fig. 1: INPUT presents a simulation of non-deterministic input, and CHOICE shows how one can model non-deterministic choice.

INPUT	CHOICE
`n=malloc(1);`	`x = malloc(1);`
`while (n!=1)`	`y = malloc(1);`
`  if (n%2) n=3*n+1;`	`if (x<y) {}`
`  else n=n/2;`	`else {}`

Fig. 1. "Accidental" memory safety

Based on the fact that standards for writing "good" C code [6] advise against taking advantage of this kind of non-determinism, we define *strong memory safety as the least restrictive notion of memory safety amenable for runtime verification*. The runtime verification works as follows: we introduce SafeKernelC, an executable definition for the same language, except that it handles memory allocation symbolically; this makes the memory allocator deterministic; and since SafeKernelC is deterministic, there is only one possible execution of any program; therefore, strong memory safety can be monitored along that execution, obtaining a guarantee for memory safety for all possible (partial) executions of the program on the KernelC operational semantics. Note that strong memory safe programs do not only guarantee memory safety, they also enforce good coding practices and ensure platform portability.

Our contributions are as follows:

- We present KernelC, a formal definition for the dynamic semantics of a fragment of the C language, and formally define memory safety for an execution, and for a program, in this context. Since our definition is executable, it yields a procedure for runtime verifying memory safety along one possible execution of a program.
- We prove that, even if the input program is closed and known to be terminating, the verification of memory safety is undecidable.
- We refine KernelC to SafeKernelC, and introduce in this context strong memory safety as a meaningful restriction of memory safety.
- We prove that strong memory safety of closed programs can be effectively runtime verified. Since SafeKernelC is also executable, this gives us a sound semi-procedure for checking memory safety of KernelC programs.

The remainder of the paper is structured as follows. Section 2 introduces the KernelC definition and illustrates it through several examples. Section 3 for-

mally defines memory safety, and shows that, although we can monitor memory safety along any execution path, proving memory safety is generally undecidable. Section 4 introduces strong memory safety as a reasonable and decidable restriction of memory safety and shows that runtime verification of strong memory safety yields a sound technique for verifying memory safety. Section 5 concludes.

2 Formal Semantics of KernelC

We here discuss the definition of KernelC using K [7], a technique for defining languages within the Rewriting Logic Semantics [8, 9]. Within this framework, languages $\mathcal{L}$ are defined as rewrite theories $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$, where $\Sigma_{\mathcal{L}}$ is a signature extending the syntax of $\mathcal{L}$, $E_{\mathcal{L}}$ is a set of $\Sigma_{\mathcal{L}}$-equations, which are thought of as structural rearrangements preparing the context for rules and carrying no computational meaning, while $R_{\mathcal{L}}$ is a set of $\Sigma_{\mathcal{L}}$-rules, used to model irreversible computational steps. Since our base logic, rewriting logic [10], is a conservative extension of equational logic, terms can be replaced by equal terms in any context and in any direction. We write $\mathcal{R} \vdash t = t'$ whenever t can be proved equal to t' using equational deduction with the equations in $\mathcal{R}$. Like in term rewriting, rules can be applied in any context, but only from left-to-right. One way to think of rewriting logic is that equations apply until the term is *matched* by the left-hand-side (lhs) of some rule, which then irreversibly transforms the term. We write $\mathcal{R} \vdash t \rightarrow t'$ when t can be rewritten, using arbitrarily many equational steps but only one rewrite step in $\mathcal{R}$, into t'. Also, we write $\mathcal{R} \vdash t \rightarrow^* t'$ when t can be rewritten, using the equations and rules in $\mathcal{R}$, into t'. Rewriting logic thus captures *rewriting modulo equations* into a logic, with good mathematical properties (loose and initial models, complete deduction, proofs = computations, etc.). It is simple to understand and efficiently executable.

K is a *modular* definitional framework: rules match only what they need from the configuration, so one can change the configuration (e.g., adding store, input/output, stacks, etc.) without having to revisit existing rules.

Sequences, bags and maps. Sequences, bags and maps are core to K language definitions and are defined as standard (equational) data-structures. We use notations $\mathsf{Seq}_u^{\_@\_}[S]$ for sequences and $\mathsf{Bag}_u^{\_@\_}[S]$ for bags, resp., where u is their unit and $\_@\_$ is their binary construct. Formally, if added for sort S', these correspond to adding subsorting $S < S'$ (i.e., production $S' ::= S$, not needed when $S' = S$), operations $u :\rightarrow S'$ (a constant) and $\_@\_ : S' \times S' \rightarrow S'$, and appropriate unit and associativity equations for sequences, and unit, commutativity and associativity equations for bags. For example, an environment is a finite bag of pairs, $\rho[X]$ retrieves the *Int* associated to the *Id* X in ρ, $\rho[X \leftarrow J]$ updates the *Int* corresponding to X in ρ to J, and $\rho \backslash X$ removes pair $X \mapsto \_$ from ρ (if there is any). One can also define, in the same style, an operation *Dom* giving the domain of a map as a bag of elements, as well as an operation checking whether the map term is indeed a partial function. These operations are easy to define algebraically and therefore we assume them from here on; in fact, we assume that each map that

occurs in an equation or rule is a well-formed map (e.g., the maps $\sigma \circledast \sigma'$ in the rules for `malloc` and `free` in Fig. 2). In general, $\mathsf{Map}_u^{@}[S_1, S_2]$ corresponds to bags of pairs of elements of sorts S_1 and S_2, respectively, each pair written $s_1 \mapsto s_2$, with additional operations $\_[\_] : S' \times S_1 \to S_2$ and $\_[\_ \leftarrow \_] : S' \times S_1 \times S_2 \to S'$ and $\_\backslash\_ : S' \times S_1 \to S'$ for lookup, update (adding a new pair if map undefined on that element) and deletion (i.e., removing an element binding), respectively, where S' is the sort corresponding to the maps.

