Size-Change Termination as a Contract

Dynamically and Statically Enforcing Termination for Higher-Order Programs

Anonymous Author(s)

Abstract

Termination is an important but undecidable program property. This situation has led to a large body of work on static methods for conservatively predicting or enforcing termination. One such method is the *size-change termination* approach 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 infinite decreasing sequences.

We transpose these two phases with an operational semantics that accounts for the *run-time enforcement of the size-change property*, postponing (or entirely avoiding) program abstraction. This choice has two key consequences: (1) size-change termination can be checked at run-time and (2) termination can be rephrased as a safety property analyzed using existing methods for systematic abstraction.

We formulate run-time size-change checks as *contracts* in the style of Findler and Felleisen. The result 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 without requiring an abstraction tailored to size-change graphs. Using the higher-order symbolic execution method of Nguyễn *et al.*, we to obtain a termination analysis that is competetive with existing, purpose-built termination analyzers for higher-order languages.

1 Size-change contracts

A fool's errand Imagine for a moment there existed a run-time mechanism for checking whether a program, in its current state, will run forever or eventually terminate. Such a check would be eminently useful. Any run-time mechanism for enforcing partial correctness could easily be made to enforce total correctness by use of this check. Moreover, static verification of termination would boil down to proving these run-time checks always succeed, much like how type systems prove run-time tag checks always succeed.

Of course, whether a program eventually terminates is one of the most useful, yet fundamentally and famously unknowable, properties of programs [15, 46]. Moreover, due to its nature as a liveness property—it cannot be violated in a finite execution—it cannot be directly checked at run-time.

An indirect tack Despite this situation, an indirect partial solution is possible by instead considering a safety property that implies the liveness property. Enforcing such a safety property at run-time would ensure a nonterminating program would eventually "go wrong" by violating the safety property, at which point it could be stopped. The one, unavoidable, wrinkle is that there will be some programs that run astray of the safety property, despite eventually terminating. In this approach, static verification of termination could, as suggested before, be phrased and designed just any other safety verification problem by proving the impossibility of a run-time check failure.

A safety property for termination To design a run-time termination checker, the critical question is: what is a good safety property to enforce that implies termination? One promising candidate is the so-called *size-change principle* of Lee et al. [28]. The principle has already proved useful in static termination checking and has a well understood theory. Unfortunately, the original size-change for termination work, which was developed for static verification, defines the size-change principle as a property of a program *abstraction*: a set of so-called size-change graphs (roughly a program call graph annotated with information about decreasing or non-ascending data flows between function parameters).

This paper We propose a run-time check inspired by the size-change principle for program termination that *dynamically* builds and checks precise size-change graphs. This dynamic mechanism is useful in its own right, but also can be used as a semantic basis for designing static termination checkers. Such static checkers can benefit from advances in static analysis techniques, particularly in abstract interpretation, since termination checks are integrated into the language specification and do not require purpose-built abstractions or algorithms.

We formalize a size-change-principle-enforcing semantics for a core higher-order functional language that ensures all programs terminate (§3). Moreover, we introduce a behavioral software contract, in the style of Findler and Felleisen [16], that enables the selective enforcement of size-change termination. Such a contract, when combined with traditional pre- and post-condition contracts, contributes a notion of contracts for total-correctness.

We also develop a static termination checker (§4) by applying the static contract verification technique of Nguyễn et al. [33] to the size-change semantics. The resulting tool has no termination analysis specific abstractions, it simply treats the size-change principle check as it would any run-time check, and yet an empirical evaluation (§5) demonstrates it is competitive with several state-of-the-art purpose-built termination analyzers: Liquid Haskell, Isabelle, and ACL2.

Contributions This paper contributes:

1. a semantic account of the size-change principle,
2. a proved-correct contract for size-change-based termination of functions,
3. an implementation technique that preserves proper tail-calls and enables tunable run-time overhead,
4. a static termination checker obtained by generic abstract interpretation techniques, and

2 Examples and intuitions

This section develops intuitions for how dynamic checking of *size-change termination* (SCT) works via worked examples. We begin by sketching how SCT works in the original static setting of Lee et al. [28].

2.1 The factorial of termination papers

Consider the Ackermann function, the standard-bearer of examples for papers on termination due its simplicity as a total—but not primitive recursive—function, presented here in Scheme notation:

```
1  (define (ack m n)
2    (cond [(= 0 m) (+ 1 n)]
3          [(= 0 n) (ack (- m 1) 1)]
4          [else    (ack (- m 1)
5                         (ack m (- n 1)))]))
```

For the moment, assume the function is only applied to natural numbers. Under that assumption, ack always terminates and the SCT method suffices to prove it.

Safe size-change graphs: The approach starts by using some form of program analysis or abstract interpretation to enumerate the ways in which a call to ack could result in a subsequent call to ack before returning. We can see there are three potential recursive calls within the function definition on lines 3, 4, and 5. For each of these calls, describe the pairwise relations between the arguments of the call and recursive call in terms of their size. (The original SCT approach assumes the language has only well-founded data types with a known partial order.)

So for example, consider the possible call:

$$\texttt{(ack m n)} \rightsquigarrow \texttt{(ack (- m 1) 1)}.$$

There are two parameters, so we consider four possible size-change relations between the inputs and recursive call. It is clear that the m parameter is strictly smaller in the recursive call compared to the input of the original call. This change is described with a "size-change graph," $\{(\texttt{m} \downarrow \texttt{m})\}$, which is a binary relation saying that whatever value is given for m in the original call will become a strictly smaller argument m in the recursive call. But there is no size-change relation between the original input n and recursive parameter m or n, nor between the original m and recursive n, which we know is 1: each could become larger, smaller, or stay the same.

Moving on to the call in line 5:

$$\texttt{(ack m n)} \rightsquigarrow \texttt{(ack m (- n 1))},$$

we can see that m is unchanged and n is strictly smaller between calls (but there's no relation between m and n), so we describe this call with the graph: $\{(\texttt{m} \,\bar{\downarrow}\, \texttt{m}), (\texttt{n} \downarrow \texttt{n})\}$, which says m is non-ascending and n is descending.

