Size-Change Termination as a Contract

Dynamically and Statically Enforcing Termination for Higher-Order Programs

PHÚC C. NGUYỄN, University of Maryland, USA
THOMAS GILRAY, University of Maryland, USA
SAM TOBIN-HOCHSTADT, Indiana University, USA
DAVID VAN HORN, University of Maryland, USA

Program termination is an undecidable—yet important—property relevant to program verification, optimization, debugging, partial evaluation, and dependently-typed programming, among many other topics. This has given rise to a large body of work on static methods for conservatively predicting or enforcing termination. A simple effective approach is the *size-change termination* method of Lee, Jones, and Ben-Amram, which operates in two-phases: (1) abstract programs into "size-change graphs," and (2) check these graphs for the *size-change property*: the existence of paths that lead to infinitely decreasing value sequences.

This paper explores the program termination problem, inspired by the size-change approach, but starting from a different vantage point: we propose transposing the two phases of the size-change termination analysis by developing an operational semantics that accounts for the *run-time checking of the size-change property*, postponing program abstraction or avoiding it entirely. This choice has two important consequences: size-change termination can be monitored and enforced at run-time and termination analysis can be rephrased as a traditional safety property and computed using existing abstract interpretation methods.

We formulate this run-time size-change check as a *contract* in the style of Findler and Felleisen. To the best of our knowledge, this contributes the first run-time mechanism for checking termination in a general-purpose programming language. The result nicely compliments existing contracts that enforce partial correctness specifications to obtain the first *contracts for total correctness*. Our approach combines the robustness of the size-change principle for program termination with the precise information available at run-time. It has tunable overhead and can check for nontermination without suffering from the conservativeness necessary in static checking. To obtain a sound and computable termination analysis, it is possible to apply existing abstract interpretation techniques directly to the operational semantics; there is no need for an abstraction tailored to size-change graphs. We apply the higher-order symbolic execution method of Nguyễn, *et al.* to obtain a novel termination analysis that is competetive with existing, purpose-built termination analyzers.

Additional Key Words and Phrases: termination, size-change principle, contracts, verification

1 SIZE-CHANGE CONTRACTS

Whether a program eventually terminates is one of the most useful, yet fundamentally and famously unknowable, properties [Davis 1958; Turing 1937]. This has given rise to a large body of work on termination analysis, which aims to predict termination for a useful, but necessarily incomplete, set of programs. Much of this work takes the form of sound static termination analysis, which conservatively predict, before a program is run, whether the program will terminate on all possible inputs (e.g. Cook et al. [2006a]; Jones and Bohr [2004]; Manolios and Vroon [2006];

Authors' addresses: Phúc C. Nguyễn, University of Maryland, College Park, MD, USA; Thomas Gilray, University of Maryland, College Park, MD, USA; Sam Tobin-Hochstadt, Indiana University, Bloomington, IN, USA; David Van Horn, University of Maryland, College Park, MD, USA.

2475-1421/2018/1-ART1
https://doi.org/

Walther [1994]). A prominent example of automated termination analysis is the *size-change termination* (SCT) approach of Lee et al. [2001], which operates in two phases: the first analyzes a program to generate an abstraction consisting of a set of well-founded data size-change assertions in the form of a "size-change graph;" the second analyzes the size-change graph to determine if it has the *size-change property*, namely:

> If every infinite computation would give rise to an infinitely decreasing value sequence (according to the size-change graphs), then no infinite computation is possible.

Importantly, this size-change property of graphs is decidable. When combined with a computable abstract interpretation of programs into size-change graphs this results in an effective, automated technique for termination analysis.

Size-change termination has several favorable characteristics: it is simple, effective, and can be applied to programs in any language, so long as they can abstracted to size-change graphs. However, the abstract-then-check is the source of both the strengths and weaknesses: SCT cannot, since it depends on abstraction to size-change graphs, form the basis of a dynamic termination check and this abstraction process is itself nontrivial to design. The latter often constitutes a research contribution in its own right, for example, the original SCT paper established an abstract interpretation to size-change graphs for first-order, purely functional programs; Jones and Bohr [2004] then showed how to extend the analysis to higher-order programs in the untyped λ-calculus; and later Sereni and Jones [2005] extended the analysis further to handle ML-style user-defined datatypes and general recursion. This highlights a problem: desigining the abstraction from programs to the size-change graph domain is difficult and there are a myriad of design possibilities. Moreover, there is a kind of *double abstraction* problem that occurs. If a program fails to provably terminate using the SCT method, it means one of three things: (1) the program does not terminate, (2) the abstraction to size-change graphs was too coarse, or (3) the program violates the size-change principle, but nonetheless terminates. So when verification fails, it's unclear whether it's due to the abstraction to size-change graphs or the abstraction inherent to the size-change principle, which is a sufficient, but not necessary, condition for termination.

1.1 Checking (a Sufficient Condition for) Termination at Run-Time

In this paper, we demonstrate a different path to termination checking based on the size-change principle. Rather than "abstract-then-check," we propose the "check-then-(maybe)-abstract" approach: we give a semantic account of size-change *checking* integrated into the operational semantics of a higher-order functional language. This checking provides a run-time enforcement mechanism for size-change termination. We show that this run-time mechanism is useful in practice and theory. Pragmatically, it provides a flexible run-time mechanism for detecting violations of the size-change principle, and therefore potential non-termination—with no abstraction required. Theoretically, it provides a "ground truth" semantics which can form the basis of termination analyses.

Some of the benefits of this approach are:

(1) Static analysis, especially that for higher-order programs, inherently suffers from approximate control-flow graphs, which results in spurious paths, preventing termination proofs of some programs. Dynamic analysis, on the other hand, only observes the actual control-flow, and solves an easier problem of checking for termination of a particular run.
(2) That dynamic checking can be more precise and specialized introduces new use-cases not practical for static checking. For example, programmers can optionally enforce termination on programs that do not terminate in general (e.g. interpreters on particular programs,

event-handlers, plugin systems, etc.). Many of these uses require enforcing termination on *unknown* code, which is also potentially useful in gradual type systems where the typed component needs termination but cannot check that for the untyped component.

(3) Finally, by being formulated as an operationally checkable property, termination checking can be systematically approximated into a static analysis with tunable precision using existing abstract interpretation techniques that do not need to be purpose-built for the size-change domain.

1.2 Termination as a Contract for Total Correctness

We formulate the size-change termination check as a behavioral software contract in the style of Findler and Felleisen [2001]. Behavioral software contracts have found wide success and adoption because runtime monitoring allows specifications to be expressed as programs and avoids the problem of false positives inherent to static checking. However, run-time monitoring is necessarily limited to safety properties—it is impossible to determine precisely that a computation has entered an infinite loop, for example. This limits existing contracts to partial rather than total correctness. By incorporating a *size-change termination contract*, we lift this restriction, providing a terminating function contract combinator that enforces total correctness.

1.3 Termination Analysis for "Free"

Having an operational account of the size-change termination check provides a semantic basis for designing and deriving termination static analyses following familiar techniques. What is essential is that SCT approximates a classic example of a liveness property (termination), with a safety property (size-change). Checking this property and "going wrong" when it is violated renders termination (or rather an approximation to it) something analogous to an array out-of-bounds error or a run-time type mismatch. By using familiar techniques such as type systems or abstract interpretation it is possible to statically eliminate run-time SCT errors, thereby verifying termination before a program is run.

We design a termination analysis following the outline of Nguyễn et al. [2014, 2017], which is a general purpose technique for abstracting an operational semantics using a combination of SMT-backed symbolic execution and sound overapproximation of run-time behavior; it was not designed with termination analysis in mind. When applied to the SCT checking semantics it results in an termination analysis that is competitive with a number of existing systems tailor-made for termination analysis.