Nat ::= naturals, Int ::= integers	(abstract syntax)
Id ::= identifiers, to be used as variable names	
K ::= $Int \mid Id \mid$ `null` $\mid$ `*`$K \mid$ `!`$K \mid K_1$ `op` $K_2 \mid K_1$ `&&` $K_2 \mid K_1$ `||` $K_2 \mid$	
$\mid K_1$`=`$K_2 \mid K$`;` $\mid K_1 \; K_2 \mid$ `{`K`}` $\mid$ `{}` $\mid$ `malloc(`K`)` $\mid$ `free(`K`)`	
$\mid$ `if (`K_1`)` $K_2 \mid$ `if (`K_1`)` K_2 `else` $K_3 \mid$ `while (`K_1`)` K_2	
`null` = 0	(desugaring of non-core constructs)
`!` K = `if (`K`) 0 else 1`	
K_1 `&&` K_2 = `if (`K_1`)` K_2 `else 0`	K_1 `||` K_2 = `if (`K_1`) 1 else` K_2
`{`K`}` = K	`if (`K_1`)` K_2 = `if (`K_1`)` K_2 `else {}`
Cfg ::= $\langle \mathsf{Bag}[CfgItem] \rangle$	(configuration)
$CfgItem$ ::= $\langle K \rangle_k \mid \langle Env \rangle_{env} \mid \langle Mem \rangle_{mem} \mid \langle Ptr \rangle_{ptr}$	
K ::= ... $\mid \mathsf{Seq}^{\curvearrowright}[K]$	Mem ::= $\mathsf{Map}^{\circledast}[Nat^+, Int]$
Env ::= $\mathsf{Map}[Id, Int]$	Ptr ::= $\mathsf{Map}[Nat^+, Nat]$
`*`K = ($K \curvearrowright$ `*`$\square$)	(computation structural equations)
K_1 `op` K_2 = ($K_1 \curvearrowright \square$ `op` K_2)	I_1 `op` K_2 = ($K_2 \curvearrowright I_1$ `op` $\square$)
`if (`K_1`)` K_2 `else` K_3 = ($K_1 \curvearrowright$ `if (`$\square$`)` K_2 `else` K_3)	K`;` = $K \curvearrowright \square$`;`
(K_1`=`K_2) = ($K_2 \curvearrowright K_1$`=`$\square$)	(`*`K_1`=`K_2) = ($K_1 \curvearrowright$ `*`$\square$`=`K_2)
`malloc(`K`)` = ($K \curvearrowright$ `malloc(`$\square$`)`)	`free(`K`)` = ($K \curvearrowright$ `free(`$\square$`)`)
`{}` = $\cdot$	(semantic equations and rules)
$K_1 \; K_2 = K_1 \curvearrowright K_2$	I`;` $\to \cdot$
I_1 `op` $I_2 \to I_1 \; op_{Int} \; I_2$	
`if (`I`)` K_2 `else` $K_3 \to K_3$, where $I = 0$	`if (`I`)` K_2 `else` $K_3 \to K_2$, where $I \neq 0$
$\langle X \curvearrowright K \rangle_k \; \langle X \mapsto I, \rho \rangle_{env} \to \langle I \curvearrowright K \rangle_k \; \langle X \mapsto I, \rho \rangle_{env}$	$\langle X$`=`$I \curvearrowright K \rangle_k \; \langle \rho \rangle_{env} \to \langle K \rangle_k \; \langle \rho[X \leftarrow I] \rangle_{env}$
$\langle$`*`$P \curvearrowright K \rangle_k \; \langle P \mapsto I \circledast \sigma \rangle_{mem} \to \langle I \curvearrowright K \rangle_k \; \langle P \mapsto I \circledast \sigma \rangle_{mem}$	
$\langle$`*`P`=`$I \curvearrowright K \rangle_k \; \langle P \mapsto I' \circledast \sigma \rangle_{mem} \to \langle K \rangle_k \; \langle P \mapsto I \circledast \sigma \rangle_{mem}$	
$\langle$`while(`K_1`)`$K_2 \curvearrowright K \rangle_k = \langle$`if(`$K_1$`){`$K_2$`while(`$K_1$`)`$K_2$`}` $\curvearrowright K \rangle_k$	
$\langle$`malloc(`N`)` $\curvearrowright K \rangle_k \langle \sigma \rangle_{mem} \langle \pi \rangle_{ptr} \to \langle P \curvearrowright K \rangle_k \langle \sigma \circledast \sigma' \rangle_{mem} \langle \pi[P \leftarrow N] \rangle_{ptr}$ where $Dom(\sigma') = \overline{P, P+N-1}$	
$\langle$`free(`P`);` $\curvearrowright K \rangle_k \; \langle \sigma \circledast \sigma' \rangle_{mem} \; \langle P \mapsto N, \pi \rangle_{ptr} \to \langle K \rangle_k \; \langle \sigma \rangle_{mem} \; \langle \pi \rangle_{ptr}$ where $Dom(\sigma') = \overline{P, P+N-1}$	

(range of variables: $X \in Id; K, K_1, K_2 \in K; I, I_1, I_2 \in Int; P \in Nat^+; N \in Nat$)

Fig. 2. KernelC in K: Complete Semantics

Abstract Syntax. Fig. 2 shows the complete K definition of KernelC, a C-like language with dynamic memory allocation and deallocation. K definitions

typically use only one (abstract) syntactic category, K, serving as minimal syntactic infrastructure to define terms; it is not intended to be used for parsing or type-checking. We make no distinction between algebraic signatures and their context-free notation: syntactic categories correspond to sorts and productions to operations in the signature; for example, production "K ::= *Id*`=`K`;`" is equivalent to defining an operation "`_=_;` : $Id \times K \to K$". In Fig. 2, `op` stands for the various arithmetic and relational operations that one may want to include in one's language, and op_{Int} stands for the mathematical counterpart (function or relation) of `op` which operates on integers. For example, `op` can range over standard arithmetic operator names `+`, `-`, `*`, `/`, etc., and over standard relational operator names `==`, `!=`, `<=`, `>=`, etc., in which case $+_{Int}$ is the addition operation on integers (e.g., $3 +_{Int} 7 = 10$), etc., and $==_{Int}$ is the equality on integers (e.g., $(3 ==_{Int} 5) = 0$ and $(3 ==_{Int} 3) = 1$. Like in C, we assume that boolean values are special integer values, but, unlike C we assume unbounded integers.

We also assume the C meaning of the language constructs. In particular, `malloc(`N`)` allocates a block of N contiguous locations and returns a pointer to the first location, and `free(`P`)` assumes that a block of N locations has been previously allocated using a corresponding `malloc` and deallocates it.

Definition 1. *A* KernelC *computation K is* ***well-formed*** *iff it is equal (using equational reasoning within* KernelC*'s semantics) to a well-formed statement list or expression in C. Also, a computation is* ***well-terminated*** *iff it is equal to the unit computation "·" or to an integer value $I \in Int$.*

Syntactic Sugar. The desugaring equations are self-describing; we prefer to desugar derived language constructs wherever possible. The "boolean" constructs `&&` and `||` are shortcut. Even though the conditional is a statement, once all syntactic categories are collapsed into one, K, it can be used to desugar expression constructs as well.

Configurations. We use sequences, bags, maps and abstract syntax as configuration constructors, henceforth just called *cells*.

The configuration of KernelC is a top $\langle ... \rangle$ cell containing a "soup" of four sub-cells: a cell $\langle ... \rangle_k$ wrapping the computation; a cell $\langle ... \rangle_{env}$ holding the mapping for the stack variables; a cell $\langle ... \rangle_{mem}$ holding the memory (or heap) which can be dynamically allocated/deallocated; and a cell $\langle ... \rangle_{ptr}$ associating to pointers returned by `malloc` the number of locations that have been allocated (this info is necessary for the semantics of `free`).

K definitions achieve context-sensitivity in two ways: (1) by adding algebraic structure to configurations and using it to control matching; and (2) by extending the original language syntax with a special task sequentialization construct, "$\curvearrowright$" pronounced "then", as well as frozen variants of existing language constructs. Frozen operators have a "□" as part of their name and are used to "freeze" fragments of a program until their turn comes.