Finally, consider the call in line 4:

$$\texttt{(ack m n)} \rightsquigarrow \texttt{(ack (- m 1) ...)},$$

where the elided code is the nested call to ack of line 5. Here it is clear that m strictly descends, but unclear what happens with n. So we can describe this call with the size-change graph as used for the call in line 3.

At this point, we have a sound collection of size-change graphs for all possible successive calls to ack. They are sound since they properly account for all possible strict descent or non-ascending transitions that occur in recursive calls at run-time. As a side note: it is always safe to omit graph arcs (potentially losing sufficient evidence to prove termination), but all arcs included in a graph must soundly over-approximate all possible run-time behaviors.

Size-change principle: The next task is to check this set of graphs for the *size-change termination principle* (SCP) to see if every infinite computation would give rise to an infinitely decreasing value sequence, according to the size-change graphs. To do this, we consider closing the set of graphs under sequential composition of size-change graphs. The sequential composition of two graphs models two successive calls to construct the size-change from the first to last call, and is defined, informally, as follows: there is a strict descending arc between two parameters, if there exists a path between the parameters containing a strict descent; there is a is a non-ascending arc if there exists a path containing only non-ascent arcs. Otherwise, there is no path.

Coincidentally, the set of graphs for ack is already closed under sequential composition, but to see an example, here's the sequential composition of calling ack on line 3 (or 4) followed by ack on line 5:

$$\{(\texttt{m} \downarrow \texttt{m})\}; \{(\texttt{m} \,\bar{\downarrow}\, \texttt{m}), (\texttt{n} \downarrow \texttt{n})\} = \{(\texttt{m} \downarrow \texttt{m})\},$$

which is equivalent, in terms of size-change, as calling ack on line 3 (or 4).

Once closed, we check each size-change graph to see if it

1. is idempotent, i.e. $g; g = g$, and
2. lacks a self descending arc, i.e. $(x \downarrow x)$ for some parameter x.

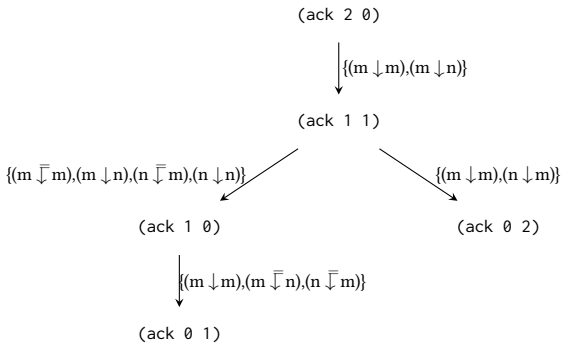

Figure 1. Calls and size changes for `(ack 2 0)`

If such a graph exists, it represents a potential sequence of calls that can be iterated infinitely often with no descent and thus it violates the size-change principle. If it lacks such a graph, the program terminates. In the case of ack both graphs contain self-descending arcs and therefore terminates.

Dynamic size-change graphs: Having established the basic notions of the static SCT approach, we now turn to a dynamic approach to monitoring size-change termination.

The main idea is that rather than rely upon a program analysis to enumerate the various ways a function may call itself, we simply run the program and observe such calls. Each time a function invokes itself, a size-change graph is dynamically generated. Throughout a computation, the call sequence of size-change graphs is accumulated, Before entering a function all, the current call sequence is checked for the size-change principle. If it is violated, the program is stopped and an error signalled; otherwise the call proceeds.

Any program violating the size-change principle eventually accumulates a call sequence witnessing the violation; any program maintaining the principle eventually terminates.

In a similar vein, we need not rely on static analysis to infer the size-change relation between arguments. At runtime, there are concrete values available at both the call and recursive call site. Inferring the size-change graph boils down to checking a partial order on values pairwise on the arguments. This is both easy to do and potentially much more precise than the static approach. For example, there may be size-change relations that hold on the particular path of execution under scrunity, which do not hold in general.

To make things concrete, reconsider ack. When switching perspectives to the dynamic setting, we are no longer concerned with proving termination for all possible executions of the function, but rather with a particular application. Consider `(ack 2 0)`. The complete tree of call sequences and generated size-change graphs are shown in Figure 1, but let us step through its construction. In calling `(ack 2 0)`, control reaches the recursive call on line 3, so we have the call sequence:

$$\texttt{(ack 2 0)} \rightsquigarrow \texttt{(ack 1 1)},$$

from which we can read off the size-change graph. Just as in the static case, we have $(\texttt{m} \downarrow \texttt{m})$, but additionally, we know that $(\texttt{m} \downarrow \texttt{n})$. This fact does not hold in all runs of ack, but it holds in this one.

Aside: it is worth noting that this additional program fact is not necessary in this particular example. After all, we have statically proven ack terminates in all cases using less information. But for the purposes of illustration, we can see that more information is available at run-time; and in principle, it is possible to safely execute size-change terminating programs that are not statically verifiable, just as by analogy it is possible to dynamically monitor type safety of programs that do not trigger run-time type errors, yet are statically ill-typed.

Returning to the example: having generated the graph for this call, we then check the SCT principle for the active sequence of calls; in this case there is just the one graph: $\{(\texttt{m} \downarrow \texttt{m}), (\texttt{m} \downarrow \texttt{n})\}$, which satisfies the size-change property, so execution proceeds.

Now `(ack 1 1)` reaches the `else` branch and first invokes a recursive call to `(ack 1 0)` on line 5. This call generates the graph $\{(\texttt{m} \mathbin{\bar{\downarrow}} \texttt{m}), (\texttt{m} \downarrow \texttt{n}), (\texttt{n} \mathbin{\bar{\downarrow}} \texttt{m}), (\texttt{n} \mathbin{\bar{\downarrow}} \texttt{n})\}$. We now check the size-change graphs of the sequence leading to this point, i.e., the size-change graphs of:

$$\texttt{(ack 2 0)} \rightsquigarrow \texttt{(ack 1 1)} \rightsquigarrow \texttt{(ack 1 0)},$$

and determine if the size-change property holds, which it does. Now `(ack 1 0)` reaches `(ack 0 1)` with graph $\{(\texttt{m} \downarrow \texttt{n}), (\texttt{n} \downarrow \texttt{m}), (\texttt{m} \mathbin{\bar{\downarrow}} \texttt{m}), (\texttt{n} \mathbin{\bar{\downarrow}} \texttt{n})\}$, and the call sequence still satisfies SCP. At this point `(ack 0 1)` terminates with 2. This brings control back to the evaluation of `(ack 1 1)`, which is now ready to proceed to second call to ack on line 4 with the arguments `(ack 0 1)`. At this point, we have the call sequence:

$$\texttt{(ack 2 0)} \rightsquigarrow \texttt{(ack 1 1)} \rightsquigarrow \texttt{(ack 0 2)}.$$

Note the call to `(ack 1 0)` and `(ack 0 1)` are no longer active since they have returned. Again we check the SCP of the size-change graph sequence for active calls, which holds and the program terminates.