Contributions

This paper makes the following contributions:

(1) We formulate size-change termination as a correctness property that can be dynamically checked at run-time.
(2) We present a technique for run-time monitoring of size-change termination that preserves tail-calls and that exhibits tuneable run-time overhead.
(3) We prove that our run-time monitoring of size-change termination is sound and SCT-complete up to a (parametric) well-founded partial order.
(4) We evaluate our implementation of dynamic size-change termination checking on a set of benchmarks and report precision and slow-down for terminating programs, as well as time to detect diverging programs.

(5) We evaluate the static analysis that results from adding support for our termination contract to an existing contract verifier (used as both a program verifier capable of verifying termination in complex control-flow, as well as a theorem prover). We compare the results against several state-of-the-art static termination verifiers.

2 EXAMPLES

This section develops intuitions for how dynamic checking of *size-change termination* (SCT) works via worked examples. When the `terminating/c` contract wraps a function `f`, it ensures that whenever `f` is applied to any argument list, all computations within the dynamic extent of `f`'s application to those arguments eventually terminate, either because `f` can only adhere to the size-change principle a finite number of times, or because `f` violates SCT and is blamed. The size-change principle is enforced each time a function in the dynamic extent of `f` recursively calls itself, either directly or indirectly (*e.g.*, through mutual recursion or higher-order recursion). We monitor SCT by accumulating a size-change graph for each function along the call-stack. The monitoring system allows a recursive call to proceed only if it can identify a sequence of strictly descending argument lists for that function, according to some well-founded partial order.

2.1 Ackermann

The function `ack` defined in this example behaves like the standard Ackermann function, but with `terminating/c` enforcing that any computation following the application of `ack` terminates.

The contract `terminating/c` around `ack` ensures that within the dynamic extent of any callsite `(ack m n)`, all recursive calls to `ack` only apply `ack` to an argument list that is strictly "smaller". To dynamically detect the lexicographical order without any annotational help, we construct the size-change graph, as presented in the original SCT paper [Lee et al. 2001], on the fly.

```
(define ack
  (terminating/c
    (λ (m n)
      (cond
        [(= 0 m) (+ 1 n)]
        [(= 0 n) (ack (- m 1) 1)]
        [else    (ack (- m 1) (ack m (- n 1)))]))))
```

At runtime, there are three ways that `ack` can recursively call itself, extending its size-change graph as each call occurs. First, `(ack m n)` can call `(ack (- m 1) 1)`, which strictly descends on `m` and maintains no order on `n` (case 1). Second, `(ack m n)` can call `(ack m (- n 1))` as part of `ack`'s third clause, which maintains order on `m` and strictly decreases on `n` (case 2). Finally, `(ack m n)` can call `(ack (- m 1) ...)`, which strictly descends on `m` and maintain no order on `n` (case 3). In each of these three cases, we extend a size-change graph accordingly (shown in Figure 1),

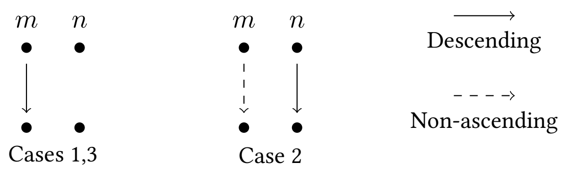

Fig. 1. Size-change graphs for the three recursive calls in ack.

with no extension breaking the size-change graph's requirements, that: (i) each extension to the graph is observed to make strict progress on some argument, and (ii) the extended graph is also guaranteed to indicate strict progress since the start of the monitoring up to the current point.

It is important that both conditions (i) and (ii) are met. Condition (i) is clearly necessary for progress at each step. If only (i) was checked however, an incorrect implementation of `ack` could have subverted monitoring by increasing `m` while decreasing `n` for one call, and doing the reverse for another, in a way that yields nontermination. Below Figure 2 shows the different ways the size-change graph of `ack` can be incrementally extended (*i.e.*, concatenated and compressed). That the graph is maintained in all observed cases means that `ack` will eventually exhaust argument `m` and terminate.

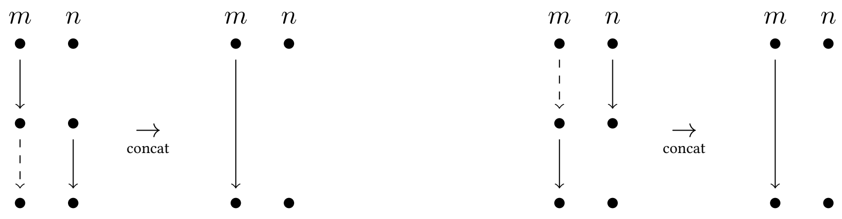

Fig. 2. Concatenating size-change graphs.

2.2 Length in CPS

Consider a `len` function for lists, written in continuation-passing style (below). Static analysis of size-change termination relies on an underlying control-flow graph, which must eventually conflate all closures generated by `(λ (n) (k (+ 1 n)))`, regardless of call-sensitivity. This results in a spurious loop where each closure bound to `k` may appear to call one with a *larger* argument, failing the size-change principle.

```
(define len
  (terminating/c
    (λ (l)
     (let loop ([l l] [k (λ (x) x)])
      (match l
        [(cons h t) (loop t (λ (n) (k (+ 1 n))))]
        [_ (k 0)])))))
```

Dynamic checking of size-change termination does not have this problem, because all the closures are exact and distinct. Even though the number of closures, are arbitrary, they are finite up to the previous loop descending on `l`, which has been proven to terminate.

2.3 Lambda-calculus interpreter

Checking termination of an interpreter for a turing-complete language is challenging—after all, the interpreter does not terminate on all programs. Nevertheless, dynamic size-change monitoring allows the interpretation of many interesting programs to finish. In this example, we present a λ-calculus interpreter that first compiles the term to a closure and then applies this closure to an environment. The compilation itself terminates by structural recursion, which is simple to check, but the compilation result is not. In fact, in this example, the first residual procedure c₁ terminates when run, but the second one c₂ loops infinitely. Dynamic size-change monitoring flexibly allows the first one to finish, and quickly catches the divergence in the second one.

```
(define comp
  (terminating/c
    (λ (e)
     (match e
       [`(λ (,x) ,e) (let ([c (comp e)])
                       (λ (ρ) (λ (z) (c (hash-set ρ x z)))))]
       [`(,e₁ ,e₂) (let ([c₁ (comp e₁)]
                         [c₂ (comp e₂)])
                     (λ (ρ) ((c₁ ρ) (c₂ ρ))))]
       [(? symbol? x) (λ (ρ) (hash-ref ρ x))]))))
(define c₁ (terminating/c (comp '((λ (x) (x x)) (λ (y) y)))))     ; Okay
(define c₂ (terminating/c (comp '((λ (x) (x x)) (λ (y) (y y)))))) ; Okay
(c₁ (hash)) ; Okay
(c₂ (hash)) ; Error
```

Procedure c₁ terminates because it never calls itself with a non-decreasing argument. In contrast, c₂ calls itself with the same environment when executed, hence caught by the size-change monitoring. As demonstrated in the evaluation section (§5), the monitoring system is able to check for non-termination of a Scheme interpreter running merge-sort on a particular list.

3 DYNAMIC MONITORING FOR SIZE-CHANGE TERMINATION

3.1 Syntax and Semantics

This section introduces λ_{SCT}, which extends λ-calculus with base values, primitive operations, and special form (term/c e) that guards (e) with a contract, ensuring it behaves as a size-change-terminating function. Figure 3 shows our core language.

