Certified Size-Change Termination

Alexander Krauss

Technische Universität München, Institut für Informatik
`http://www.in.tum.de/~krauss`

Abstract. We develop a formalization of the Size-Change Principle in Isabelle/HOL and use it to construct formally certified termination proofs for recursive functions automatically.

1 Introduction

Program termination plays an important role in verification, and in particular in theorem provers based on logics of total functions, where termination proofs are usually necessary to ensure logical consistency.

Although there has been continuous progress in the field of automated termination proofs, only few of the results have been applied to interactive theorem proving. One possible reason is that many existing methods are relatively complex, often combining several different criteria and heuristics. Another is that they do not usually produce proofs that can be checked by an independent system. This makes their integration difficult, especially when following the LCF approach, where all inferences must be checkable by a minimal logical core.

In this paper, we formalize the size-change principle [14] and prove it correct in Isabelle/HOL [16]. Then we apply it to recursive function definitions in the logic itself, essentially following an approach by Manolios and Vroon [15], but with full proofs. We integrate the results to a fully automated proof procedure to certify size-change termination of Isabelle/HOL functions.

To our knowledge, this is the first formalization of the size-change principle, and also the first mechanically verified implementation. Our results show that it is practically feasible to combine the power of state-of-the-art termination criteria with the high assurance of LCF-style theorem proving. Moreover, we think that the formalization also gives a better insight in the structure of termination proofs, and in particular in the relation between the analysis of Manolios and Vroon and the size-change principle.

As a practical benefit, a significant class of previously hard termination proofs are now automatic.

1.1 Size-Change Termination - abstractly

"A program is size-change terminating iff every infinite execution of the program would cause an infinite descent in some well-founded data value." Although its first presentation by Lee, Jones and Ben-Amram [14] was in the context of a

simple functional language, this criterion, called *size-change termination* (SCT), is independent from the actual language or programming paradigm used.

We will emphasize this generality, which leads to a neat abstraction boundary in our formalization, by using slightly more general terminology than the original paper.

SCT abstracts from the actual program by viewing it as a set of *control points* and transitions between them, forming a directed graph (the *control graph*). Each control point has a finite set of abstract *data positions* associated to it, which can be seen as slots, where runtime data is passed around.

Each transition is labeled by a *size-change graph*, which contains information about how the values in the data positions are related. The size-change graph contains an edge $p \xrightarrow{\downarrow} q$, if the value at data position q (after the transition) is always smaller than the value at position p (before the transition), and $p \xrightarrow{\overline{\downarrow}} q$ if it is smaller or equal. Size-change graphs are usually drawn as bipartite graphs. Fig. 1 shows a simple graph with two control points A and B, with two and three data positions.

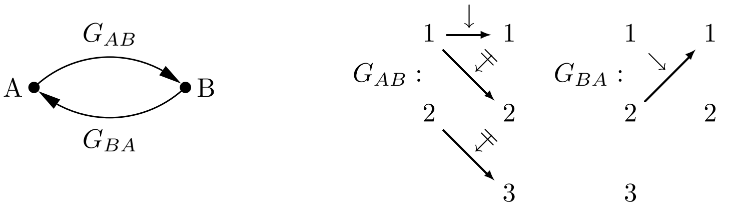

Fig. 1. A simple size-change problem

By connecting the size-change graphs along a control flow path, the data flow becomes visible. Chains of such connected edges are called *threads*. A thread has infinite descent iff it contains infinitely many $\xrightarrow{\downarrow}$-edges.

Definition. *A control graph $\mathcal{A}$ satisfies SCT iff every infinite path has a thread with infinite descent.*

The example in Fig. 1 satisfies SCT, since the only infinite path is $A, B, A, B, \ldots$ and it has a thread going through data positions $1, 2, 1, 2, \ldots$, which has infinite descent.