A sometimes-buggy Ackermann: We have seen how runtime SCT monitoring works for programs that maintain the size-change principle, but what about buggy programs that do not? Consider the ack example, but change the call on line 4 from `(ack (- m 1) ...)` to `(ack m ...)`. Computing `(ack 2 0)` would proceed as before until reaching the call on line 5, corresponding to the right branch of the tree in Figure 1, i.e. representing the call sequence:

$$\texttt{(ack 2 0)} \rightsquigarrow \texttt{(ack 1 1)} \rightsquigarrow \texttt{(ack 1 2)},$$

whose last size-change graph is now $\{(\texttt{m} \overline{\downarrow} \texttt{m}), (\texttt{n} \overline{\downarrow} \texttt{m})\}$. But this graph is idempotent and contains no self-descents, so at the point of this call a size-change violation is signalled.

2.2 Keeping closures in order

The original formulation of SCT was for a first-order functional language with a well-founded partial order for values. This was done largely to simplify the first phase of static SCT verification where call-graphs and size-change relations are generated. In a higher-order language, computing a call-graph is itself a significant, extensively studied problem [31]. However, in the dynamic formulation, higher-order functions do not pose a serious challenge since calls are observed as they occur.

The one remaining issue concerns the choice of partial order for functions. We make a simple choice and consider all closures to be incomparable. Consequently, no termination proof goes through by an argument about closure size. This is not to say that all programs that use higher-order functions will be rejected by the size-change monitor, just that they must have some descent on base values between calls to the same function. Our empirical evaluation (§5) confirms this is a reasonable choice. To illustrate, let us consider a program that recursively accumulates a closure and eventually applies it in the base case of the function.

Consider a `len` function for lists, written in CPS:

```
1  (define (len l) (loop l (λ (x) x)))
2  (define (loop l k)
3    (cond [(empty? l) (k 0)]
4          [(cons? l)
5           (loop (rest l) (λ (n) (k (+ 1 n))))]))
```

Static analysis of size-change termination relies on an underlying control-flow graph, which must eventually conflate all closures generated on line 5, 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.

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 is arbitrary, they are finite up to the previous loop descending on `l`, which has been proven to terminate. The call sequence for `(len '(2 1))`, which is a sequence of tail-calls:

$$\texttt{(len '(2 1))} \rightsquigarrow \texttt{(loop '(2 1) (λ (x) x))} \rightsquigarrow \texttt{(loop '(1)} \ k_1\texttt{)} \rightsquigarrow \texttt{(loop '()} \ k_2\texttt{)} \rightsquigarrow (k_2\ \texttt{0}) \rightsquigarrow (k_1\ \texttt{1}) \rightsquigarrow \texttt{((λ (x) x) 2)} \rightsquigarrow \texttt{2}$$

The recursive calls of `loop` to itself are easily proven safe through descent on the list. The successive calls to continuations are arbitrarily many but finite. Here k_1 and k_2 stand for different closures of the `(λ (n) (k (+ 1 n)))` term. The computation proceeds to an answer since SCP is checked only between calls to the *same* closure, directly or indirectly.

It is possible to define a partial order on closures, and this may be a worthwhile addition to our approach. For example, Jones and Bohr [23] extend SCT to the untyped λ-calculus and use a partial order based on closure depth to order functions. In theory, this could be used to dynamically order closures in our approach, too, however pragmatically, it requires run-time facilities for "opening" closures [42], which are not typically available.

2.3 Termination and blame

It is useful to assert size-change termination of particular functions, without necessarily asserting termination of the whole program. For this reason, we introduce a contract, `terminating/c`, in the style of Findler and Felleisen [16]. One key component of contract semantics is *blame* to explain the party at fault in contract errors. While our formal model does not represent blame, our implementation does. The addition of blame is at once simple and powerful in the setting of termination contracts. Each use of `terminating/c` marks a blame party, and if the function so wrapped fails to terminate on some call, that location in the program is blamed. No sophisticated run-time machinery is required.

The addition of blame enables a virtuous cycle in program development. If a terminating function `f` calls `g`, then any failure to terminate on the part of `g` will be blamed on `f`. To protect themselves from being blamed, the author of `f` can in turn impose the same contract on `g`, leading to richer specifications and precise errors pinpointing the faulty component. Finally, the provision of size-change termination contracts enables a gradual-typing-style integration of total and partial program components.

2.4 The power of dynamic enforcement

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 Figure 2, we present a λ-calculus implementation that first compiles the term to a procedure and then applies this procedure to an environment. The compilation itself terminates by structural recursion, which is simple to check, but the compilation result is a procedure whose termination is not obvious. In fact, in this example, the first test program $\texttt{c}_1$ terminates when run, but $\texttt{c}_2$ loops infinitely. Dynamic size-change monitoring flexibly allows the first one to finish, and quickly catches the divergence in the second one. The ability to check for termination of specialized programs highlights the advantages of dynamic termination checking.