We present λ_{SCT}'s semantics through reduction over machine states (ς). A state is either an intermediate state, which consists of a value (v) on top of the stack (κ), or an error indicating that an asserted size-change contract has been violated. For simplicity, we assume programs in λ_{SCT} are well-typed, and the only possible error is a SCT violation.

Values (v) in λ_{SCT} includes primitives (o), integers (n), closures (Clo$(\vec{x}, e, \rho)$), and termination-checked closure (term/c(Clo$(\vec{x}, e, \rho)$)). We assume no primitive in λ_{SCT} causes divergence.

Stacks (κ) represent the rest of the computation, paired with a size-change table (m) remembering function call history. The table maps each function (v) to the most recent arguments it was applied to in the current dynamic extent, as well as a size-change graph (g) recording ways in which arguments of (v) descend. A size-change graph is a set of edges of the form ($i \downarrow j$) or ($i \overline{\downarrow} j$), indicating that argument at index i always strictly descends ($\downarrow$) or never ascends ($\overline{\downarrow}$) to parameter j.

Figure 4 shows the loading of expression (e) into an initial state. The helper function ($distr$) "pushes" outer parts of the expression onto the stack.

3.1.1 Reduction rules. Figure 5 highlights where λ_{SCT}'s semantics differ from that of a standard functional intermediate language.

Rule *[Wrap]* shows the introduction of a termination-checked function. When a value v is returned to a continuation expecting a function to wrap with the `term/c` monitor, the value v is

$$
\begin{array}{rl}
\text{[Expressions]} & e ::= o \mid b \mid (\lambda\ (\vec{x})\ e) \mid (e\ \vec{e}) \mid (\mathsf{term/c}\ e) \\
\text{[Value Literals]} & b ::= 0 \mid -1 \mid 1 \mid \ldots \\
\text{[Primitives]} & o ::= + \mid \mathsf{cons} \mid \mathsf{car} \mid \mathsf{cdr} \mid \ldots \\
\text{[States]} & \varsigma ::= (v, \kappa) \mid \mathsf{error} \\
\text{[Values]} & v ::= o \mid b \mid (\mathsf{cons}\ v\ v) \mid \mathsf{Clo}(\vec{x}, e, \rho) \mid \mathsf{term/c}(\mathsf{Clo}(\vec{x}, e, \rho)) \\
\text{[Stacks]} & \kappa ::= \mathsf{halt} \mid (\vec{v} \,\square\, \vec{e}\ \rho\ \kappa\ m) \mid (\mathsf{term/c} \,\square\, \kappa\ m) \\
\text{[Size-change Table]} & m \in v \rightharpoonup \vec{v} \times g \\
\text{[Size-change Graph]} & g \in \mathcal{P}(\mathbb{N} \times r \times \mathbb{N}) \\
\text{[Change]} & r ::= \downarrow \mid \overline{\downarrow}
\end{array}
$$

Fig. 3. Syntax of λ_{SCT}.

$$
\begin{array}{rcl}
inj & : & e \to (v, \kappa) \\
inj(e) & = & distr(e, \{\}, \mathsf{halt}, \{\}) \\
\\
distr & : & e \times \rho \times \kappa \times (m \cup \{\bot\}) \to \varsigma \\
distr((\lambda\ (\vec{x})\ e), \rho, \kappa, m) & = & (\mathsf{Clo}(\vec{x}, e, \rho), \kappa) \\
distr(o, \_, \kappa, m) & = & (o, \kappa) \\
distr(n, \_, \kappa, m) & = & (n, \kappa) \\
distr(x, \rho, \kappa, m) & = & (\rho(x), \kappa) \\
distr((e_1\ \vec{e}), \rho, \kappa, m) & = & distr(e_1, \rho, ([] \,\square\, \vec{e}\ \rho\ \kappa\ m), m) \\
distr((\mathsf{term/c}\ e), \rho, \kappa, m) & = & distr(e, \rho, (\mathsf{term/c} \,\square\, \kappa\ m), m) \\
distr(\_, \_, \_, \bot) & = & \mathsf{error}
\end{array}
$$

Fig. 4. Program injection and loading components into states.

embeded in a guarded closure $\mathsf{term/c}(v)$ and returned to the continuation κ. Only closures are capable of violating SCT in λ_{SCT}, so we only wrap closures and return other values as-is.

Rule *[AppClo]* shows application of a closure. If the size-change table is empty, SCT is not currently being enforced in this evaluation context, and the application steps to the function's body as usual. If the table is non-empty, we update it with this new application before proceeding. Helper function update updates the size-change table with the function's latest arguments and size-change graph, potentially returning $\bot$ is there is a size-change violation.

Rule *[AppTerm]* shows the application of a monitored closure, which enables SCT enforcement if it is not enabled, and proceeds otherwise as application of a plain closure.

3.2 Updating and monitoring size-change graphs

Figure 6 lists helper functions that update and monitor SCT.

Function *update* takes the size-change table (m), function (v), and its latest arguments ($\overrightarrow{v_{\mathrm{cur}}}$). It computes a new size-change graph (g_1) for the transitions from the previous arguments ($\overrightarrow{v_{\mathrm{prev}}}$) to these new arguments, ensures that the new graph (g_1) makes size-change progress on its own as well as when composed with the old graph g_0, and then updates the graph in m. If function v has

$$\longmapsto \quad : \quad \varsigma \to \varsigma$$

$$\begin{array}{rll}
(v, (\mathsf{term/c} \,\square\, \kappa\, m)) & \longmapsto \; (v', \kappa) & \textit{[Wrap]} \\
\text{where} & v' = \begin{cases} \mathsf{term/c}(v), \text{ if } v = \mathsf{Clo}(\_,\_,\_) \\ v, \text{ otherwise} \end{cases} &
\end{array}$$

$$\begin{array}{rll}
(v_n, (\vec{v} \,\square\, ()\ \_\ \kappa\, m)) & \longmapsto \; \mathit{distr}(e, \rho[\overrightarrow{x \mapsto v_x}], \kappa, m') & \textit{[AppClo]} \\
\text{where} & \mathsf{Clo}(\vec{x}, e, \rho)\ \overrightarrow{v_{\text{args}}} \ \equiv\ \vec{v}\ v_n & \\
\text{and} & m' = \begin{cases} \{\}, \text{ if } m = \{\} \\ \mathit{update}(m, \mathsf{Clo}(\vec{x}, e, \rho), \overrightarrow{v_{\text{args}}}), \text{ otherwise} \end{cases} &
\end{array}$$

$$\begin{array}{rll}
(v_n, (\vec{v} \,\square\, ()\ \_\ \kappa\, m)) & \longmapsto \; \mathit{distr}(e, \rho[\overrightarrow{x \mapsto v_x}], \kappa, m') & \textit{[AppTerm]} \\
\text{where} & \mathsf{term/c}(\mathsf{Clo}(\vec{x}, e, \rho))\ \overrightarrow{v_{\text{args}}} \ \equiv\ \vec{v}\ v_n & \\
\text{and} & m' = \mathit{update}(m, \mathsf{Clo}(\vec{x}, e, \rho), \overrightarrow{v_{\text{args}}}) &
\end{array}$$

Fig. 5. Non-standard reduction rules of λ_{SCT}.

not been applied before and there is no entry in m, a trivial graph is inserted, where each argument is assumed to have descended from an arbitrary one (e.g. itself).

Function *graph* computes a size-change graph from two value lists. For each value v_j at index j in the latter list that is known to be strictly smaller than some value v_i at index i in the former list, an edge $(i \downarrow j)$ is included in the graph. When the values are equal, we include $(i \overline{\downarrow} j)$ instead.