Definition 2. *Let (Σ, E) be the algebraic specification of* KernelC *configurations: Σ contains all the configuration constructs (for bags, maps, etc.) and E*

contains all their defining equations (associativities, commutativities, etc.). Let $\mathcal{T}$ be the Σ-algebra of ground terms; the E-equational classes (i.e., provably equal using equational reasoning with E) of (ground) terms in $\mathcal{T}$ of sort Cfg which have the form $\langle\langle K\rangle_k\,\langle\rho\rangle_{env}\,\langle\sigma\rangle_{mem}\,\langle\pi\rangle_{ptr}\rangle$ are called ***(concrete) configurations****. We distinguish several types of configurations:*

- *Configurations of the form $\langle\langle K\rangle_k\,\langle\cdot\rangle_{env}\,\langle\cdot\rangle_{mem}\,\langle\cdot\rangle_{ptr}\rangle$ where K is a well-formed computation, also written more compactly $[\![K]\!]$, are called* ***initial configurations****;*
- *Configurations $\langle\langle K\rangle_k\,\langle\rho\rangle_{env}\,\langle\sigma\rangle_{mem}\,\langle\pi\rangle_{ptr}\rangle$ whose embedded computation K is well-terminated (a "$\cdot$" or an $I \in Int$) are called* ***final configurations****;*
- *Configurations $\gamma \in \mathcal{T}$ which cannot be rewritten anymore (i.e., there is no configuration $\gamma' \in \mathcal{T}$ such that* KernelC $\vdash \gamma \rightarrow \gamma'$*) are* ***normal form configurations****;*
- *Normal form configurations which are not final are called* ***stuck (or junk, or core dump) configurations****;*
- *Configurations γ which cannot be rewritten infinitely (i.e., there is no infinite set of configurations $\{\gamma_n\}_{n\in Nat}$ such that $\gamma_0 = \gamma$ and* KernelC $\vdash \gamma_n \rightarrow \gamma_{n+1}$ *for any $n \in Nat$) are called* ***terminating configurations****.*

Computation structures. Sort K contains computation structures, or simply *computations*, obtained by adding to the original abstract syntax *computation sequences* (terms in $\mathsf{Seq}_{\curvearrowright}[K]$) and *frozen computations* (wrapped by operators containing a "□" in their name). Intuitively, $K_1 \curvearrowright K_2$ means "first process K_1, *then* process K_2". Frozen computations are structurally inhibited from advancing until their turn comes. For example, "K_1 `op` K_2" first processes K_1 and in the meanwhile keeps K_2 frozen: "K_1 `op` $K_2 = K_1 \curvearrowright \square$ `op` K_2". After K_1 is processed, its result is placed back in context and K_2 is "scheduled": "I_1 `op` $K_2 = K_2 \curvearrowright I_1$ `op` $\square$". As equations, these can be applied forth (to "schedule" for processing) and back (to "plug" results back). We assume all freezing operators are automatically added to sort K (i.e., the "..." in "K ::= ..." in Fig. 2 include "Id=□;" and "□ `op` K | K `op` □" for all operations `op` that one specifies in the language syntax). Computation equations give the evaluation strategy of each language construct; note the one for the conditional, which schedules for processing the condition, keeping the two branches frozen. They accomplish the same role as the context productions of evaluation contexts [11], but logically rather than syntactically.

Semantic equations and rules. Empty blocks and sequential composition are dissolved into the unit and the sequentialization of K. The rules for `+`, `==` and `if` are clear. The rule for variable assignment updates the environment, at the same time dissolving the assignment statement. We chose to let lookup of uninitialized variables be undefined. Pointer lookup and update are similar, replacing the environment by memory.

The equation of `while` shows a use of the cell structure to achieve context sensitivity; if replacing it with the simple-minded equation (or rule in case one prefers to regard loops unrolling as a computational step) `while(`K_1`)`K_2 =

`if(`K_1`){`K_2`;while(`K_1`)`K_2`}` then there is nothing to prevent the application of this equation again on the while term inside the conditional, and so on. While proof-theoretically one could argue that there is no problem with that, operationally it is problematic as it leads to operational non-termination even though the program may terminate. Therefore, we chose to restrict the unrolling of `while` to only the cases when `while` is the first computation task.

The rules for `free` and `malloc` make subtle use of matching modulo associativity and commutativity of $\_\circledast\_$. In the case of `free(`P`)`, a σ' is matched in the $\langle ... \rangle_{mem}$ cell whose domain is the N contiguous locations $\overline{P, P+N-1}$, where N is the natural number associated to P in the $\langle ... \rangle_{ptr}$ cell (i.e., the number of locations previously allocated at P using a `malloc`); then the `free` statement in cell $\langle ... \rangle_k$, the memory map σ' in cell $\langle ... \rangle_{mem}$ and the pointer mapping $P \mapsto N$ in cell $\langle ... \rangle_{ptr}$ are discarded; this way, the memory starting with location P can be reclaimed and reused in possible implementations of KernelC. Recall that we assume that all (partial) maps appearing in any context are well-formed; in particular, the map $\sigma \circledast \sigma'$ in the rule of `free` is well-formed, which means that there is only one such matching in the memory cell (P and N are given), which means that the rule for `free` is deterministic. Such a compact and elegant definition is possible only thanks to the strength of matching and rewriting modulo equations. Maude [12] provides efficient support for these operations, which is what makes it a very convenient execution vehicle for K. The well-formedness of maps can either be assumed (one can prove aside that each equation/rule preserves it) or checked as a condition attached to the rule. Fig. 3 shows a rewriting logic derivation using the K semantics in Fig. 2; $\rightarrow^*$ stands for one or more rewrite steps, with arbitrarily many equational steps in between.

The most intricate rule in Fig. 2 is that of `malloc` which is an almost exact dual of the rule for `free`. Like in the `free` rule, the σ' is doubly constrained: its domain is disjoint from σ's (because $\sigma \circledast \sigma'$ is well-formed) and its domain is the set of contiguous locations $\overline{P, P+N-1}$ with P the returned pointer. However, the constraints on σ' are loose enough to allow a high degree of semantic non-determinism. E.g., program "`BAD` ≡ `p=malloc(2);*2=7;`" may exhibit three different types of behavior, two in which it terminates normally but in non-isomorphic configurations, and one in which it gets stuck looking up for location 1 which is not allocated. E.g., $\langle\langle \texttt{BAD} \rangle_k \langle \cdot \rangle_{env} \langle \cdot \rangle_{mem} \langle \cdot \rangle_{ptr} \rangle$ rewrites to any of the following (each being a normal form):

$\langle\langle \cdot \rangle_k \ \langle \texttt{p} \mapsto 1 \rangle_{env} \ \langle (1 \mapsto i) \circledast (2 \mapsto 7) \rangle_{mem} \ \langle 1 \mapsto 2 \rangle_{ptr} \rangle$, where $i \in Int$
$\langle\langle \cdot \rangle_k \ \langle \texttt{p} \mapsto 2 \rangle_{env} \ \langle (2 \mapsto 7) \circledast (3 \mapsto j) \rangle_{mem} \ \langle 2 \mapsto 2 \rangle_{ptr} \rangle$, where $j \in Int$
$\langle\langle *2 \curvearrowright \square\texttt{=7;} \rangle_k \langle \texttt{p} \mapsto 5 \rangle_{env} \langle (5 \mapsto k) \circledast (6 \mapsto -3) \rangle_{mem} \langle 5 \mapsto 2 \rangle_{ptr} \rangle$, where $k \in Int$

In concrete implementations of KernelC, one may see the last type of behavior more frequently than the other two, as it is unlikely that `malloc` allocates at the "predicted" location, 2 in our case. We tried this code in `gcc` on a Linux machine (casting 2 to `(T*)2`) and it compiled (but it gave an expected segmentation fault when run). Thus, we can regard the third normal form term above as a "core dump".