Execution of $(\texttt{c}_1\ \texttt{(hash)})$ terminates because no function ever calls itself with a non-decreasing argument. In contrast, during the execution of $(\texttt{c}_2\ \texttt{(hash)})$, the compilation result of `(λ (y) (y y))` calls itself (indirectly) with a non-decreasing argument (in this case, identical), hence caught

```
(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 ; Okay
    (comp '((λ (x) (x x)) (λ (y) y)))))
(define c2
  (terminating/c ; Okay
    (comp '((λ (x) (x x)) (λ (y) (y y))))))
(c1 (hash)) ; Okay
(c2 (hash)) ; Error
```

Figure 2. A checked λ-calculus implementation

by the monitoring. As shown in the evaluation section (§5), our implementation is able to confirm the termination of a Scheme interpreter executing merge-sort.

3 Dynamic SCT monitoring

This section introduces language λ_{SCT}, which is λ-calculus, extended with base values and primitive operations, and with a modified semantics ensuring that all programs terminate. Figure 3 shows λ_{SCT}'s syntax and semantics.

3.1 A terminating semantics

The domain of values (v) in λ_{SCT} includes primitives (o), integers (n), pairs (v_1, v_2), and closures ($\text{Clo}(\vec{x}, e, \rho)$). No primitive in λ_{SCT} is allowed to cause divergence.

We present the semantics of λ_{SCT} in Figure 3. The semantics is defined by relation $\rho, m \cup \{\bot\} \vdash e \Downarrow a^{\text{SC}}$, which extends the standard semantics by accumulating a size-change table m. The size-change table maps each function (v) to the most recent arguments it was applied to, in the current dynamic extent, as well as a sequence of size-change graphs ($\vec{g}$) recording ways in which arguments of (v) descend. A size-change graph (g) is a set of arcs of the form $(i \downarrow j)$ or $(i \;\overline{\downarrow}\; j)$, indicating that the i-th argument always strictly descends ($\downarrow$) or never ascends ($\overline{\downarrow}$) to the j-th argument.

An evaluation answer (a^{SC}) can be a value, run-time error (error^{RT}), or size-change error (error^{SC}). A run-time error is one resulting from mis-use of language constructs as standard in a programming language (e.g. applying a primitive to arguments not in its intended domain, applying a non-function, or a function of the wrong arity, etc.). A size-change error is one raised by size-change monitoring upon

$$
\begin{array}{rl}
\text{[Expressions]} & e ::= o \mid b \mid (\lambda\ (\vec{x})\ e) \mid x \mid (e\ \vec{e}) \\
 & \quad \mid (\text{if0}\ e\ e\ e) \\
\text{[Value Literals]} & b ::= 0 \mid -1 \mid 1 \mid \ldots \\
\text{[Primitives]} & o ::= + \mid \text{cons} \mid \text{car} \mid \text{cdr} \mid \ldots \\
\text{[Values]} & v ::= o \mid b \mid (v, v) \mid \text{Clo}(\vec{x}, e, \rho) \\
\text{[Standard Answers]} & a ::= v \mid \text{error}^{\text{RT}} \\
\text{[Answers]} & a^{\text{SC}} ::= a \mid \text{error}^{\text{SC}} \\
\text{[Environments]} & \rho = x \to v \\
\text{[Size-change Table]} & m \in v \rightharpoonup \vec{v} \times \vec{g} \\
\text{[Size-change Graph]} & g \in \mathcal{P}(\mathbb{N} \times r \times \mathbb{N}) \\
\text{[Change]} & r ::= \downarrow \mid \overline{\downarrow}
\end{array}
$$

$$\text{SC-Err}\quad \frac{}{\rho, \bot \vdash e \Downarrow \text{error}^{\text{SC}}} \qquad \text{SC-Prim}\quad \frac{}{\rho, m \vdash o \Downarrow o} \qquad \text{SC-Base}\quad \frac{}{\rho, m \vdash b \Downarrow b}$$

$$\text{SC-Lam}\quad \frac{}{\rho, m \vdash (\lambda\ (\vec{x})\ e) \Downarrow \text{Clo}(\vec{x}, e, \rho)} \qquad \text{SC-Var}\quad \frac{}{\rho, m \vdash x \Downarrow \rho(x)}$$

$$\text{SC-If-T}\quad \frac{\rho, m \vdash e \Downarrow 0 \qquad \rho, m \vdash e_1 \Downarrow a^{\text{SC}}}{\rho, m \vdash (\text{if0}\ e\ e_1\ e_2) \Downarrow a^{\text{SC}}}$$

$$\text{SC-If-F}\quad \frac{\rho, m \vdash e \Downarrow v \text{ where } v \neq 0 \qquad \rho, m \vdash e_2 \Downarrow a^{\text{SC}}}{\rho, m \vdash (\text{if0}\ e\ e_1\ e_2) \Downarrow a^{\text{SC}}}$$

$$\text{SC-App-Clo}\quad \frac{\rho, m \vdash e \Downarrow \text{Clo}(\vec{x}, e', \rho') \qquad \rho, m \vdash \overrightarrow{e_x} \Downarrow \overrightarrow{v_x} \qquad \rho'[\vec{x} \mapsto \overrightarrow{v_x}], \mathit{update}(m, \text{Clo}(\vec{x}, e', \rho'), \overrightarrow{v_x}) \vdash e' \Downarrow a^{\text{SC}}}{\rho, m \vdash (e\ \overrightarrow{e_x}) \Downarrow a^{\text{SC}}}$$

Figure 3. Syntax and semantics of λ_{SCT}.

detecting a size-change violation. We omit rules that introduce run-time errors and error propagation, as they are entirely standard and not the focus of this paper.

Rule *[SC-App-Clo]* shows application of a closure. In λ_{SCT}, all applications are enforced to have the size-change property. Before executing the function's body as in the standard semantics, we update the size-change table and guard against a violation to the size-change property. Helper function *upd* updates the size-change table with the function's latest arguments and size-change graph, potentially returning $\bot$ if there is a size-change violation. If *upd* does not return a table, the evaluation aborts with an error as in rule *[SC-Err]*.

3.2 Updating and monitoring size-change graphs

Figure 4 lists helper functions that update and monitor SCT.

Function *upd* takes the size-change table (m), function (v), and its latest arguments ($\vec{v}_n$). It computes a new size-change graph (g_n) for the transitions from the previous arguments ($\vec{v}_{n-1}$) to these new arguments, ensures that the new graph sequence ($g_n :: \overrightarrow{g_{n-1}}$) does not violate the size-change property, and then updates the graph in m. If function v has not been applied before and there is no entry in m, a trivial entry with the current argument list as well as the empty graph sequence is stored.