The composition ($\circ$) of two size-change graphs (g_0 and g_1) includes an edge $(i \downarrow k)$ if there is an edge $(i\ r\ j)$ in g_0 and $(j\ r\ k)$ in g_1, with at least one edge being a strict descent. If i propagates to k only through non-ascendence, the weaker edge $(i \overline{\downarrow} k)$ is included.

Finally, predicate *progress?* checks if there is any strict descendence in a size-change graph.

3.3 Well-founded partial order

Figure 7 shows an example of a well-founded patial order ($\preceq$) on values in λ_{SCT}. It is defined on integers by comparing absolute values, and (transitively) a field of a data-structure is considered smaller than the data-structure that contains it. Although simple, this relation is sufficient to check for termination in most programs that descend on integers and data-structures. If a program descends following a different order, the user of λ_{SCT} can replace the default order with an appropriate one.

3.4 Soundness and Completeness

The size-change property is a safe over-approximation to ensure termination. The correctness of monitoring this property can therefore be understood as any strategy that satisfies the following properties:

- soundness: monitoring any program that diverges will result in a contract error;
- completeness: any contract error is the result of a program violating the size-change principle.

We now formally establish the correctness of the size-change monitoring semantics.

$$
\begin{array}{rcl}
\textit{update} & : & m \times v \times \vec{v} \to m \cup \{\bot\} \\
\textit{update}(m, v, \overrightarrow{v_{\text{cur}}}) & = & m[v \mapsto (\overrightarrow{v_{\text{cur}}}, \{(i \downarrow i) \mid \ldots v_i \ldots \equiv \overrightarrow{v_{\text{cur}}})], \text{if } v \notin m \\
\textit{update}(m, v, \overrightarrow{v_{\text{cur}}}) & = & \begin{cases} \bot, \text{if } \neg\textit{progress?}(g_1) \text{ or } \neg\textit{progress?}(g_0 \circ g_1) \\ m[v \mapsto (\overrightarrow{v_{\text{cur}}}, g_0 \circ g_1)], \text{otherwise} \end{cases} \\
\text{where} & & (\overrightarrow{v_{\text{prev}}}, g_0) \equiv m(v) \\
\text{and} & & g_1 = \textit{graph}(\overrightarrow{v_{\text{prev}}}, \overrightarrow{v_{\text{cur}}}) \\
\\
\textit{graph} & : & \vec{v} \times \vec{v} \to g \\
\textit{graph}(\overrightarrow{v_{\text{prev}}}, \overrightarrow{v_{\text{cur}}}) & = & \{(i \downarrow j) \mid \ldots v_i \ldots \equiv \overrightarrow{v_{\text{prev}}}, \ldots v_j \ldots \equiv \overrightarrow{v_{\text{cur}}}, v_j \prec v_i\} \\
& \cup & \{(i \downarrow\!\!\!\!- j) \mid \ldots v_i \ldots \equiv \overrightarrow{v_{\text{prev}}}, \ldots v_j \ldots \equiv \overrightarrow{v_{\text{cur}}}, v_j = v_i\} \\
\\
(\circ) & : & g \times g \to g \\
g_0 \circ g_1 & = & \{(i \downarrow k) \mid \quad ((i \downarrow j) \in g_0 \text{ and } (j\ r\ k) \in g_1) \text{ or} \\
& & \qquad\qquad\quad ((i\ r\ j) \in g_0 \text{ and } (j \downarrow k) \in g_1)\} \\
& \cup & \{(i \downarrow\!\!\!\!- k) \mid \quad (i \downarrow\!\!\!\!- j) \in g_0 \text{ and } (j \downarrow\!\!\!\!- k) \in g_1 \text{ and} \\
& & \text{for all } j, (i\ r_0\ j) \in g_0 \text{ and } (j\ r_1\ k) \in g_1 \text{ implies } (r_0 = r_1 = \downarrow\!\!\!\!-)\} \\
\\
\textit{progress?} & : & g \to \mathbb{B} \\
\textit{progress?}(g) & = & \exists\, i\ j, (i \downarrow j) \in g
\end{array}
$$

Fig. 6. Updating and monitoring size-change graphs

$$
\begin{array}{rcll}
\prec, \preceq & \subseteq & v \times v & \\
n_1 & \prec & n_2 & \text{if } |n_1| < |n_2| \\
v & \prec & (\text{cons } v'\ \_) & \text{if } v \preceq v' \\
v & \prec & (\text{cons } \_\ v') & \text{if } v \preceq v' \\
v & \preceq & v' & \text{if } v \prec v' \text{ or } v = v'
\end{array}
$$

Fig. 7. Example well-founded partial order $\preceq$

Theorem 3.1 (Soundness of size-change monitoring). *If* (ee_1) *diverges, then* $((\text{term/c } e)e_1)$ *terminates with* error.

Proof. Consider an infinite chain of function calls. By Lemma 3.2 below, there's a closure that keeps being called. The sequence of arguments to this closure cannot satisfy the size-change property an infinite number of times. The diverging program that results in this call chain will be killed. □

Lemma 3.2 (Recurring closure). *Along any infinite chain of function calls, there is at least one closure that is called infinitely often.*

Proof. Consider the sequence of closures $\text{Clo}(x_1, e_1, \rho_1), \ldots, \text{Clo}(x_i, e_i, \rho_i), \ldots$ along the infinite call chain.

- Case 1: The closures come from a finite set. At least one must repeat infinitely often.
- Case 2: There are fresh closures that keep being generated dynamically. Because new infinite closures must be generated through finite λ forms, there must be some infinite subset of

closures $\mathsf{Clo}(x_i, e_i, \_)$ generated by one same form `(λ (`x_i`)` e_i`)`. Let $\mathsf{Clo}(x_m, e_m, \_)$ be the set of closures whose body e_m contains this form `(λ (`x_i`)` e_i`)`.
- Claim: There must be some closure $\mathsf{Clo}(x_j, e_j, \rho_j)$ that keeps being called infinitely often.
- Proof: By induction on the lexical depth of `(λ (`x_m`)` e_m`)`.
  * Subcase 1: `(λ (`x_m`)` e_m`)` has lexical depth 0 (i.e. it is a top-level λ). Because it is not enclosed by any λ, the closure $\mathsf{Clo}(x_m, e_m, \{\})$ is created only once. By assumption, $\mathsf{Clo}(x_m, e_m, \{\})$ is called infinitely often to dynamically create the infinite closure set $\mathsf{Clo}(x_i, e_i, \_)$.
  * Subcase 2: `(λ (`x_m`)` e_m`)` is directly enclosed by `(λ (`x_n`)` e_n`)`.
    · Subsubcase 2a: The set $\mathsf{Clo}(x_m, e_m, \_)$ is finite: at least one of them is called infinitely often to generate the infinite closure set $\mathsf{Clo}(x_i, e_i, \_)$.
    · Subsubcase 2b: The set $\mathsf{Clo}(x_m, e_m, \_)$ is infinite: apply the induction hypothesis on `(λ (`x_n`)` e_n`)` (where new i is m and new m is n).

□

Theorem 3.3 (SCT-completeness). *If* `((term/c` e`)` e_1`)` *steps to* error, *and* stermc *does not appear in* e *or* e_1, *then* `(`e e_1`)` *steps to an application that violates the size-change property.*

Proof. By inspection of the last step to error by `((term/c` e`)` e_1`)`. □

4 STATIC VERIFICATION OF SIZE-CHANGE TERMINATION

Given termination formulated as a dynamically checkable property, we can systematically turn these dynamic checks into static verification by building on previous work in higher-order symbolic execution [Nguyễn et al. 2014; Tobin-Hochstadt and Van Horn 2012; Van Horn and Might 2011].