Let `REVERSE` be the list reverse program in Introduction, and let
`WHILE ≡ while(x!=null){y=*(x+1);*(x+1)=p;p=x;x=y;}`
`IF ≡ if(□){x=*(p+1); *(p+1)=null; WHILE}`.
Also, let us assume the environment and memory maps: $(\rho_1 \equiv p \mapsto 1, x \mapsto 0, y \mapsto 0)$, $(\rho_2 \equiv p \mapsto 1, x \mapsto 5, y \mapsto 0)$, $(\rho_3 \equiv p \mapsto 5, x \mapsto 0, y \mapsto 0)$, $(\sigma_1 \equiv 1 \mapsto 7 \circledast 2 \mapsto 5 \circledast 5 \mapsto 9 \circledast 6 \mapsto 0)$, $(\sigma_2 \equiv 1 \mapsto 7 \circledast 2 \mapsto 0 \circledast 5 \mapsto 9 \circledast 6 \mapsto 0)$, $(\sigma_3 = 1 \mapsto 7 \circledast 2 \mapsto 0 \circledast 5 \mapsto 9 \circledast 6 \mapsto 1)$. The following derivation shows an execution reversing a list with the elements 7, 9:

$\langle$`REVERSE(p)`$\rangle_k \langle\rho_1\rangle_{env}\langle\sigma_1\rangle_{mem} = \langle$`p!=null`$\curvearrowright$`IF`$\rangle_k \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} =$

$\langle$`!(p==null)` $\curvearrowright$ `IF`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} =^*$

$\langle$`if (p==0) 0 else 1` $\curvearrowright$ `IF`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} =$

$\langle$`p==0` $\curvearrowright$ `if (□) 0 else 1` $\curvearrowright$ `IF`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} =$

$\langle$`p` $\curvearrowright$ `□==0` $\curvearrowright$ `if (□) 0 else 1` $\curvearrowright$ `IF`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} \rightarrow$

$\langle$`1` $\curvearrowright$ `□==0` $\curvearrowright$ `if (□) 0 else 1` $\curvearrowright$ `IF`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} =$

$\langle$`1==0` $\curvearrowright$ `if (□) 0 else 1` $\curvearrowright$ `IF`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} \rightarrow$

$\langle$`0` $\curvearrowright$ `if (□) 0 else 1` $\curvearrowright$ `IF`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} =$

$\langle$`if (0) 0 else 1` $\curvearrowright$ `IF`$\rangle_k \langle\rho_1\rangle_{env}\langle\sigma_1\rangle_{mem} \rightarrow \langle$`1` $\curvearrowright$ `IF`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} \rightarrow^*$

$\langle$`x=*(p+1);` $\curvearrowright$ `*(p+1)=0;` $\curvearrowright$ `WHILE`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} \rightarrow^*$

$\langle$`x=*2;` $\curvearrowright$ `*(p+1)=0;` $\curvearrowright$ `WHILE`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} \rightarrow^*$

$\langle$`x=5;` $\curvearrowright$ `*(p+1)=0;` $\curvearrowright$ `WHILE`$\rangle_k \quad \langle\rho_1\rangle_{env} \langle\sigma_1\rangle_{mem} \rightarrow^*$

$\langle$`*(p+1)=0;` $\curvearrowright$ `WHILE`$\rangle_k \quad \langle\rho_2\rangle_{env} \langle\sigma_1\rangle_{mem} \rightarrow^*$

$\langle$`if (x!=0) {y=*(x+1);*(x+1)=p;p=x;x=y;WHILE}`$\rangle_k \quad \langle\rho_2\rangle_{env} \langle\sigma_2\rangle_{mem} \rightarrow^*$

$\langle$`y=*(x+1);` $\curvearrowright$ `*(x+1)=p;p=x;x=y;WHILE`$\rangle_k \quad \langle\rho_2\rangle_{env} \langle\sigma_2\rangle_{mem} \rightarrow^*$

$\langle$`*(x+1)=p;` $\curvearrowright$ `p=x;` $\curvearrowright$ `x=y;` $\curvearrowright$ `WHILE`$\rangle_k \quad \langle\rho_2\rangle_{env} \langle\sigma_2\rangle_{mem} \rightarrow^*$

$\langle$`p=x;` $\curvearrowright$ `x=y;` $\curvearrowright$ `WHILE`$\rangle_k \quad \langle\rho_2\rangle_{env} \langle\sigma_3\rangle_{mem} \rightarrow^*$

$\langle$`if (x!=0) {y=*(x+1);*(x+1)=p;p=x;x=y;WHILE}`$\rangle_k \quad \langle\rho_3\rangle_{env} \langle\sigma_3\rangle_{mem} \rightarrow^*$

$\langle\cdot\rangle_k \langle\rho_3\rangle_{env} \langle\sigma_3\rangle_{mem}$

Fig. 3. Rewriting logic derivation using the KernelC semantics in Fig. 2.

We claim that, in spite of this apparently undesired non-determinism, this is the most general semantics of `malloc` that a language designer may want to have. Any other additional constraints, such as "always allocate a fresh memory region", or "always reuse existing memory if possible", etc., may lead to a restrictive definition of KernelC, possibly undesired by some implementors. The actual C language makes no specific requirements on memory allocation, allowing C interpreters or compilers freedom to choose among various memory allocation possibilities; it is programmers' responsibility to write programs that do not rely on particular memory allocation strategies. Note that, for simplicity, our semantics abstracts from the fact that `malloc` can fail, in which case a `null` pointer is returned; that is similar to saying we assume an unbounded memory.

The language. We can now formally state what KernelC is:

Definition 3. *The language* KernelC *discussed here is the rewrite logic theory* $(\Sigma_{\text{KernelC}}, E_{\text{KernelC}}, R_{\text{KernelC}})$ *depicted in Fig. 2. If* KernelC $\vdash \gamma \rightarrow^* \gamma'$ *we say that, in* KernelC*, configuration* γ ***rewrites to*** *configuration* γ'*.*

Both the abstract syntax of KernelC and Σ are included in Σ_{KernelC}, and also both the desugaring equations of derived KernelC constructs and E are

included in E_{KernelC}; recall from Definition 2 that (Σ, E) is the equational definition of KernelC configurations.

Therefore, the rewrite logic semantics of KernelC, identified with KernelC from here on, can produce by means of rewriting all the possible complete or intermediate executions that the language can yield. In particular, if KernelC $\vdash [\![K]\!] \rightarrow^* \gamma$ with K a well-formed computation and γ a well-terminated configuration, then γ contains the (possibly non-deterministic) "result" obtained after "evaluating" K. In addition to comprising all the good executions, the rewrite theory KernelC also comprises all the bad executions of KernelC programs, namely all those that can get stuck; as seen shortly, this is very important as it will allow us to formally define memory safety of KernelC programs.

Note that like in any other formal operational semantics, our rewrite logic definition of KernelC has the property that informal execution steps and whole executions of programs become, respectively, formal proof steps and whole proofs in rewriting logic. Interestingly, unlike in other operational semantic frameworks, rewriting logic also provides models which are complete for its proof system, so the very same K definition of KernelC is also a loose "denotational" semantics in addition to being an "operational" one; moreover, since rewriting logic admits initial models, which are essentially built as a fix point over the algebra of terms, there is a selected subset of models, the "reachable" ones, for which induction is valid. In other words, once one has a K definition of a language, one needs no other formal semantics of that language because its K definition already provides everything one may need from a formal semantics. This is also one of the reasons for which we call K semantics *executable* rather than operational; the latter may give the wrong impression that the K semantics can only be used to yield an interpreter for the language.

Even though K is executable by its very nature, here we actually *defined*, and not *implemented*, KernelC. We therefore wanted to keep our semantics as loose, or unconstrained, as possible. As usual, when implementing non-deterministic specifications one needs not (and typically does not) provide all the non-deterministic behaviors in one's implementation. In fact, each implementation of KernelC is expected to be deterministic. The non-determinism of `malloc` in our KernelC definition is a result of a deliberate language *under-specification*, not a desired non-deterministic feature of the language. General details on under-specification versus non-determinism are beyond our scope here, but the interested reader is referred to [13] for an in-depth discussion on these subjects. An additional advantage of the under-specified `malloc` in our definition of KernelC is that it allows us to elegantly yet rigorously define memory safety in the next section: a program is memory-safe iff it cannot get stuck, i.e., it cannot be rewritten to a normal form whose computation cell is not well-terminated.