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 observed to be strictly smaller than some value v_i at index i in the former list, an arc $(i \downarrow j)$ is included in the graph. When the values are equal, we include $(i \overline{\downarrow} j)$ instead.

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

Finally, predicate *prog?* checks for the lack of violation to the size-change termination principle: a graph sequence $g_n \ldots g_1$ violates the size-change termination principle if there exists a sub-sequence $g_i; \ldots; g_j$ (where $1 \le i \le j \le n$) that is both idempotent and lacking of an strict descending arc of a parameter to itself.

3.3 Well-founded partial order

Figure 5 shows an example of a well-founded partial order ($\preceq$) on values in λ_{SCT}. It is defined on integers by comparing absolute values, and a field of a data-structure is considered smaller than any data-structures that contain it (i.e., the tail of any list is considered *less than* than the original list). 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 Totality of evaluation

We may note that all programs in λ_{SCT} terminate, either by adhering to the size-change principle, of by violating it and aborting with an error.

Theorem 3.1 (Termination of λ_{SCT}). *For all e, ρ, m, where $fv(e) \subseteq dom(\rho)$, $\rho, m \vdash e \Downarrow a^{\text{SC}}$ for some a^{SC}.*

3.5 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:

$$
\begin{aligned}
\mathit{upd} &: m \times v \times \vec{v} \to m \cup \{\bot\} \\
\mathit{upd}(m,v,\vec{v}_n) &= m[v \mapsto (\vec{v}_n, [])], \text{if } v \notin m \\
\mathit{upd}(m,v,\vec{v}_n) &= \begin{cases} m[v \mapsto (\vec{v}_n, g_n :: \vec{g}_{n-1})] \\ \quad \text{if } \mathit{prog?}(g_n :: \vec{g}_{n-1}) \\ \bot \quad \text{otherwise} \end{cases} \\
&\text{where } (\vec{v}_{n-1}, \vec{g}_{n-1}) \equiv m(v) \\
&\text{and } g_n = \mathit{graph}(\vec{v}_{n-1}, \vec{v}_n)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{graph} &: \vec{v} \times \vec{v} \to g \\
\mathit{graph}(\vec{v}, \vec{v}') &= \{(i \downarrow j) \mid v_i \in \vec{v}, v_j \in \vec{v}', v_j \prec v_i\} \\
&\cup \{(i \overline{\downarrow} j) \mid v_i \in \vec{v}, v_j \in \vec{v}', v_j = v_i\}
\end{aligned}
$$

$$
\begin{aligned}
(;) &: g \times g \to g \\
g_0 ; g_1 &= \{(i \downarrow k) \mid (i \downarrow j) \in g_0, (j\ r\ k) \in g_1\} \\
&\cup \{(i \downarrow k) \mid (i\ r\ j) \in g_0, (j \downarrow k) \in g_1)\} \\
&\cup \{(i \overline{\downarrow} k) \mid (i \overline{\downarrow} j) \in g_0, (j \overline{\downarrow} k) \in g_1, \\
&\quad \forall j.(i r_0 j) \in g_0, (j r_1 k) \in g_1 .\ r_0 = r_1 = \overline{\downarrow}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{prog?} &: \vec{g} \to \mathbb{B} \\
\mathit{prog?}(g_n \ldots g_1) &= \textstyle\bigwedge_{1 \le i \le j \le n} \mathit{desc?}(g_i; \ldots; g_j)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{desc?} &: g \to \mathbb{B} \\
\mathit{desc?}(g) &= (g = g; g) \implies \exists i.(i \downarrow i) \in g
\end{aligned}
$$

Figure 4. Updating and monitoring size-change

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

Figure 5. Example well-founded partial order $\preceq$

- soundness: if a program evaluates to a value under the modified semantics, running that program without termination checking gives the same result.
- completeness: if a program size-change terminates to a value under the standard semantics, running that program under the modified semantics with termination checking gives the same result.

In addition, because all programs terminate under the modified semantics when termination checking is enabled, all diverging programs are caught as error-raising programs.

We now formally establish the correctness of λ_{SCT}'s size-change monitoring semantics with respect to its standard semantics.[1]

[1] λ_{SCT}'s standard dynamic semantics is unsurprising and can be found in the appendix.

$$\textsc{CC-Base}\quad \frac{}{\rho, m \vdash b \Downarrow b, \{m\}}$$

$$\textsc{CC-App}\quad \frac{\begin{array}{c}\rho, m \vdash e \Downarrow \mathsf{Clo}(\vec{x}, e', \rho'), \{m' \ldots\} \\ \rho, m \vdash \overrightarrow{e_x} \Downarrow \overrightarrow{v_x}, \overrightarrow{\{m_x \ldots\}} \\ \rho'[\overrightarrow{x \mapsto v_x}], \mathit{ext}(m, \mathsf{Clo}(\vec{x}, e', \rho'), \overrightarrow{v_x}) \vdash e' \Downarrow a, \{m'' \ldots\}\end{array}}{\rho, m \vdash (e\ \overrightarrow{e_x}) \Downarrow a, \{m' \ldots\} \cup \overrightarrow{\{m_x \ldots\}} \cup \{m'' \ldots\}}$$

$$\begin{array}{rcl} \mathit{ext} & : & m \times v \times \vec{v} \to m \\ \mathit{ext}(m, v, \vec{v}_n) & = & m[v \mapsto (\vec{v}_n, [])], \text{if } v \notin m \\ \mathit{ext}(m, v, \vec{v}_n) & = & m[v \mapsto (\vec{v}_n, g_n :: \vec{g}_{n-1})] \\ \text{where} & & (\vec{v}_{n-1}, \vec{g}_{n-1}) \equiv m(v) \\ \text{and} & & g_n = \mathit{graph}(\vec{v}_{n-1}, \vec{v}_n) \end{array}$$

Figure 6. Call-sequence Semantics of λ_{CSCT}.