Symbolic execution extends the standard semantics with symbolic values that can stand for any values (including higher-order values), and maintain a path-condition, which is a formula about facts that must hold for symbolic values on each path. Because termination checks ultimately decompose into “less-than” checks, which check for a definite descent of values along a well-founded partial order, there is no special challenge in using symbolic execution for size-change termination checking. Symbolic execution can readily leverage SMT solvers for precise reasoning about path-conditions, proving termination that depends on sophisticated path-sensitivity.

Although symbolic execution has traditionally been used to find bugs [Cadar et al. 2008; King 1976; Majumdar and Sen 2007; Sen 2007; Sen et al. 2005] as opposed to verifying programs as correct, we can apply a well studied technique for abstracting the operational semantics through finitizing the program’s dynamic components [Van Horn and Might 2010] and obtain a verification that particular errors cannot actually occur at runtime.

4.1 Extended semantics

Figure 8 shows extension to λ_{SCT} that allows symbolic execution. We extend the set of values (v) with *symbolic values* (s), which can stand for any value. The modified semantics of λ_{SCT} must then account for symbolic values, which means some state can non-deterministically step to multiple states to soundly over-approximate all the cases the result from possible instantiations of symbolic values. Symbolic execution maintains a *path-condition* (ϕ) to characterize each path, which is a set of symbolic values assumed to have evaluated to true (interpreted as a conjunction).

With symbolic values, established orders between values are more conservative, and the size-change graphs computed between symbolic value lists as in Figure 6 have, in general, no more edges than in the concrete case as each edge now represents a must-descend or must-not-ascend

$$
\begin{array}{rl}
\text{[States]} & \varsigma ::= (v, \kappa, \phi) \mid \mathsf{error} \\
\text{[Values]} & v ::= \ldots \mid s \\
\text{[Symbolic Values]} & s ::= x \mid b \mid (o\, \vec{s}) \\
\text{[Path Conditions]} & \phi = \vec{s}
\end{array}
$$

Fig. 8. Syntax of symbolic λ_{SCT}.

relationship over all possible concrete paths. A sufficiently precise symbolic execution, coupled with effective SMT solving, can maintain a graph with enough edges to prove that functions will always maintain their size-change properties.

4.2 Ackermann revisited

Now consider again the example `ack`, a termination-checked Ackermann function shown in Figure 9 We subscript recursive calls with 1, 2, and 3.

```
(define ack
  (terminating/c
    (λ (m n)
      (cond
        [(= 0 m) (+ 1 n)]
        [(= 0 n) (ack (- m 1) 1)₁]
        [else    (ack (- m 1)
                      (ack m (- n 1))₂)₃]))))
```

Fig. 9. Ackermann

To verify `ack`, we apply the function on symbolic natural numbers `m` and `n` that have passed `ack`'s precondition with the path-condition `{(≥ m 0), (≥ n 0)}`. With these symbolic inputs, execution non-deterministically takes all three branches, and accumulates in the path-condition assumptions about the values: in the first branch, `m` is `0`; in the second branch, `m` is positive and `n` is `0`; in the last branch, both `m` and `n` are positive. To ease notation of states, we re-construct the stack (κ) around the value (v) as an expression and omit displaying components such as the size-change table (m). The initial state $((\mathsf{ack}\ m\ n), \emptyset)$, steps to three states:

$$
\begin{array}{rcl}
((\mathsf{ack}\, m\, n), \emptyset) & \rightsquigarrow & ((+\, 1\, n), \{(=\ m\, 0), (\geq\ n\, 0)\}) \\
& \rightsquigarrow & ((\mathsf{ack}\ (-\, m\ 1)\ 1), \{(>\ m\, 0), (=\ n\, 0)\}) \\
& \rightsquigarrow & ((\mathsf{ack}\ (-\, m\ 1)\ (\mathsf{ack}\ m\ (-\, n\ 1))), \{(>\ m\, 0), (>\ n\, 0)\})
\end{array}
$$

The first case simply returns and does not trigger any size-change monitoring.

The second reaches call site 1 with the path-condition `(= n 0)`. The recursive call proceeds, checking for all relationships that can be established between the old and new arguments as in Figure 6. In this case, with the path-condition that both `m` and `n` are non-negatives, symbolic execution easily proves that `(- m 1)` is less than `m` according to the partial order defined in Figure 7. No other definite order can be established between the new arguments `(- m 1)`, `1` and the old ones `m`, `n`. This gives the new size-change graph of $\{(0 \downarrow 0)\}$. We extend the initial size-change of `ack` with this new graph. (The initial size-change graph for any function assumes every index has been descending, as described in Section 3.)

The third path reaches call site 2. The path-condition, again, is sufficient for establish the descending order from `n` to `(- n 1)` and maintenance of `m`, yielding the new graph $\{(0 \downarrow\!\!\!\!\!\!\!\overline{\ \ } 0), (1 \downarrow 1)\}$. This graph, when composed with the initial graph, maintains the strict descending of `m`. When execution reaches the outer recursive call of `ack`, the descendence from `m` to `(- m 1)` be straightforwardly established. In all three paths, the added size-change graph makes strict progress on some parameter, either `m` or `n`, and the final concatenated graph also makes strict progress overall on `m`. Furthermore, it can be proved through the path-condition that each recursive call maintains the invariant that the arguments are natural numbers, which repeats the above reasoning for any new recursive call. Therefore, a size-change violation cannot happen in `ack`, and the function terminates for all natural numbers.

4.3 Soundness of static size-change termination verification

PROPOSITION 4.1 (SOUNDNESS OF STATIC VERIFICATION). *If* $(e\ e_1)$ *diverges, symbolic execution of* $(((\mathsf{term/c}\ e)\ s), \emptyset)$ *steps to* error*, where* s *is a fresh symbolic value.*

PROOF. Follows from soundness of dynamic checking of size-change termination (theorem 3.1) and soundness of higher-order symbolic execution [Nguyễn et al. 2018]. □

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

We implement the semantics presented in Section 3 as a library in the Racket programming language through instrumentation of the application form.

An application form `(f x ...)` in Racket is syntactic sugar for `(#%app f x ...)`, and libraries can modify what an application means by redefining the `#%app` form. For our purpose, we redefine the application form to implement the rules [AppClo] and [AppTerm] in Figure 5. If size-change termination is being enforced, the `#%app` form looks up the size-change table to guard against any violation. We implement the size-change table as a continuation-mark [Clements and Felleisen 2004], which preserves tail-calls. Finally, we expose a parameter specifying the custom partial order for use in termination checks, with a default implementation as described in Figure 7.

Although a naive implementation would be prohibitively expensive, with a few optimizations, the overhead can be brought down to acceptable for the goal of debugging

- Reducing monitoring frequency: The construction of size-change graphs is expensive, but need not be performed each time a function calls itself recursively. Because strict progress down any well-founded partial order can only be maintained a finite number of times, any non-SCT program will violate the size-change principle regardless of the monitoring frequency. We therefore only extend and monitor each function's size-change graph exponentially less frequent (e.g. at iterations 1, 2, 4, 8, etc.). This significantly reduces the monitoring overhead, although risks keeping data from earlier iterations live for longer than would be otherwise.
- Avoiding instrumentation for known terminating functions: In many cases, functions that are known to terminate need no instrumentation. For example, we assume all primitives terminate.
- Monitoring size-change graphs only for loop entries: We identify "loop entries" to monitor instead of constructing and monitoring a size-change graph for each function. For example, suppose `even?` and `odd?` are mutual recursive functions, where the top-level context calls `even?`, then only `even?` is a loop-entry and have a size-change graph constructored and monitored.

The next section evaluates overhead and effectiveness of size-change monitoring.