3 Memory Safety

We here give a formal definition to *memory safety* in KernelC, capturing the intuition that a program is memory safe iff it is so under any possible imple-

mentation of KernelC, i.e., under any possible choice the rule for `malloc` may make. Due to the undecidability of termination in general, our notion of memory safety, like any other practical (i.e., not unreasonably restricted) notion of memory safety, is undecidable in general. In this section we show that memory safety is actually undecidable even for terminating KernelC programs. That means, in particular, that KernelC semantics as well as any faithful implementation of it, cannot detect memory safety violations even on programs which always terminate, no matter whether that is attempted statically or at runtime.

To check memory safety, one therefore needs either to rely on user help (e.g., annotations), as detailed in [14], or to restrict the class of memory safe programs, which is what we do in next section.

Definition 4. *Well-formed computation K is* ***terminating*** *iff $[\![K]\!]$ is a terminating configuration in* KernelC*, and is* ***memory safe*** *iff any normal form of $[\![K]\!]$ in* KernelC *is final.*

Program "`BAD ≡ p=malloc(2);*2=7;`" is terminating but not memory safe: $[\![\texttt{BAD}]\!]$ rewrites, as seen, to normal form $\langle\langle \texttt{*2} \curvearrowright \Box\texttt{=7;}\rangle_k \langle \texttt{p} \mapsto 5\rangle_{env} \langle (5 \mapsto -1) \circledast (6 \mapsto -3)\rangle_{mem} \langle 5 \mapsto 2\rangle_{ptr}\rangle$. Program "`GOOD ≡ p=malloc(2);*(p+1)=7;`", on the other hand, is both terminating and memory-safe: $[\![\texttt{GOOD}]\!]$ rewrites only to normal form configurations of the form $\langle\langle\cdot\rangle_k\ \langle \texttt{p} \mapsto i\rangle_{env}\ \langle (i \mapsto j)\ \circledast\ (i{+}1 \mapsto 7)\rangle_{mem}\ \langle i \mapsto 2\rangle_{ptr}\rangle$, where $i \in Nat^+$ and $j \in Int$. Program "`p=malloc(1);while(*p){}`" is memory safe but not terminating (when `*p` $\neq 0$), and finally, program "`p=malloc(1);while(*1){}`" is neither memory-safe (when `p` $\neq 1$) nor terminating (when `p` $= 1$ and `*1` $\neq 0$).

For our simple language, memory is the only source of unsafety; for more complex languages, one may have various types of safety, depending upon the language construct at the top of the computation in t when t is a normal form, which tells why the computation got stuck; e.g., if the language has division and $3/0$ is at the top of the computation, then K got stuck because a division by zero was attempted.

KernelC is Turing complete (we assumed both arbitrarily large integers and infinite memory), so termination of KernelC programs is undecidable. That immediately implies that memory safety is also undecidable in general: for any memory safe program `PGM`, the program "`PGM;BAD`" is memory safe iff `PGM` does not terminate. What is not so obvious is the decidability or undecidability of memory safety on terminating programs. In the remainder of this section we show that this is actually an undecidable problem.

A hasty reader may think that, since programs have no symbolic inputs or data, memory safety must be decidable on terminating programs: one can simply run the program and check each memory access. The complexity of the problem comes from the non-determinism/under-specification of `malloc`, which makes any particular execution of the program to mean close to nothing wrt memory safety. Consider, for example, an execution of the program "`x=malloc(1); free(x); y=malloc(1); *x=1;`" in which the second `malloc` *just happens* to return the same pointer as the first `malloc`. Since this particular execution taking place on a hypothetical particular implementation of KernelC terminates normally, one may be wrongly tempted to say that it is memory safe; this program

is clearly *not* memory safe (gets stuck if second `malloc` chooses a different location) and even the execution itself can be argued as memory unsafe, because of a memory leak on `x` (dangling pointer).

Proposition 1. *Memory safety of terminating* KernelC *programs is an undecidable property.*

Proof. Since KernelC is Turing complete, we can encode any decidable property $\varphi(n)$ of input $n \in Nat$ as a *terminating* and *memory-safe* KernelC program "$\texttt{x=}n\texttt{;PGM}_\varphi$" which writes some variable `out`, such that $\varphi(n)$ holds iff KernelC $\vdash [\![\texttt{x=}n\texttt{;PGM}_\varphi]\!] \rightarrow^* \langle\langle\cdot\rangle_k\langle \texttt{out} \mapsto 1, ...\rangle_{env}...\rangle$ and $\varphi(n)$ does not hold iff KernelC $\vdash [\![\texttt{x=}n\texttt{;PGM}_\varphi]\!] \rightarrow^* \langle\langle\cdot\rangle_k\langle \texttt{out} \mapsto 0, ...\rangle_{env}...\rangle$. Since the pointer returned by `malloc` is non-deterministic, we can use it to "choose a random" n to assign to `x`: consider the program "$\texttt{PGM}'_\varphi \equiv \texttt{x=malloc(1);PGM}_\varphi\texttt{;if(out)GOOD else BAD}$". $\texttt{PGM}'_\varphi$ terminates because "$\texttt{x=}n\texttt{;PGM}_\varphi$" terminates for any $n \in Nat$ returned by `malloc(1)` and the conditional always terminates. On the other hand, $\texttt{PGM}'_\varphi$ is memory safe iff the variable `out` is 1 in the environment when $\texttt{PGM}_\varphi$ terminates, which happens iff $\varphi(n)$ holds for all $n \in Nat$. The undecidability of memory safety then follows from the fact that there are decidable properties φ for which $(\forall n)\varphi(n)$ is a proper co-recursively-enumerable property [15].

Since our notion of memory safety refers to a program rather than a path, the proposition above says that it is also impossible to devise any runtime checker for memory safety of general purpose KernelC (and hence C) programs. One could admittedly argue that such anomalies occur as artifacts of poorly designed languages like C, that allow for (too) direct memory access and complete freedom in handling pointers as if they are natural numbers. However, it is actually precisely these capabilities that make C attractive when performance is a concern, and performance is indeed a concern in many applications. That memory unsafe programs may execute just fine is a *must* feature of any formal semantic definition of C that is worth its salt, because all C implementations deliberately "suffer" from this problem.

Since unrestricted use of pointers returned by `malloc` can lead to non-deterministic executions of programs, one could, in principle, introduce some notion of "path memory safety". For example, one could argue that an execution of the program "`x=malloc(1); y=malloc(1); if (y==x+1) {} else BAD`" in which `y` just happens to be `x+1` is memory safe, or that an execution of the program "`x=malloc(2); if (*x==*(x+1)) {} else BAD`" in which `*x` just happens to be `*(x+1)` is memory safe. Encouraged by the informal so-called "C rules for pointer operations" [6], we prefer to not introduce such a notion of "path memory safety" and, instead, to keep our notion of memory safety of programs in Definition 4; with it, these terminating programs are not memory safe. We will next introduce a stronger notion of memory safety, supported by an executable semantics that will always get stuck on these programs.

4 Strong Memory Safety

We propose the semantic notion of *strong memory safety*: a program is strongly memory safe iff it does not get stuck in the executable semantics SafeKernelC, a variant of KernelC semantics with symbolic pointers. Interestingly, our formal definition of strong memory safety includes the informal notion of memory safety implied by the "C rules for pointer operations" [6]. Strong memory safety is shown decidable for terminating programs, but, of course, it is undecidable in general.