SCT is decidable:

Theorem. *A control graph $\mathcal{A}$ satisfies SCT iff for every edge in $\mathcal{A}^+$ of the form $n \xrightarrow{G} n$ with $G = G \cdot G$, G has an edge of the form $p \xrightarrow{\downarrow} p$.*

Here, $\mathcal{A}^+$ denotes the transitive closure of $\mathcal{A}$, where the composition of two graphs is defined in the obvious way (for details see §2). This theorem suggests an algorithm which simply computes the transitive closure and checks the above property.

Since SCT is a purely combinatorial graph problem, generating size-change problems from programs is a separate issue.

Here lies the power of the abstraction: Since nothing is said about what the control points and data positions actually are, we can talk about different types of programs. The original paper treated simple functional programs, and used functions as control points. Function calls were the transitions, and the data positions were given by the sizes of the function arguments. For imperative programs, one could take program instructions as control points and program variables as data positions.

Other interpretations are equally valid, as long as (a) infinite executions of the program correspond to infinite paths in the control graph, and (b) the information in the size-change graphs reflects actual size-changes in some well-founded data. Then a non-terminating execution would imply an infinitely decreasing sequence of data values, which is impossible.

Since the $\xrightarrow{\overline{\overline{\downarrow}}}$- and $\xrightarrow{\downarrow}$-edges in a size-change graph reflect knowledge about the data flow in the program, a suitable analysis is required to derive this information. The authors of [14] apparently had some syntactic size analysis in mind, but in fact we have the choice of weapons here, and we choose theorem proving, which does very well on this task.

1.2 Function definitions and termination proofs in Isabelle/HOL

Recursive functions are defined in Isabelle/HOL following the definitional approach: An automated package [13] transforms the recursive specification into a non-recursive form, which can be processed by existing means. Then the original specification is proved from this definition. Internally, the package constructs the call relation of the function and a domain predicate characterizing values where the function terminates.

Although the package has some support for partial (i.e. non-terminating) functions (for details, see [13]), reasoning with partial functions is more complicated than with total ones. Specifically, the recursive equations are constrained by the domain predicate.

Now “proving termination” just means showing that the domain predicate is always true or, equivalently, that the call relation is wellfounded.

1.3 No simple certificates

Things would be simpler and more elegant, if we could just generate short certificates of some kind, which can be easily checked and which prove that a function is size-change terminating. Then just the checking would have to be proved correct and executed in the theorem prover, while the certificates could be generated by untrusted (but probably more efficient) code.

However, by a complexity argument, such certificates are unlikely to exist, due to the PSPACE-hardness result for SCT [14]:

Corollary 1. *If there were certificates proving $x \in SCT$ that could be checked in polynomial time, then $PSPACE = NP$, which complexity theorists find unlikely [18].*

Proof. Assume such certificates exist, then SCT $\in$ NP by a simple guess-and-check argument. But SCT is PSPACE-hard, thus PSPACE $\subseteq$ NP.

This result shows that size-change termination is fundamentally different from many other methods in that it does not produce simple and short termination arguments (like simple wellfounded relations). Instead, it is more like an exhaustive search for possible sources of non-termination, ruling them out systematically.

1.4 Related Work

The quest for automated termination proofs is continously receiving a large amount of attention, way too much to be cited here.

But when it comes to full formal certification of termination proofs, the air gets thin: Termination proofs in major proof assistants like Coq [4], PVS [17] and Isabelle are usually based on user-specified measure functions. HOL4 [7], HOL Light [10] and ACL2 [11] support a rudimentary automatic guessing of measures.

Recently, Manolios and Vroon [15] successfully combined the size-change principle with theorem proving to obtain a powerful termination checker for the ACL2 system. They make the following modifications to standard SCT:

- Instead of using the functions as control points, they used the function calls and also take the context of a recursive call into account. This allows to analyze reachability between calls.
- Instead of using a syntactic analysis to generate size-change graphs, they use the ACL2 prover.
- Instead of just looking at the size of concrete data values, they are able to use arbitrary measure functions.
- In an additional processing step, calls can be substituted into one another. This step (called *context merging*) allows for a limited treatment of problems where a temporary increase of data happens.