5.2 Evaluation

We evaluate the effectiveness and efficiency of size-change monitoring. Effectiveness monitoring should allow all or most terminating programs to finish execution, and quickly catch diverging programs. Efficient monitoring should introduce little overhead compared to execution without monitoring.

5.2.1 Effectiveness and efficiency on terminating programs. Table 1 shows terminating programs we use to evaluate the dynamic checks and static analysis of terminating contracts. The programs were collected from previous work on termination checking:

(1) `sct`: size-change termination for first-order programs [Lee et al. 2001],
(2) `ho-sct`: size-change termination for higher-order programs [Sereni and Jones 2005],
(3) `lh`: Liquid Haskell [Vazou et al. 2014b],
(4) `isabelle`: Isabelle [Krauss 2007],
(5) `acl2`: ACL2 [Manolios and Vroon 2006], and
(6) a collection of larger Scheme benchmarks that terminate by the size-change principle.

The table shows the precision of dynamic checking and static analysis, as well as comparison with other systems where possible. Most programs are small and under 15 lines. The largest program is `scheme` with 1,100 lines, which implements an interpreter for R5RS Scheme that interprets the mergesort algorithm on a list of strings. We did our best efforts to translate programs from one system to another. For example, `sct-2` is originally an untyped program composing a heterogenous list which cannot be typed in Liquid Haskell and Isabelle. We translated `sct-2` to work with an equivalent custom tree data-type.

Several cases where the programs need modifications to be successfully verified by the systems are annotated in the table. For example, `sct-1` and `sct-2` originally use conditionals, and can only be verified when converted to use pattern-matching. Some other programs are only verified successfully with annotational help on termination, such as explicit lexical ordering in the case of Liquid Haskell (e.g. `lh-merge`), or a custom partial-order in dynamic monitoring (e.g. `acl2-fig-2`). Some programs cannot be expressed in all systems. For example, ACL2 cannot check higher-order programs, and the type systems in Liquid Haskell and Isabelle do not support the Y-combinator that has self-applications (e.g. `ho-sct-ack`). To our surprise, the current versions of the tools cannot check some of their own benchmarks despite our best efforts to reproduce (e.g. `isabelle-poly` for Isabelle, and `acl2-fig-2` and `acl2-fig-6` for ACL2). Overall, our system works well for a wide range of programs and idioms, including higher-order untyped programs with moderate side effects (such as in the Scheme benchmarks).

Figure 10 shows the slowdown when enabling dynamic checks for select programs: `factorial`, `sum`, and `scheme`. These programs demonstrate different patterns in recursive programs that incur different levels of overhead from size-change monitoring. For programs that do heavy computation such as `factorial`, multiplying large integers, overhead is neglible. For programs that consist mainly of tight loop such as `sum` or repeatedly loop on large data-structures such as `scheme`, overhead is much more significant. That the overhead stays fixed (approximately two orders of magnitude) when the input grows suggests that further optimization effort to trim down the constant factor can make monitoring suitable for realistic uses.

5.2.2 Effectiveness on diverging programs. We also evaluate dynamic monitoring on diverging programs on how quickly the monitoring system catches divergence. These programs include modified versions of correct programs, as well as one originally incorrect program (e.g., `nfa`) that our static analysis discovered. Because violation of the size-change principle tend to show up in early

Program	Dynamic	Static	Liquid Haskell	Isabelle	ACL2
sct-1 (rev)	✓	✓	✓^{R}	✓	✓
sct-2	✓	✓	✗	✓^{R}	✓
sct-3 (ack)	✓	✓	✓^{A}	✓	✓
sct-4	✓	✓	✗	✓	✓
sct-5	✓	✓	✗	✓	✓
sct-6	✓	✓	✗	✓	✓
ho-sc-ack	✓	✗	-^{T}	-^{T}	-^{H}
ho-sct-fg	✓	✓	✓	✓	-^{H}
ho-sct-fold	✓	✓	✓^{A}	✓	-^{H}
isabelle-permute	✓	✓	✗	✓	✓
isabelle-f	✓	✗	✗	✓	✓
isabelle-foo	✓	✗	✗	✓	✓
isabelle-bar	✓	✗	✗	✓	✓
isabelle-poly	✓	✗	✗	✗	✗
acl2-fig-2	✓^{O}	✗	✗	✗	✗
acl2-fig-6	✓	✓	✗	✗	✗
acl2-fig-7	✓	✗	✗	✗	✓
lh-gcd	✓	✗	✓	✓	✓
lh-map	✓	✓	✓	✓	-^{H}
lh-merge	✓	✓	✓^{A}	✓	✓
lh-range	✓^{O}	✗	✓^{A}	✗	✓
lh-tfact	✓	✓	✓	✓	✓
dderiv	✓	✓			
deriv	✓	✗			
destruct	✓	✗			
div	✓	✓			
nfa	✓	✓			
scheme	✓	✗			

A: With annotational help   H: Cannot handle higher-order functions
O: With custom partial-order   T: Type system cannot expression untyped program
R: Rewritten using pattern-matching

Table 1. Evaluation on terminating programs

iterations, our dynamic contracts catch the error very early, resulting in immeasurable delay from the start of the program to the point where divergence is detected.

The `nfa` program is particularly interesting, because it is a Scheme benchmark that has been around for decades. The bug was never discovered, because the particular benchmark input did not trigger the divergence, and most static analysis only check for partial correctness. Our analysis was the first to discover this error after many years.

6 RELATED WORK

Our work builds on the size-change termination (SCT) approach [Lee et al. 2001] and on approaches to static contract verification via symbolic execution [Nguyễn et al. 2018, 2014]. We relate our current contributions to dynamic and static termination checking, and then to static contract verification.

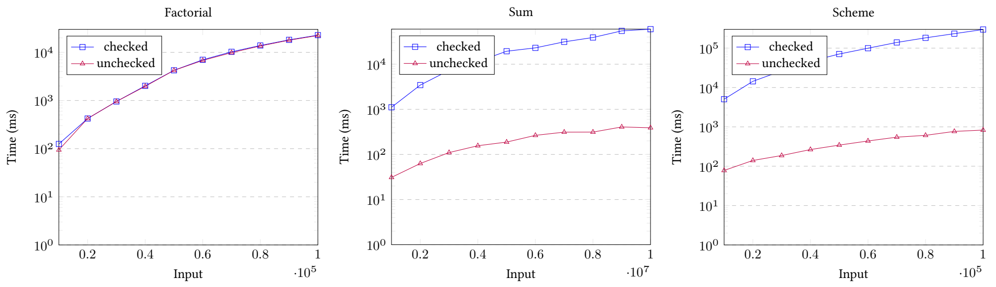

Fig. 10. Slow-down of monitoring `factorial`, `sum`, and `scheme`

6.1 Dynamic termination checking

To the best of our knowledge, no existing work enforces termination dynamically using behavioral contracts. Related work has investigated dynamic loop detection, nontermination auditing, and more restricted declarative query languages.

The auditing tool Looper [Burnim et al. 2009] dynamically monitors a Java program in order to prove nontermination using concolic (concrete and symbolic) execution. Along the potentially nonterminating loop, it derives a path condition paired with a memory map (an encoding of heap values at the end of a loop iteration as a function of their initial values), and uses an SMT solver to check if the initial path condition (after zero iterations) implies itself under the loop iteration's memory map. If this fails, Looper will observe another iteration (or several more) and record a new path condition and memory map. When each path condition implies the next (under that iteration's memory map), in a cyclic chain that terminates with the original (iteration-zero) path condition, the program provably nonterminates.