Note that we are not attempting to fix C's problems here, nor to propose a better language design. However, the high degree of non-determinism in the semantics of `malloc` may be problematic in formal verification. We prefer to give a slightly different semantics to our language, one which captures the non-determinism of `malloc` *symbolically*. Fig. 4 shows the formal K semantic definition of SafeKernelC, which essentially adds symbolic numbers and gives `malloc` a symbolic semantics. Everything else stays unchanged, like in the definition of KernelC in Fig. 2.

The first distinction between KernelC and SafeKernelC is that, although both of them are deterministic for all rules except the `malloc` rule, KernelC can introduce non-determinism based on the values returned by the `malloc` function, while SafeKernelC, using symbolic values, is deterministic up to symbolic variable renaming.

Proposition 2. *For any* $\mathcal{L} \in \{\text{KernelC}, \text{SafeKernelC}\}$*, if* $\mathcal{L} \vdash \gamma_0 \rightarrow^* \gamma$ *such that* $\mathcal{L} \vdash \gamma = \langle\langle K \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \langle \pi \rangle_{ptr} \rangle$*, and* $\mathcal{L} \not\vdash K = \texttt{malloc(}N\texttt{);} \curvearrowright K'$*, then there exists at most one configuration* γ' *such that* $\mathcal{L} \vdash \gamma \rightarrow \gamma'$*.* SafeKernelC *is deterministic, modulo renamings of the symbols from NatVar.*

Proof. First part can be formally proved by induction on the length of the derivation. Intuitively, the property holds because, at any moment, there exists a unique way to match a configuration which is reachable from an initial state, and the rules preserve this invariant. Moreover, except for the `malloc` rule, which allows for a choice of the value introduced (but does not violate the invariant), all other rules have the variables in the right hand side completely determined by those in the left hand side. For the second part, since the value introduced by the `malloc` rule is only operated with symbolically, which corresponds to the fact that future side conditions must hold for all possible valuations of variable, it follows that we can choose a canonical way to generate fresh symbols and thus completely eliminate the non-determinism.

Therefore, we can choose a representative derivation for any SafeKernelC derivation of an initial state, say one in which choosing of fresh variables is done in order from the countable infinite sequence $nv_1, nv_2, \ldots, nv_N, \ldots$.

Definition 5. *Well-formed computation* K *is* ***strongly terminating*** *iff* $[\![K]\!]$ *is terminating in* SafeKernelC*, and is* ***strongly memory safe*** *iff any normal form of* $[\![K]\!]$ *in* SafeKernelC *is final.*

Since SafeKernelC adds symbolic values (for pointers and initial values in allocated memory locations), the assumed machinery for naturals and integers is now expected to work with these symbolic values as well. In particular, the rule (side) conditions may be harder to check. For example, the rule "I_1==$I_2 \rightarrow N$" applies only when one proves that $I_1 = I_2$, and in that case N is 1, or when one proves that $I_1 \neq I_2$, and in that case N is 0; if one cannot prove any of the two, then the term "I_1==I_2" remains unreduced and the execution of the program may get stuck because of that. For example, both "`p=malloc(1);while(*p){}`" and "`p=malloc(1);while(*1){}`" are now strongly terminating (but remain memory unsafe, also in the strong sense). Also, both programs discussed in front of Proposition 1 get stuck when processing the conditions of their `if` statements. On the positive side, programs obeying the recommended safety rules for pointer operations in C [6], e.g., reading only initialized locations and comparing pointers only if they are within the same data-structure contiguously allocated in memory, are strongly memory safe. For example, "`n=100;a=malloc(n);x=a;while(x!=a+n){*x=0;x=x+1;}`" is both strongly memory safe and strongly terminating.

Since the side conditions can get arbitrarily complicated (they depend on the program), the problem of deciding their validity itself can potentially become undecidable. Therefore, we will assume an oracle for the logic involving the side conditions, which, given a formula, can give one of the following answers: YES, if the formula holds, NO, if it does not hold, or MAYBE, if the oracle cannot decide the problem. For example, this oracle could be a sound automatic theorem prover, which will attempt to prove/disprove the theorem, but, being incomplete, might also fail on complex formulas. The oracle we used together with our definition of SafeKernelC is very simple and based only on simplification rules (see Appendix C). However, since the logic (regarding pointer comparison) involved in programs following "good coding standards" should be relatively simple, we believe this oracle will probably act as a decision procedure for the majority of code. Matching logic [14] shows how one could use the same framework (together with code annotations), to prove general purpose properties about programs, which would allow checking memory safety for "bad" coded programs, as well. The advantage of the technique presented here is that it is fully automatic and requires no additional user input.

$NatVar$ ::= infinite set of symbolic natural numbers	(abstract syntax)
Nat ::= ... | $NatVar$	
	(semantic equations and rules)
$\langle \texttt{malloc}(N) \curvearrowright K \rangle_k \langle \sigma \rangle_{mem} \langle \pi \rangle_{ptr} \rightarrow \langle P \curvearrowright K \rangle_k \langle \sigma \circledast \sigma' \rangle_{mem} \langle \pi[P \leftarrow N] \rangle_{ptr}$	
where P is a fresh symbol in $NatVar$ and $Dom(\sigma') = \overline{P, P+N-1}$	

Fig. 4. Formal semantics of SafeKernelC.
(figure only shows how it differs from the semantics of KernelC in Fig. 2)

Proposition 3. *Any execution of a program p in* SafeKernelC*, can be step-by-step simulated in* KernelC*.*

Proof. Suppose $(\gamma_i)_{i\geq 0}$ is a sequence of configurations satisfying that γ_0 is an initial configuration corresponding to p, and, for any i, SafeKernelC $\vdash \gamma_i \xrightarrow{\theta_{i+1}(\rho_{i+1})} \gamma_{i+1}$. Also suppose this derivation sequence is representative, in the sense that fresh *NatVar* symbols are generated (in order) from the sequence $(nv_j)_{j\geq 1}$. Let $(i_j)_{j\geq 1}$ be a sequence of positive numbers such that ρ_{i_j} is the instance of the `malloc` rule introducing nv_j. If follows that, for any $i \leq i_{j_0}$, γ_i is completely determined by $(nv_j)_{1\leq j\leq j_0}$; that is, all values in the environment and store are algebraic expressions with variables from $(nv_j)_{1\leq j\leq j_0}$. We define the following sequence of functions $V_j : \{nv_1, \ldots, nv_j\} \to Nat$, by: $V_1(nv_1) = 1$, and $V_{n+1}(nv_j) = \begin{cases} V_n(nv_j), j \leq n \\ 1 + V_n(nv_n) + \overline{V_n}(N), j = n+1 \end{cases}$, where N is the number associated to nv_n in γ_{i_n}, and $\overline{V_n}$ is the canonical extension of V_n to expressions and configurations (mapping $\sigma(nv_j)$ to 0). Let V be the limit of $(V_j)_{j\geq 1}$, i.e., $V(nv_j) = V_j(nv_j)$. It remains to show that, for any $i \geq 0$, $\overline{V}(\gamma_i)$ is a configuration for KernelC, and KernelC $\vdash \overline{V}(\gamma_i) \xrightarrow{\overline{V}(\theta_{i+1}(\rho_{i+1}))} \overline{V}(\gamma_{i+1})$ (whence $\overline{V}(\theta_{i+1}(\rho_{i+1}))$ is an instance of a KernelC rule). For $i = 0$, $\overline{V}(\gamma_0) = \gamma_0$, since γ_0 does not contain any *NatVar* symbols, whence it also is a configuration for KernelC. Now, suppose $\overline{V}(\gamma i)$ is a KernelC configuration, and that SafeKernelC $\vdash \gamma_i \xrightarrow{\theta_{i+1}(\rho_{i+1})} \gamma_{i+1}$. If ρ_{i+1} is not the `malloc` rule, then it basically is the same rule schema as in KernelC, with the difference that it can be instantiated for terms with symbols from *NatVar*. Since the *NatVar* symbols are free variables in the instance $\theta_{i+1}(\rho_{i+1})$, it follows that $\overline{V}(\theta_{i+1}(\rho_{i+1}))$ is an instance for both KernelC and SafeKernelC; therefore it must be that KernelC $\vdash \overline{V}(\gamma_i) \xrightarrow{\overline{V}(\theta_{i+1}(\rho_{i+1}))} \overline{V}(\gamma_{i+1})$. Similarly, if $\theta_{i+1}(\rho_{i+1})$ is an instance of the `malloc` rule, then $\overline{V}(\theta_{i+1}(\rho'_n))$ is an instance of the `malloc` rule in KernelC (by the construction of V).