However, the non-trivial analysis is part of the trusted code base and even if the metatheory is sound, it is not clear if it can be justified within the first-order framework of ACL2. In this paper, we essentially follow their approach (excluding context merging), but we formally verify both the underlying theory and the implementation, which allows us to produce Isabelle proofs.

The CoLoR project [5] aims to provide the formal basis and the tools to certify termination proofs in Coq. Proofs can be imported from various other systems, all from the area of term rewriting. However, since these tools only work on a formalization of term rewriting inside Coq, they cannot easily be applied to Coq function definitions.

1.5 Overview of this paper

In §2, we describe a formalization of the size-change principle. We formalize Kleene algebras, graphs, paths and threads and define the SCT predicate. Then we present the main theorem, which states the equivalence between the declarative and the algorithmic version of SCT.

In §3, we apply the principle to Isabelle function definitions: We formalize what it means that a control graph approximates a relation. Then we show that for such an approximation, the size-change property implies wellfoundedness of that relation.

It then remains to provide an algorithm for building and inspecting the transitive closure of a graph. In §4 we give a simple implementation and prove it correct. From these three ingredients we obtain a fully automated method to prove termination of recursive functions in Isabelle.

We present some small example applications in §5 and discuss practical implications in §6.

2 Formalizing SCT

2.1 Kleene Algebras

Since the core of SCT checking is the computation of a transitive closure, we will start by defining an axiomatic type class [20] of Kleene algebras, which provide the most general structure for such an operation. With this approach, the formulation of the algorithm is kept seperate from the concrete data structures.

Following the axiomatization by Kozen [12], Kleene algebras are idempotent semirings with an order defined as $(a \le b) = (a + b = b)$. Additionally, they include a star-operation satisfying the following four laws:

$$\begin{array}{ll} \mathit{1} + a{\cdot}a^* \le a^* & \qquad a{\cdot}x \le x \Longrightarrow a^*{\cdot}x \le x \\ \mathit{1} + a^*{\cdot}a \le a^* & \qquad x{\cdot}a \le x \Longrightarrow x{\cdot}a^* \le x \end{array}$$

These axioms follow from a stronger property, called *-continuity:

$$a{\cdot}b^*{\cdot}c = (\mathit{SUP}\ n.\ a{\cdot}b^n{\cdot}c)$$

where b^n denotes iterated multiplication. We define transitive closures as $a^+ = a^*{\cdot}a$.

In §4, we will give the transitive closure algorithm in terms of arbitrary Kleene algebras, which allows us to reason in a very abstract way, using simple algebraic laws. Since our graphs will be special Kleene algebras, the corresponding theorems simply follow as instances.

2.2 Graphs

We represent directed edge-labeled graphs as sets of triples. Graphs may have self-edges, and between two nodes there may be several edges:

datatype (α, β) *graph* = *Graph* $((\alpha \times \beta \times \alpha)$ *set*$)$

Instead of using the set type directly, we wrap graphs into their own type constructor. This will allow us to use axiomatic type classes to overload common notation for graph composition (written as multiplication), exponentiation and transitive closure. We write $x \xrightarrow[G]{e} y$ if G has an edge between nodes x and y, which is labeled with e. If we do not care about the label, we just write $x \xrightarrow[G]{} y$.

If the type of the edges has a multiplication and unit operation, these can be lifted to graphs, preserving monoid structure:

$$p \xrightarrow[G \cdot H]{b} q \quad = \quad \exists\, k\ e\ e'.\ p \xrightarrow[G]{e} k \xrightarrow[H]{e'} q \wedge b = e \cdot e'$$

$$p \xrightarrow[1]{b} q \quad = \quad p = q \wedge b = \mathit{1}$$

With addition defined as set union, we get a semiring structure with additive and multiplicative identity. Moreover, by taking the corresponding set operations for supremum and infimum, graphs form a complete lattice and we can define the star operation as $G^* = (\mathit{SUP}\ n.\ G^n)$. It is then not hard to show that graphs form a (*-continuous) Kleene algebra.