Theorem 3.2 (Soundness of size-change monitoring in λ_{SCT}). *If $\rho, m \vdash e \blacklozenge a$, then $\rho \vdash e \Downarrow a$.*

Proof. By induction on the derivation of $\rho, m \vdash e \blacklozenge a$. □

Corollary 3.3 (Size-change monitoring catches divergence). *If program e diverges under the standard semantics, then $\{\}, \{\} \vdash e \blacklozenge \text{error}^{\text{SC}}$.*

Proof. From Lemma 3.1, e either evaluates to a standard answer or error^{SC} under size-change monitoring. By contrapositive of Theorem 3.2, e evaluates to error^{SC} if e diverges. □

A semantics that evaluates to call sequences Before stating and proving completeness of size-change monitoring, we define a mostly-standard semantics that also evaluates to set of size-change tables along with the answer, but performs no guarding against any size-change violation. It is in lock-step with the standard semantics, and resembles the terminating semantics in accumulating the size-change table. Figure 6 shows this semantics.

Lemma 3.4 (Completeness of call-sequence semantics). *If $\rho \vdash e \Downarrow v$ then $\rho, \{\} \vdash e \Downarrow v, \{m \ldots\}$ for some $\{m \ldots\}$.*

Proof. By induction on the derivation of $\rho \vdash e \Downarrow v$. □

Lemma 3.5 (Completeness of size-change monitoring with respect to call-sequence semantics). *If $\rho, m \vdash e \blacklozenge \text{error}^{\text{SC}}$ and $\rho, m \vdash e \Downarrow v, \{m' \ldots\}$ then there exists m_i in $\{m' \ldots\}$ and v such that $\neg \mathit{prog?}(g)$ where $(\overrightarrow{v_x}, g) = m_i(v)$.*

Proof. By induction on the derivation of $\rho, m \vdash e \blacklozenge \text{error}^{\text{SC}}$. □

Theorem 3.6 (Completeness of size-change monitoring in λ_{SCT}). *If $\rho, \{\} \vdash e \blacklozenge \text{error}^{\text{SC}}$ and $\rho \vdash e \Downarrow v$ then $\rho, \{\} \vdash e \Downarrow v, \{m \ldots\}$ such that there exists m_i in $\{m \ldots\}$ and v such that $\neg \mathit{prog?}(g)$ where $(\overrightarrow{v_x}, g) = m_i(v)$.*

Proof. Follows from Lemma 3.4 and Lemma 3.5. □

$$\begin{array}{rl} \text{[Expressions]} & e ::= \ldots \mid (\mathsf{term/c}\ e) \\ \text{[Values]} & v ::= \ldots \mid \mathsf{term/c}(\mathsf{Clo}(\vec{x}, e, \rho)) \end{array}$$

$$\textsc{Wrap-Lam}\quad \frac{\rho \vdash e \Downarrow \mathsf{Clo}(\vec{x}, e, \rho)}{\rho \vdash (\mathsf{term/c}\ e) \Downarrow \mathsf{term/c}(\mathsf{Clo}(\vec{x}, e, \rho))}$$

$$\textsc{App-Term}\quad \frac{\begin{array}{c}\rho \vdash e \Downarrow \mathsf{term/c}(\mathsf{Clo}(\vec{x}, e', \rho')) \qquad \rho \vdash \overrightarrow{\rho_x} \Downarrow \overrightarrow{v_x} \\ \rho'[\overrightarrow{x \mapsto v_x}], \mathit{upd}(\{\}, \mathsf{Clo}(\vec{x}, e', \rho'), \overrightarrow{v_x}) \vdash e' \blacklozenge a^{\text{SC}}\end{array}}{\rho \vdash (e\ \overrightarrow{e_x}) \Downarrow a^{\text{SC}}}$$

Figure 7. Syntax and semantics of λ_{CSCT}.

3.6 Termination checking as a contract

It can be useful to enforce termination checking selectively on parts of the code rather than on the entire program. We present a simple extension to λ_{SCT} called λ_{CSCT}, which adds a construct $(\mathsf{term/c}\ e)$ that guards (e) with a contract ensuring it behaves as a size-change-terminating function. Other than executing the bodies of contract-guarded functions, λ_{CSCT}'s semantics is similar to that of the standard semantics. Figure 7 shows the key extension to λ_{CSCT}'s syntax and semantics.

Rule *[Wrap-Lam]* shows the introduction of a termination-checked function. Only closures are capable of violating SCT in λ_{SCT}, so we only wrap closures and return other values as-is.

4 Static SCT Verification

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 [33, 44, 48].

Symbolic execution extends the standard semantics with symbolic values that can stand for any values (including higher-order values), and maintains 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.

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

SYM-IF-T

$$\frac{\rho, \phi \vdash e \Downarrow_s s, \phi' \qquad \rho, (= s\ 0) :: \phi' \vdash e_1 \Downarrow_s a^{\mathrm{SC}}, \phi''}{\rho, \phi \vdash (\mathsf{if0}\ e\ e_1\ e_2) \Downarrow_s a^{\mathrm{SC}}, \phi''}$$

SYM-IF-F

$$\frac{\rho, \phi \vdash e \Downarrow_s s, \phi' \qquad \rho, (\neq s\ 0) :: \phi' \vdash e_2 \Downarrow_s a^{\mathrm{SC}}, \phi''}{\rho, \phi \vdash (\mathsf{if0}\ e\ e_1\ e_2) \Downarrow_s a^{\mathrm{SC}}, \phi''}$$

Figure 8. Semantics of symbolic λ_{SSCT}.

Although symbolic execution has traditionally been used to find bugs [6, 24, 29, 37, 38] 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 [14, 47] and obtain a verification that particular errors cannot occur at run-time.

4.1 Extended semantics

Figure 8 shows extension to λ_{SCT}, called λ_{SSCT} that allows symbolic execution, as well as the key extension to the semantics that enables symbolic execution.

We extend the set of values (v) with *symbolic values* (s), which can stand for any value. The semantics of λ_{SSCT} must then account for symbolic values, which means some expressions can nondeterministically evaluate to multiple answers to soundly over-approximate all the cases resulting 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 4 have, in general, no more arcs than in the concrete case as each arc now represents a must-descent or must-non-ascent relationship over all possible concrete paths. A sufficiently precise symbolic execution, coupled with effective SMT solving, can maintain a graph with enough arcs to prove that functions will always maintain their size-change properties.