Unlike our `terminating/c` contract, Looper does not monitor code for nontermination during normal execution; instead, it is deployed by an auditor to determine whether an apparent loop is an actual one. While Looper can provide an affirmative proof that code will not terminate, our approach will signal that a function does not obey SCT, a more conservative notion of termination. This means our approach is susceptible to false positives and may blame functions which do always terminate, but will never permit nontermination. Looper, on the other hand, is susceptible to false negatives and may fail to prove an execution to be definitively nonterminating. Looper's soundness is also contingent on all changes to memory being visible and accounted for in the memory map, which is not necessarily the case in C due to externally visible state and shared-memory parallelism.

Jolt [Carbin et al. 2011] is an infinite-loop detection and recovery tool for C programs. It instruments C code to dynamically monitor for loops that are in the exact same state at two consecutive iterations. Compared with Looper, this is an especially conservative detection for nontermination, however Jolt also has a facility for skipping the program counter past the end of the loop to recover from nontermination and show that this simple technique is effective in many cases (sometimes depending on inputs). Bolt [Kling et al. 2012] extends Jolt with the ability to analyze compiled binaries, bootstrap itself atop running executables, and with multiple recovery strategies.

There are also dynamic termination schemes for more restricted query languages. For example, dynamic checking for active database rules [Bailey et al. 2000], or queries in general logic programs [Codish and Taboch 1997; Shen et al. 2003]. Shen et al. [2003] exploits features unqiue to SLDNF-trees to identify loop goals with a provably finite term-size. Codish and Taboch [1997] provides a declarative fixed-point semantics that captures termination properties (for an interpretation of

Prolog) with the explicit goal of facilitating the extraction of a static analysis using abstract interpretation. The main idea is to use clauses themselves as semantic objects which imply their call patterns—then termination is implied when there is no infinite chain of transitively reachable calls.

6.2 Static termination checking

A wide variety of approaches have previously been used for static verification of termination and for nontermination proving. None of these systems combine dynamic and static verification in a single system, or allow terminating and nonterminating components to be composed using contracts in a user-specified manner.

Jones and Bohr [2004] extend the SCT approach to higher-order languages—specifically, the untyped λ-calculus. As the only values in this language are functions, they select the "height" of a closure as its size. Sereni and Jones [2005] then extended this approach to handle user-defined datatypes and general recursion. This work was not empirically evaluated in the context of a real programming system [Sereni 2006], but establishes techniques we've built on to do termination proving in the context of Racket. SCT has been extended to monotonicity constraints, which have been shown to be more general than traditional SCT [Codish et al. 2005]; it is conceivable these could also be formulated as a dynamic contract in future work.

TNT is a concolic executor for statically enumerating nonterminating *lassos* in C programs—paths that fold back on themselves, forming a nonterminating loop [Gupta et al. 2008]. TNT uses concolic execution with backtracking to systematically explore the execution space of its target, enumerating possible lassos (reachable loops). A second phase then attempts to prove nontermination of each loop by finding a recurrent set of states (assignments to variables) that form an infinite sequence over a non-well-founded relation. A bit-precise solver is used that can take into account machine-dependent characteristics of C programs such as overflow. Unlike dynamic approaches such as ours, or that of Looper, TNT is not statically precise enough to handle cases that rely on symbolic shape information such as cyclicly linked lists.

Velroyen and Rümmer [2008] uses a modal logic allowing predicates to be written that are qualified by a program expression they pertain to. Qualified formulae are trivially rendered true by a diverging program, so a manifest contradiction (i.e., false) being interpreted as true constitutes a proof of nontermination for the qualifying expression. The approach then uses a refinement process to identify the specific conditions on data that will lead to proving this contradiction. It is so-far unclear, however, if this approach can scale as it was implemented for small programs (≤ 25 lines) in a language of pure built-in expressions, assignments, conditionals, and `while` loops.

AProVE is a system for automating termination (and nontermination) proofs of term-rewriting systems (TRSs) Giesl et al. [2006a,b] built using the dependency pair framework [Arts and Giesl 2000; Giesl et al. 2005a]. Unlike previous methods for proving TRSs terminating, which required the right-hand side of each rewrite-rule to be simplified compared with its left-hand side, the dependency pair framework only requires corresponding subterms at recursive calls be simplified. This innovation is analagous to the SCT approach's requirement that arguments be descending over some well-founded order as opposed to static control-flow being strictly stratified. Giesl et al. [2005b] extends the dependency pair framework to higher-order functions. In practice, however, AProVE cannot prove termination of a function that takes integers x and y and recurs until x exceeds y, so it hasn't been successfully scaled to encodings of realistic functional languages [Manolios and Vroon 2006]. The dependency pair framework is contrasted and synthesized with SCT by Thiemann and Giesl [2003].

Terminator [Cook et al. 2006a,b] is a program analysis and verification tool for proving termination of C targets statically. It is a path-sensitive model checker using predicate abstraction and counter-example guided abstraction refinement (CEGAR) [Clarke et al. 2003]. Terminator reduces the problem of proving termination to checking a (disjoined) set of possible rank functions (obtained via Podelski and Rybalchenko [2004a]) using a reachability analysis. It has been used to prove termination of Windows device drivers.

Albert et al. [2008] converts Java bytecode to a rule-based representation which is then abstracted to a constraint logic program (CLP) whose values encode a size metric for the corresponding Java values. Existing techniques for proving CLP termination can then be used directly [Bruynooghe et al. 2007]. This approach consists of inferring both size-change information, as in SCT, and rigidity (sufficient groundedness of terms to ensure that further instantiation will not change its size).

Manolios and Vroon [2006] develops a static analysis for automating termination proofs in the context of the full ACL2 system—a functional language and first-order logic for theorem proving. All programs admitted by ACL2 must be terminating, as nontermination could render it inconsistent, however manual termination proofs are complex and require deep expertise. The paper's approach uses precise calling-context graphs that refine static control flow with path feasibility based on accumulating governors (sets of branch points governing control flow for a subexpression). Strongly connected components are then further refined using a calling-context measure in order to discover a well-founded order over which parameters descend. A major innovation on traditional SCT approaches is the refinement of feasible paths using governors. Our approach analagously tracks path conditions for static verification. Their method was evaluated to successfully prove $> 98\%$ of the more than 10k functions of the ACL2 regression suite terminating. Krauss [2007] then extends the approach to Isabelle/HOL and certifies the termination proofs with LCF-style theorem proving.

LiquidHaskell uses termination proving to ensure precision and soundness for its refinement type system in the presence of lazy evaluation [Vazou et al. 2014a]. Subtle unsoundness can result using refinement types in conjunction with call-by-name evaluation and the direct approach to fixing this unsoundness, by expressing possible nontermination as a type refinement, leads to substantial imprecision. LiquidHaskell bridges this gap by encoding size-change invariants, over user-specified well-founded metrics, directly into the existing type system (as further type refinements). This permits proofs over programs to circularly depend on termination proofs during SMT solving. Broadly this same approach is taken to directly encode termination proofs, via size-change refinements, with dependent types in DependentML [Xi 2002]. LiquidHaskell has a scalable implementation, used to verify correctness and termination properties over a corpus of real-world Haskell libraries ($\geq 10k$ LOC). TEA is also a termination analysis for Haskell, based on techniques of path analysis and abstract reduction [Panitz and Schmidt-Schauß 1997].

Transition invariants [Podelski and Rybalchenko 2004b] represent an automata-theoretic model for proving termination (or liveness), designed to recast SCT in the tradition of model checking for ease of automation. A transition invariant is a well-founded relation (or disjunction of well-founded relations) that subsumes the transitive closure of an automata's state transition relation. Tsitovich et al. [2011] uses loop summarization and transition invariants to verify termination properties of Windows device drivers. Heizmann et al. [2010] gives a detailed comparison of SCT and transition invariants, showing how the approaches may be synthesized. Kuwahara et al. [2014] extends this approach to higher-order languages.