2.3 Paths

We represent infinite paths as sequences of node-edge-pairs:

types α *sequence* $=$ *nat* $\Rightarrow \alpha$
(α, β) *ipath* $= (\alpha \times \beta)$ *sequence*

The paths of a graph G are characterized by the predicate *has-ipath*:

$$\mathit{has\text{-}ipath} :: (\alpha, \beta)\ \mathit{graph} \Rightarrow (\alpha, \beta)\ \mathit{ipath} \Rightarrow \mathit{bool}$$

$$\mathit{has\text{-}ipath}\ G\ p = (\forall\, i.\ \mathit{fst}\ (p\ i) \xrightarrow[G]{p_{[i]}} \mathit{fst}\ (p\ (i + \mathit{1})))$$

Here, $p_{[i]}$ just abbreviates *snd* $(p\ i)$, yielding the value of the i-th edge in p.

For the proofs for size-change termination we also need to talk about finite paths and relate them to infinite paths (by taking sub-paths, constructing infinite paths from finite loops). We omit these details for space reasons, as they are essentially straightforward.

2.4 Size-Change Graphs

Size-change graphs have ↓ and ↧ as edge labels, and natural numbers as nodes, representing data positions. Control graphs have size-change graphs as their edges.

datatype *sedge* = *LESS* (↓) | *LEQ* (↧)
types
  scg = (*nat*, *sedge*) *graph*
  acg = (*nat*, *scg*) *graph*

Given an infinite path in the control graph, a thread is a sequence of natural numbers denoting argument positions for every node in the path, such that there are corresponding connected edges. A thread is descending, if it contains infinitely many ↓-edges:

$$\textit{is-desc-thread} \quad :: \quad \textit{nat sequence} \Rightarrow (\textit{nat},\ \textit{scg})\ \textit{ipath} \Rightarrow \textit{bool}$$

$$\textit{is-desc-thread}\ \theta\ p = ((\exists\, n.\ \forall\, i{\geq}n.\ \theta\ i \xrightarrow[p_{[i]}]{} \theta\ (i + \mathit{1})) \wedge (\exists_{\infty} i.\ \theta\ i \xrightarrow[p_{[i]}]{\downarrow} \theta\ (i + \mathit{1})))$$

Note that threads may also start at a later point in the path. Now the size-change property is defined as

$$\mathit{SCT}\ \mathcal{A} = (\forall\, p.\ \textit{has-ipath}\ \mathcal{A}\ p \longrightarrow (\exists\, \theta.\ \textit{is-desc-thread}\ \theta\ p))$$

The second characterization, which will be proved equivalent, is the basis of the size-change algorithm:

$$\mathit{SCT}_{\mathsf{ex}}\ \mathcal{A} = (\forall\, n\ G.\ n \xrightarrow[\mathcal{A}^+]{G} n \wedge G{\cdot}G = G \longrightarrow (\exists\, p.\ p \xrightarrow[G]{\downarrow} p))$$

Then the following is our main equivalence result, which corresponds to [14, Thm. 4]:

Theorem 1. $\textit{finite-acg}\ \mathcal{A} \Longrightarrow \mathit{SCT}\ \mathcal{A} = \mathit{SCT}_{\mathsf{ex}}\ \mathcal{A}$

The condition *finite-acg* $\mathcal{A}$ expresses that the control graph and all its size-change graphs are finite. In the original development it is implicit.

The formal proof of Thm. 1[1] consists of about 1200 lines of proof script in the Isar structured proof language, mainly following the informal development in [14], but with many parts spelled out in much more detail. Like in the informal version, the proof uses Ramsey's Theorem, which is already present in Isabelle's Library (the formalization is due to Paulson).

Our proof uses classical logic, including the (infinite, but countable) axiom of choice. It would be interesting to investigate if the proof can be modified to work in a weaker framework.

[1] The proof can be found in recent versions of the Isabelle library

3 Generating Size-change Problems

We will now apply the size-change principle to termination problems of a specific form, namely the termination of recursive function definitions in Isabelle itself. For that, we must make the abstract notion of control points and data positions concrete, and give meaning to the size-change graphs and control graphs.