Proposition 4.1 (Soundness of static verification). *If* $\{\} \vdash e \Downarrow v$ *and* $\{\} \vdash e_1 \Downarrow v_1$ *and* $(e\ e_1)$ *diverges, then* $\{\}, \{\} \vdash ((\mathsf{term/c}\ e)\ s) \Downarrow_s \mathsf{error}^{\mathrm{SC}}, \phi'$ *(*s *is a fresh symbolic value).*

Proof. Follows from soundness of dynamic checking of size-change termination (Theorem 3.2) and soundness of higher-order symbolic execution [32]. □

4.2 Ackermann revisited

Now consider again the example `ack`, a termination-checked Ackermann function shown in Section 2.

Suppose `ack`'s precondition is that its arguments are natural numbers. 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.

The first branch simply returns and does not trigger any size-change monitoring. The second branch reaches a recursive call with the path-condition `{(≥ m 0),(≠ m 0),(= n 0)}`. The recursive call proceeds, checking for all relationships that can be established between the old and new arguments as in Figure 4. In this case, with the path-condition that `m` is positive, symbolic execution easily proves that `(- m 1)` is less than `m` according to the partial order defined in Figure 5. 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 $\{(m \downarrow m)\}$ [2]. We extend `ack`'s set of size-change graphs with this new graph. In addition, symbolic execution can prove the new call to `ack` receives the same path-condition as the previous call: both new arguments `(- m 1)` and `1` are natural numbers.

The third branch reaches the inner recursive call to `ack` before reaching the outer one. The path-condition, again, is sufficient to establish the descent from `n` to `(- n 1)` and maintenance of `m`, yielding the new graph $\{(m \overline{\downarrow} m), (n \downarrow n)\}$. When execution reaches the outer recursive call to `ack`, the descent from `m` to `(- m 1)` can be straightforwardly established. In each case, symbolic execution can also prove that the new arguments are natural numbers.

Figure 9 summarizes all the ways `ack` can call itself recursively. Because no composition of size-change graphs drawn from this set can yield a graph that violates the size-change principle (i.e. one that is both idempotent and lacking of a self-descent arc), `ack` never violates size-change termination.

5 Implementation and evaluation

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 [SC-App-Clo] in Figure 3 [App-Term] in Figure 7. If

[2] We use variable names instead of indices for graph nodes in this section

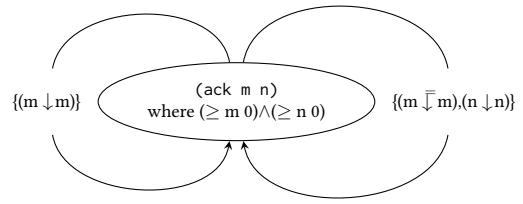

Figure 9. Abstract call and size-change graphs for ack.

size-change termination is being enforced, the #%app form looks up the size-change table to guard against violations.

There are two possible techniques of maintaining the size-change table. The first technique is wrapping each application with code that imperatively updates and restores the table. The second technique is through continuation-marks [9]. The former can be implemented in most languages, and gives relatively good performance, but breaks tail calls. The latter is simple to implement in languages with support for continuation marks, and preserves tail calls, but shows high overheads in tight loops.

Our semantics implicitly assumes that closures can be compared structurally for equality, which is not possible in practice. We instead hash the closure and consider all closures with the same hash code to be equivalent. This preserves soundness as the table m cannot grow infinitely, but could produce false positive error reports. Note that this incompleteness does not affect the static analysis, which is derived from the semantics itself. Future work includes runtime support for more precise comparison between closures.

In addition, we expose a parameter specifying the custom partial order for use in termination checks, with a default implementation as described in Figure 5.

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 and checking 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 use exponential backoff to reduce the frequency of extend and monitoring each function's size-change. This significantly reduces the monitoring overhead, although risks keeping data from earlier iterations live for longer than would be otherwise.

Avoiding instrumentation for known functions In many cases, functions that are known to terminate need no instrumentation. For example, we maintain a white-list of primitives known to 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 constructed and monitored.

5.1 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.

5.1.1 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: size-change termination for first-order programs (sct) [28]; size-change termination for higher-order programs (ho-sct) [40]; Liquid Haskell (lh) [50]; Isabelle [26]; ACL2 [30]; and 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 (e.g. lh-merge), or a custom partial-order (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, current versions of the tools cannot check some of their own benchmarks despite our best efforts to reproduce (e.g. isabelle-poly for Isabelle; 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).

Program	Dyn.	Static	LH	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-perm	✓	✓	✗	✓	✓
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	✓	✓	A: With annotations		
deriv	✓	✗	O: Custom partial order		
destruct	✓	✗	H: No H.O. functions		
div	✓	✓	T: Not typable		
nfa	✓	✓	R: Rewritten to use		
scheme	✓	✗	pattern matching		

Table 1. Evaluation on terminating programs

Figure 10 shows the slowdown when using dynamic checks for select programs: `factorial`, `sum`, and `merge-sort`, as well as their interpreted version inside a Scheme interpreter. 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, or the Scheme interpreter doing significant work, overhead is negligible. For programs that consist mainly of tight loop such as `sum` or repeatedly loop on large data-structures such as `merge-sort`, overhead is much more significant. That the overhead stays fixed when the input grows (for continuation-mark implementation on tight loops, approximately two orders of magnitude) suggests that further optimization effort to trim down the constant factor can make monitoring suitable for realistic uses.

5.1.2 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 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. It is a program that implements a non-deterministic finite automaton of the regular expression `((a|c)*bcd)|(a*bc)`, then run the automaton on the string $a^{133}bc$. The following function implements one state recoginizing the sub-expression `(a|c)*` with the bug underlined:

```
1  (define (state1 input)
2    (and (not (null? input))
3         (or (and (char=? (car input) #\a)
4                  (state1 (cdr input)))
5             (and (char=? (car input) #\c)
6                  (state1 input))
7             (state2 input))))
```

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 static analysis was the first to discover this error after many years.