Theorem 1. *Let ϵ = SafeKernelC $\vdash \gamma_0 \to^n \gamma_n$ be a prefix of the execution of a program p using* SafeKernelC*. Then any execution of p using* KernelC *has a prefix of length n which is a valuation of ϵ.*

Proof. Let SafeKernelC $\vdash \gamma_i \xrightarrow{\theta_{i+1}(\rho_{i+1})} \gamma_{i+1}$, $0 \leq i < \overline{n}$, where $\overline{n} \in Nat \cup \{\infty\}$, be the (possibly infinite) canonical derivation of γ_0 in SafeKernelC. Let KernelC $\vdash \gamma'_i \xrightarrow{\theta'_{i+1}(\rho'_{i+1})} \gamma'_{i+1}$, $0 \leq i < \overline{n'}$ be a derivation such that $\gamma'_0 = \gamma_0$. Let $(i_j)_{j\geq 1}$ be the increasing sequence of positive numbers such that ρ'_{i_j} is an instance of the `malloc` rule. We then let $V(nv_j) = \theta'_{i_j}(P)$. It can be proved by induction on i that $\overline{V}(\theta_i(\rho_i)) = \theta'_i(\rho_i)$, and therefore $\overline{V}(\gamma_i) = \gamma_i$. Moreover, if $\overline{n'} < \overline{n}$, then, using a construction similar to the one used in proving Proposition 3, we can expand the existing valuation V to one covering all symbols in the SafeKernelC derivation, say V', and use that to expand the existing KernelC derivation to KernelC $\vdash \overline{V'}(\gamma_i) \xrightarrow{\overline{V'}(\theta_{i_1}(\rho_{i+1}))} \overline{V'}(\gamma_{i+1})$, $\overline{n'} \leq i < \overline{n}$, such that $\overline{V'}(\gamma_{\overline{n'}}) = \gamma'_{\overline{n'}}$.

By showing that all KernelC executions are abstracted by the SafeKernelC deterministic execution of a program (as long as that can proceed),Theorem 1 gives us an effective procedure for runtime verification of memory safety.

Theorem 2. *1. Strong memory safety guarantees memory safety.*
2. Monitoring SafeKernelC *executions yields a sound procedure for runtime verification of* KernelC *memory safety.*

Proof. 1. Assume p is a strongly safe program and let γ_0 be an initial configuration for p. This means that either (1) the execution of p using SafeKernelC is non-terminating, or (2) there exists n such that SafeKernelC $\vdash \gamma_0 \rightarrow^n \gamma_n$ is final. If (1) is true, then for any prefix SafeKernelC $\vdash \gamma_0 \rightarrow^n \gamma_n$ of the execution of n, all KernelC executions of p must be of length at least n. Since n is arbitrarly large, this implies that all KernelC executions of p are non-terminating, and thus memory safe. If (2) is true, then since all KernelC executions of p must not only have prefixes of length n, but also those prefixed should be valuations of the SafeKernelC execution, and since the valuation of a final SafeKernelC configuration would yield a final KernelC configuration, it follows that all KernelC executions are precisely of length n and end up in final comfgurations, thus p is memory safe.
2. Given program p, one can verify its memory safety by simply "executing" it using the SafeKernelC definition. As long as the symbolic execution in SafeKernelC can proceed, strong memory safety is guaranteed to hold up to that point. That means that the execution of the program on *any* real machine using KernelC is guaranteed to be memory safe up to the same point.

Ultimately, for terminating programs, monitoring strong memory safety becomes a sound decision procedure for verifying memory safety.

Corollary 1. *Strong termination and strong memory safety remain undecidable in general, but strong memory safety of terminating programs is decidable, assuming the oracle used for side conditions is a decision procedure.*

Proof. First two claims follow from the fact that SafeKernelC is Turring complete. If a program is known to terminate, then it also strongly terminates (as a coroallary of Proposition 3), thus its canonical derivation is finite, and by constructing it (which we can, given our oracle is a decision procedure), we can effectively check whether the last configuration obtained in this derivation is indeed final.

5 Conclusions

We have presented the first (up to our knowledge) formal definition of memory safety for a language allowing direct allocation and addressing of memory. After showing that verification of memory safety is not amenable for automation in general, through a suite of undecidability results, we proposed strong memory safety, a meaningful restriction of memory safety, and proved that it is runtime

verifiable. Our main result is that runtime verification of strong memory safety is a sound decision procedure for memory safety. The preliminary executable definition is available for download (and experimentation) as a part of the K-Maude distribution [16].

References

1. Necula, G.C., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. In: POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2002) 128–139
2. Hastings, R., Joyce, B.: Purify: Fast detection of memory leaks and access errors. In: Proceedings of the Winter USENIX Conference. (January 1992) 125–136
3. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In Ferrante, J., McKinley, K.S., eds.: PLDI, ACM (2007) 89–100
4. Berger, E.D., Zorn, B.G.: Diehard: probabilistic memory safety for unsafe languages. In: PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2006) 158–168
5. Novark, G., Berger, E.D., Zorn, B.G.: Exterminator: Automatically correcting memory errors with high probability. Commun. ACM **51**(12) (2008) 87–95
6. Harbison, S.P., Steele, G.L.: C: A Reference Manual (5th Edition). Prentice Hall (2002)
7. Roşu, G.: K: A Rewriting-Based Framework for Computations – Preliminary version. Technical Report UIUCDCS-R-2007-2926, University of Illinois (2007)
8. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theor. Computer Science **373**(3) (2007) 213–237
9. Şerbănuţă, T.F., Roşu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Inf. and Comp. (2009) to appear; `http://dx.doi.org/10.1016/j.ic.2008.03.026`.
10. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theor. Comput. Sci. **96**(1) (1992) 73–155
11. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1) (1994) 38–94
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., eds.: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. In Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., eds.: All About Maude. Volume 4350 of Lecture Notes in Computer Science., Springer (2007)
13. Walicki, M., Meldal, S.: Algebraic approaches to nondeterminism: An overview. ACM Comput. Surv. **29**(1) (1996) 30–81
14. Rosu, G., Schulte, W.: Matching logic. Technical Report UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign (2009)
15. Rogers Jr., H.: Theory of Recursive Functions and Effective Computability. MIT press, Cambridge, MA (1987)
16. Serbanuta, T.F.: K-Maude web page: http://fsl.cs.uiuc.edu/index.php/K-Maude.