There are multiple possibilities for doing this. In the spirit of the original authors, we could equate control points with functions and transitions with function calls. Instead we take the same route as Manolios and Vroon [15]: We take the *calls* as control points. A transition is a pair of calls, where one call is reachable from the other. This approach allows to analyze the recursive behaviour at a finer granularity.

As a running example, consider the following function definition[2]:

$$\begin{array}{lcl} f\ (n,\ 0) & = & n \\ f\ (0,\ Suc\ m) & = & f\ (Suc\ m,\ Suc\ m) \\ f\ (Suc\ n,\ Suc\ m) & = & f\ (m,\ n) \end{array}$$

When the definition is made, Isabelle will internally define the recursion relation (or call relation) R_f of the function. In the recursion relation, arguments of recursive calls are "smaller" than the corresponding left hand sides. In this case, R_f is defined as:

$$\begin{array}{l} R_f = \\ (\lambda x_1\ x_2. \\ \quad (\exists\, m.\ x_1 = (Suc\ m,\ Suc\ m) \wedge x_2 = (0,\ Suc\ m)) \vee \\ \quad (\exists\, n\ m.\ x_1 = (m,\ n) \wedge x_2 = (Suc\ n,\ Suc\ m))) \end{array}$$

It is our goal to show that R_f is well-founded.

3.1 Call descriptors

Recursion relations generated from function definitions are always disjunctions of existential clauses, each corresponding to a recursive call. By providing explicit descriptions for such call relations, we will make this structure accessible to the logic.

A *call descriptor* is a triple $(\Gamma,\ r,\ l)$, which describes a recursive call in a function definition: r is the argument of the recursive call, l is the original argument (from the left hand side of the equation) and Γ is the condition under which the call occurs. All three values depend on variables (the pattern variables), which we replace by a single variable (possibly containing a tuple).

types
$(\alpha,\ \gamma)\ cdesc \quad = \quad (\gamma \Rightarrow bool) \times (\gamma \Rightarrow \alpha) \times (\gamma \Rightarrow \alpha)$

Here, α is the argument type of the function and γ is the type of the pattern variable.

[2] The function does not compute anything useful.

A list of call descriptors describes a relation in the obvious way:

$$\mathit{in\text{-}cdesc} \;::\; (\alpha,\gamma)\ \mathit{cdesc} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$$
$$\mathit{in\text{-}cdesc}\ (\Gamma,\ r,\ l)\ x\ y = (\exists\, q.\ x = r\ q \wedge y = l\ q \wedge \Gamma\ q)$$

$$\mathit{mk\text{-}rel} \;::\; (\alpha,\gamma)\ \mathit{cdesc\ list} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$$
$$\mathit{mk\text{-}rel}\ []\ x\ y \qquad = \mathit{False}$$
$$\mathit{mk\text{-}rel}\ (c\ \#\ cs)\ x\ y = \mathit{in\text{-}cdesc}\ c\ x\ y \vee \mathit{mk\text{-}rel}\ cs\ x\ y$$

We can now describe R_f by such a list of call descriptors:

$$R_f =$$
$$\mathit{mk\text{-}rel}$$
$$[(\lambda(m,\ n).\ \mathit{True},\ \lambda(m,\ n).\ (\mathit{Suc}\ m,\ \mathit{Suc}\ m),\ \lambda(m,\ n).\ (\mathit{0},\ \mathit{Suc}\ m)),$$
$$(\lambda(m,\ n).\ \mathit{True},\ \lambda(m,\ n).\ (m,\ n),\ \lambda(m,\ n).\ (\mathit{Suc}\ n,\ \mathit{Suc}\ m))]$$

Transforming the definition to this form is easily automated by a suitable tactic. The main task here is to determine the type of γ which must be a product large enough to express all the variables in the different clauses. The equivalence to the original version simply follows by unfolding the definitions of *mk-rel* and *in-cdesc*.

3.2 Measure functions

To each call (i.e. each control point in the graph), we will assign a list of measure functions, which correspond to the data positions.