6 Related work

Our work builds on the size-change termination (SCT) approach [28] and on approaches to static contract verification via symbolic execution [32, 33]. We relate our contributions to dynamic and static termination checking, and then to static contract verification.

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 languages.

The auditing tool Looper [5] dynamically monitors a Java program in order to prove nontermination using concolic (concrete and symbolic) execution. Along the potentially non-terminating 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.

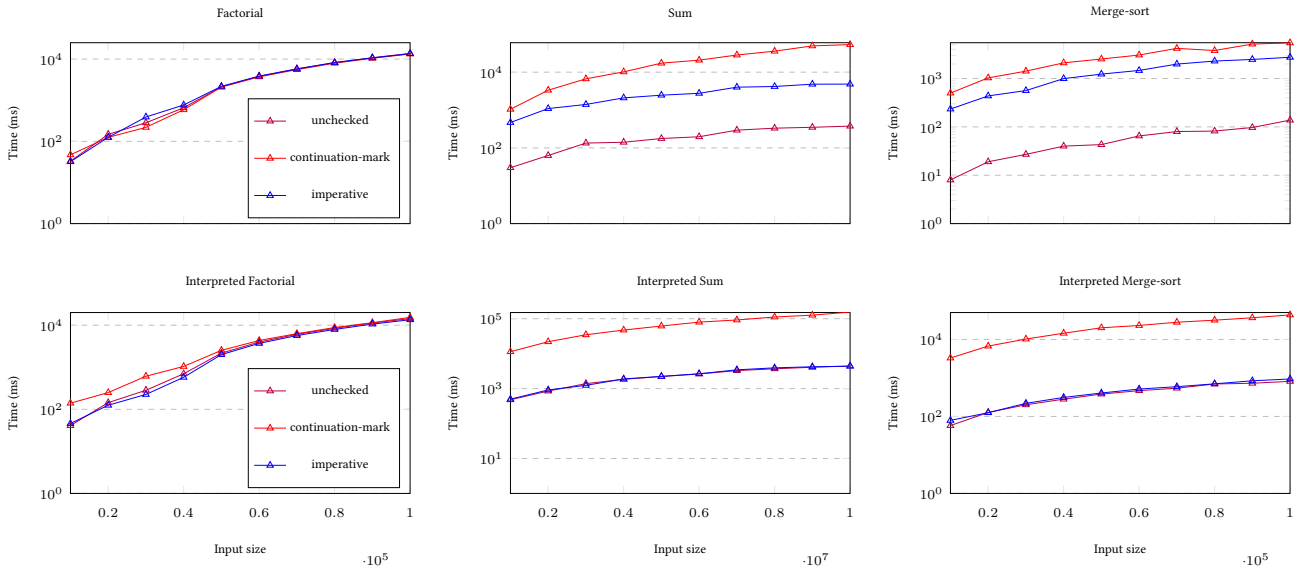

Figure 10. Slow-down of monitoring `factorial`, `sum`, and `merge-sort`, and the Scheme interpreter running them

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 always the case in C due to external state and shared-memory parallelism.

Jolt [7] (and successor Bolt [25]) 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).

There are also dynamic termination schemes for more restricted languages. For example, dynamic checking for active database rules [3], or queries in general logic programs [11, 41]. Shen et al. [41] exploits features unqiue to SLDNF-trees to identify loop goals with a provably finite term-size. Codish and Taboch [11] 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.

6.2 Static termination checking

A variety of approaches have been used for static verification of (non-)termination. None of these systems combine dynamic and static verification in a single system, or allow terminating and nonterminating components to be composed. We begin with the systems we compare with in §5.1.

Jones and Bohr [23] 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 [40] 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 [39], but establishes techniques we build on. SCT has been extended to monotonicity constraints, which have been shown to be more general than traditional SCT [10]; these could be formulated as a dynamic contract in future work.

Manolios and Vroon [30] 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 [26] 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 [49]. 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 [52]. 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 [34].

TNT is a concolic executor for statically enumerating nonterminating *lassos* in C programs—paths that fold back on themselves, forming a nonterminating loop [21]. 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 cyclic lists.

Velroyen and Rümmer [51] use 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. This system was only evaluated on small expressions (≤ 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. [17, 18] built using the dependency pair framework [2, 19]. 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 [30]. The dependency pair framework is contrasted and synthesized with SCT by Thiemann and Giesl [43].

Numerous techniques have been proposed and evaluated for verifying termination in languages such as C and Java, where higher-order programming is uncommon. Terminator [12, 13, 35] and transition invariants [22, 36, 45] as well as others [1] have seen extensive development over the past decade, but differ substantially in methods, goals, and language from our system.

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. [32, 33]. 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 a goal of our work that it composes 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—without any termination-specific work. 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.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

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

[14] David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 12 (Aug. 2017), 25 pages. DOI: http://dx.doi.org/10.1145/3110256

[15] Martin Davis. 1958. *Computability and Unsolvability*. McGraw-Hill.

[16] Robert B. Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*. ACM.

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

[18] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. 2006. 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.

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

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

[21] 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.

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

[23] 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.

[24] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976).

[25] 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.

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

[27] 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.

[28] 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.

[29] R. Majumdar and K. Sen. 2007. Hybrid Concolic Testing, In Software Engineering, 2007. ICSE 2007. 29th International Conference on. *Software Engineering, 2007. ICSE 2007. 29th International Conference on* (2007).

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

[31] Jan Midtgaard. 2012. Control-flow Analysis of Functional Programs. *ACM Comput. Surv.* 44, 3 (June 2012).

[32] 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.

[33] 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.

[34] 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.

[35] Andreas Podelski and Andrey Rybalchenko. 2004. A complete method for the synthesis of linear ranking functions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*.

Springer, 239–251.

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

[37] Koushik Sen. 2007. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM.

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

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

[40] 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.

[41] 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.

[42] Jeffrey Mark Siskind and Barak A. Pearlmutter. 2007. First-class Nonstandard Interpretations by Opening Closures. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 71–76. DOI: http://dx.doi.org/10.1145/1190216.1190230

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

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

[45] 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.

[46] 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.

[47] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM.

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

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

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

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

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