A KernelC Syntax and Configuration in K-Maude

```
(k syntax for KERNELC is
  including GENERIC-EXP-SYNTAX + STRING-SYNTAX .
  sorts Stmt StmtList Pgm .
  subsort Stmt < StmtList .
  op #include<stdio.h>‘#include<stdlib.h>‘void‘main‘(void‘)‘{_‘}
       : StmtList -> Pgm [renameTo _] .
  op *_ : Exp -> Exp [strict prec 25] .
  op !_ : Exp -> Exp [aux] .
  vars E E’ : Exp .
  eq ! E = E ? 0 : 1 .
  ops _&&_ _||_ : Exp Exp -> Exp [aux] .
  eq E && E’ = E ? E’ : 0 .
  eq E || E’ = E ? 1 : E’ .
  op _?_:_ : Exp Exp Exp -> Exp [renameTo if‘(_‘)_else_ prec 39] .
  op _:=_ : Exp Exp -> Exp [strict(2) prec 40 gather (e E)] .
  op _; : Exp -> Stmt [prec 45 strict] .
  op ; : -> Stmt [renameTo .K] .
  op __ : StmtList StmtList -> StmtList
      [prec 100 gather(e E) renameTo _->_] .
  op ‘{_‘} : StmtList -> Stmt [renameTo _] .
  op ‘{‘} : -> Stmt [renameTo .K] .
  op malloc‘(_‘) : Exp -> Exp [strict] .
  op free‘(_‘) : Exp -> Exp [strict] .
  op if‘(_‘)_ : Exp Stmt -> Stmt [aux prec 47] .
  var St St’ : Stmt .
  eq if(E) St = if (E) St else {} .
  op if‘(_‘)_else_ : Exp Stmt Stmt -> Stmt [strict (1) prec 46] .
  op while‘(_‘)_ : Exp Stmt -> Stmt .
  op printf‘("%d "‘,_‘) : Exp -> Exp [strict] .
  op null : -> Exp [aux] .
  eq null = 0 .
k)

(k configuration for KERNELC is
  including KMAP{K, K} + FRESH-ITEM{K} .
  ops env mem ptr : -> CellLabel [wrapping Map‘{K‘,K‘}] .
  op out : -> CellLabel [wrapping K] .
  op stream : String -> K .
  op void : -> KResult .
k)
```

B KernelC Semantics in K-Maude

```
(k semantics for KERNELC is including GENERIC-EXP-SEMANTICS .
  var P : Pgm . var N N' : Nat . var X : Name .
  var Env : Map{K,K} . var V V' : KResult .  var I : Int .
  var Ptr Mem : Map{K,K} . var K K1 K2 : K . var S : String .
  kcxt * K1 := K2 [strict(K1)] . --- evaluating lhs to a lVal
  eq <T> P </T> = <T> <k> mkK(P) </k> <env> .empty </env>
                      <mem> .empty </mem> <ptr> .empty </ptr>
                      <nextItem> item(1) </nextItem>
                  <out> stream("") </out> </T> .
  eq #(true) = #(1) .  eq #(false) = #(0) .
  ceq if (#(I)) K1 else K2 = K2 if I eq 0 .
  ceq if (#(I)) K1 else K2 = K1 if I neq 0 .
  eq V ; = .K . --- discarding value of an expression statement
  keq <k> [[X ==> V]] ...</k> <env>... X |-> V ...</env>  .
  keq <k> [[X := V ==> V]] ...</k> <env> [[Env ==> Env[X <- V]]] </env> .
  keq <k> [[* #(N) ==> V]] ...</k> <mem>... #(N) |-> V ...</mem> .
  keq <k> [[* #(N) := V ==> V]] ...</k>
      <mem>... #(N) |-> [[V' ==> V]] ...</mem> .
  keq <k> [[while(K1) K2
        ==> if(K1) (K2 -> while(K1) K2) else .K]] ...</k> .
  op alloc : Nat Nat -> Map{K,K} .
  eq alloc(N, 0) = .empty .
  eq alloc(N, s(N')) = (#(N) |-> #(N + 1)) &' alloc(N + 1, N') .
  keq <k> [[ malloc(#(N)) ==> #(var(N'))]] ...</k>
     <ptr>... [[.empty ==> (#(var(N')) |-> #(N))]] ...</ptr>
     <nextItem> [[item(N') ==> item(N') + s(N)]] </nextItem>
     <mem>... [[.empty ==> alloc(var(N'), N)]] ...</mem> .
  op freeMem : Map{K,K} Nat Nat -> Map{K,K} .
  eq freeMem(Mem, N, 0) = Mem .
  eq freeMem((Mem &' (#(N) |-> V)), N, s(N')) = freeMem(Mem,N + 1,N') .
  keq <k> [[free(#(N)) ==> void]] ...</k>
     <ptr>... [[#(N) |-> #(N') ==> .empty]] ...</ptr>
     <mem> [[Mem ==> freeMem(Mem, N, N')]] </mem> .
  keq <k> [[printf("%d ",#(I)) ==> void]] ...</k>
     <out> [[stream(S) ==> stream(S + string(I,10)+ " ")]] </out> .
k)
```

Where the rules for _eq_/_neq_ are defined as follows:

```
  ops _eq_ _neq_ : Int Int -> Bool .
  eq X neq X = false .
  ceq X neq Y = true if X > Y .
  ceq X neq Y = true if X < Y .
  eq X eq Y = not(X neq Y) .
```

C SafeKernelC Semantics in K-Maude

Since it relies on the same syntax and configuration, it is called a semantics for KernelC as the previous definition. All rules stay unchanged, with the exception of the semantics rule for `malloc`(and its helping function `alloc`) which are modified to generate symbolic naturals both for pointers and the uninitialized memory locations:

```
eq alloc(var(NV), s(N')) = (#(var(NV)) |-> #(var(NV + 1)))
                         &' alloc(var(NV) +  1, N') .
eq alloc(var(NV) + N, s(N')) = (#(var(NV) + N) |-> #(var(NV + N + 1)))
                             &' alloc(var(NV) + N + 1, N') .
keq <k> [[ malloc(#(N)) ==> #(var(N'))]] ...</k>
   <ptr>... [[.empty ==> (#(var(N')) |-> #(N))]] ...</ptr>
   <nextItem> [[item(N') ==> item(N') + s(N)]] </nextItem>
   <mem>... [[.empty ==> alloc(var(N'), N)]] ...</mem> .
```

To support symbolic naturals and to allow the execution to advance, an additional "oracle" for symbolic naturals must now be provided:

```
sort NatVar .  --- the type for symbolic naturals
subsort NatVar < NzNat .  --- assume all symbolic naturals are non-zero
op var : Nat -> NatVar .  --- symbolic naturals constructor
vars X Y Z T : Int .
vars Nx : NzInt .
var Nn : NzNat .
eq 1 * X = X .
eq 0 * X = 0 .
eq 0 + X = X .
eq X - Y = X + (- 1) * Y .
eq X + (-1) * X = 0 .
eq X * (Y + Z) = X * Y + X * Z .
eq (X + Y) * Z = X * Z + Y * Z .
eq Nn > 0 = true .
eq Nn >= 0 = true .
eq (-1) * Nn > 0 = false .
eq (-1) * Nn >= 0 = false .
ceq X + Y <= 0 = true if X > 0 = false /\ Y > 0 = false .
eq X >= Nx = X - Nx >= 0 .
eq Nx <= Y = Y - Nx >= 0 .
eq X > Nx = X - Nx > 0 .
eq Nx < Y = Y - Nx > 0 .
eq X >= (-1) * Y = X + Y >= 0 .
eq X > (-1) * Y = X + Y > 0 .
eq (-1) * Y <= X = X + Y >= 0 .
eq (-1) * Y < X = X + Y > 0 .
```