6.3 Soft contract verification

Our static termination checking relies on the ability to go from an operational semantics with dynamic enforcement to a sound static analyzer—a capability we take from a series of results on static contract checking by Nguyễn et al. [2018, 2014]. This work showed that sound higher-order symbolic execution could be leveraged to provide contract-based soft verification and counter-example generation for rich languages including user-defined data structures and contracts as well as higher-order functions and state. We re-use this work by retargeting it to contracts that enforce size-change termination, but otherwise retain the central ideas; it is one of the goals of our work that it compose with existing contract systems.

7 CONCLUSION

Termination is a fundamental program correctness property, but uncheckable even at runtime. To avoid this limitation, we adapt the size-change principle from static termination analysis to perform dynamic checking of termination, exploiting the insight that every infinite execution must have a call that fails to follow the size change principle. This leads to the first run-time mechanism for enforcing termination in a general-purpose programming system. As it is formulated as a behavioral contract, this also makes it the first contract for total correctness. By checking termination as a contract, we can enforce termination in settings where static checking is fundamentally impossible, as in an interpreter.

Further, we compose our dynamic checking strategy with prior work showing how to statically verify compliance with contracts in higher-order languages to produce a novel static checker for program termination. We compare our static checker against three state-of-the-art custom tools on their own benchmarks, and find that ours is able to statically verify programs that exceed the capacaties of each of the existing tools.

Sound dynamic enforcement of liveness properties opens up new possibilities for program correctness, analysis, and specification—in this paper we have taken only the first step.

ACKNOWLEDGMENTS

REFERENCES

Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2008. Termination analysis of Java bytecode. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer, 2–18.

Thomas Arts and Jürgen Giesl. 2000. Termination of term rewriting using dependency pairs. *Theoretical Computer Science* 236, 1-2 (2000), 133–178.

James Bailey, Alexandra Poulovassilis, and Peter Newson. 2000. A Dynamic Approach to Termination Analysis for Active Database Rules. In *Computational Logic — CL 2000*, John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1106–1120.

Maurice Bruynooghe, Michael Codish, John P Gallagher, Samir Genaim, and Wim Vanhoof. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 2 (2007), 10.

Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. 2009. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 161–169.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 209–224.

Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C Rinard. 2011. Detecting and escaping infinite loops with Jolt. In *European Conference on Object-Oriented Programming*. Springer, 609–633.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)* 50, 5 (2003), 752–794.

John Clements and Matthias Felleisen. 2004. A Tail-recursive Machine with Stack Inspection. *ACM Trans. Program. Lang. Syst.* 26, 6 (Nov. 2004).

Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. 2005. Testing for Termination with Monotonicity Constraints. In *Logic Programming*, Maurizio Gabbrielli and Gopal Gupta (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 326–340.

Michael Codish and Cohavit Taboch. 1997. A Semantic Basis for Termination Analysis of Logic Programs and Its Realization Using Symbolic Norm Constraints. In *Proceedings of the 6th International Joint Conference on Algebraic and Logic Programming (ALP '97-HOA '97)*. Springer-Verlag, London, UK, UK, 31–45.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006a. Termination proofs for systems code. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 415–426.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006b. TERMINATOR: beyond safety. In *International Conference on Computer Aided Verification*. Springer, 415–418.

Martin Davis. 1958. *Computability and Unsolvability*. McGraw-Hill.

Robert B. Findler and Matthias Felleisen. 2001. Contract Soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, Vol. 36. ACM.

Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. 2006a. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *International Joint Conference on Automated Reasoning*. Springer, 281–286.

Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. 2006b. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In *Term Rewriting and Applications*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 297–312.

Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. 2005a. The dependency pair framework: Combining techniques for automated termination proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 301–331.

Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. 2005b. Proving and Disproving Termination of Higher-Order Functions. In *Frontiers of Combining Systems*, Bernhard Gramlich (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 216–231.

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving Non-termination. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '08)*. ACM, 147–158.

Matthias Heizmann, Neil D Jones, and Andreas Podelski. 2010. Size-change termination and transition invariants. In *International Static Analysis Symposium*. Springer, 22–50.

Neil D Jones and Nina Bohr. 2004. Termination analysis of the untyped λ-calculus. In *International Conference on Rewriting Techniques and Applications*. Springer, 1–23.

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. 2012. Bolt: on-demand infinite loop escape in unmodified binaries. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 431–450.

Alexander Krauss. 2007. Certified size-change termination. In *International Conference on Automated Deduction*. Springer, 460–475.

Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 392–411.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The Size-change Principle for Program Termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. ACM, New York, NY, USA, 81–92.

R. Majumdar and K. Sen. 2007. Hybrid Concolic Testing. In *29th International Conference on Software Engineering (ICSE)*. IEEE, 416–426.

Panagiotis Manolios and Daron Vroon. 2006. Termination analysis with calling context graphs. In *International Conference on Computer Aided Verification*. Springer, 401–414.

Phúc C Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2018. Soft contract verification for higher-order stateful programs. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* 2, POPL (2018), 51.

Phúc C Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ACM.

Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. 2017. Higher order symbolic execution for contract verification and refutation. *Journal of Functional Programming* 27 (2017). DOI: http://dx.doi.org/10.1017/S0956796816000216

Sven Eric Panitz and Manfred Schmidt-Schauß. 1997. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In *Static Analysis*, Pascal Van Hentenryck (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 345–360.

Andreas Podelski and Andrey Rybalchenko. 2004a. A complete method for the synthesis of linear ranking functions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 239–251.

Andreas Podelski and Andrey Rybalchenko. 2004b. Transition invariants. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on.* IEEE, 32–41.

Koushik Sen. 2007. Concolic testing. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM, 571–572.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes* 30, 5 (2005).

Damien Sereni. 2006. *Termination analysis of higher-order functional programs.* Ph.D. Dissertation. Oxford University.

Damien Sereni and Neil D. Jones. 2005. Termination Analysis of Higher-Order Functional Programs. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, Vol. 3780. Springer.

Yi-Dong Shen, Jia-Huai You, Li-Yan Yuan, Samuel S. P. Shen, and Qiang Yang. 2003. A Dynamic Approach to Characterizing Termination of General Logic Programs. *ACM Trans. Comput. Logic* 4, 4 (Oct. 2003), 417–430.

René Thiemann and Jürgen Giesl. 2003. Size-change termination for term rewriting. In *International Conference on Rewriting Techniques and Applications.* Springer, 264–278.

Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-order symbolic execution via contracts. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 537–554.

Aliaksei Tsitovich, Natasha Sharygina, Christoph M Wintersteiger, and Daniel Kroening. 2011. Loop summarization and termination analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 81–95.

Alan M. Turing. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (1937), 230–265.

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP).* ACM, 51–62.

David Van Horn and Matthew Might. 2011. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Commun. ACM* 54 (Sept. 2011).

Niki Vazou, Eric L Seidel, and Ranjit Jhala. 2014a. From Safety To Termination And Back: SMT-Based Verification For Lazy Languages. *arXiv preprint arXiv:1401.6227* (2014).

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon P. Jones. 2014b. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming.* ACM.

Helga Velroyen and Philipp Rümmer. 2008. Non-termination checking for imperative programs. In *International Conference on Tests and Proofs.* Springer, 154–170.

Christoph Walther. 1994. On Proving the Termination of Algorithms by Machine. *Artif. Intell.* 71, 1 (Nov. 1994), 101–157.

Hongwei Xi. 2002. Dependent types for program termination verification. *Higher-Order and Symbolic Computation* 15, 1 (2002), 91–131.