Measure functions capture the notion of size. The whole analysis is independent from the exact form of the measure functions. and any function mapping into a wellfounded domain can be used. For simplicity, our measure functions map into the natural numbers:

types
$\alpha\ \mathit{measure} \quad = \quad \alpha \Rightarrow \mathit{nat}$

Choosing measure functions is a separate problem not addressed here. As a default choice we use just the structural size functions which Isabelle provides for each inductive data type. Product types are split into their components. So for example for an argument type $S \times T$ we use the projections $\mathit{size}_S \circ \mathit{fst}$ and $\mathit{size}_T \circ \mathit{snd}$ as measure functions. Manolios and Vroon [15] describe some other heuristics for choosing measure functions.

3.3 Approximating the control graph

We will now show how to construct a size-change problem that corresponds to a relation given by a list of call descriptors.

For two call descriptors C_i and C_j, the predicate *no-step* is true if a C_i-call can never be followed by a C_j-call:

$$\mathit{no\text{-}step} \;::\; (\alpha,\gamma)\ \mathit{cdesc} \Rightarrow (\alpha,\gamma)\ \mathit{cdesc} \Rightarrow \mathit{bool}$$

$$\begin{aligned}&\mathit{no\text{-}step}\ (\mathit{\Gamma}_1,\ r_1,\ l_1)\ (\mathit{\Gamma}_2,\ r_2,\ l_2) =\\ &(\forall\, q_1\ q_2.\ \mathit{\Gamma}_1\ q_1 \wedge \mathit{\Gamma}_2\ q_2 \wedge r_1\ q_1 = l_2\ q_2 \longrightarrow \mathit{False})\end{aligned}$$

If we can prove $\mathit{no\text{-}step}\ C_i\ C_j$, then we can be sure that these calls can never occur in sequence. Otherwise we must add an edge between i and j to our control graph. This edge will carry a size change graph which approximates the size change behaviour of the call.

The predicates $\mathit{step}_<$ and $\mathit{step}_\leq$ capture strict and non-strict decrease of measures from one call to the next:

$$\begin{aligned}&\mathit{step}_< \;::\; (\alpha,\gamma)\ \mathit{cdesc} \Rightarrow (\alpha,\gamma)\ \mathit{cdesc} \Rightarrow \alpha\ \mathit{measure} \Rightarrow \alpha\ \mathit{measure} \Rightarrow \mathit{bool}\\ &\mathit{step}_\leq \;::\; (\alpha,\gamma)\ \mathit{cdesc} \Rightarrow (\alpha,\gamma)\ \mathit{cdesc} \Rightarrow \alpha\ \mathit{measure} \Rightarrow \alpha\ \mathit{measure} \Rightarrow \mathit{bool}\end{aligned}$$

$$\begin{aligned}&\mathit{step}_<\ (\mathit{\Gamma}_1,\ r_1,\ l_1)\ (\mathit{\Gamma}_2,\ r_2,\ l_2)\ m_1\ m_2 =\\ &(\forall\, q_1\ q_2.\ \mathit{\Gamma}_1\ q_1 \wedge \mathit{\Gamma}_2\ q_2 \wedge r_1\ q_1 = l_2\ q_2 \longrightarrow m_2\ (l_2\ q_2) < m_1\ (l_1\ q_1))\end{aligned}$$

$$\begin{aligned}&\mathit{step}_\leq\ (\mathit{\Gamma}_1,\ r_1,\ l_1)\ (\mathit{\Gamma}_2,\ r_2,\ l_2)\ m_1\ m_2 =\\ &(\forall\, q_1\ q_2.\ \mathit{\Gamma}_1\ q_1 \wedge \mathit{\Gamma}_2\ q_2 \wedge r_1\ q_1 = l_2\ q_2 \longrightarrow m_2\ (l_2\ q_2) \leq m_1\ (l_1\ q_1))\end{aligned}$$

Now consider a size-change graph G and functions M_1 and M_2 which assign measures to the data positions of C_1 and C_2. We say that G approximates the pair of calls, if the claimed inequalities are actually satisfied by the respective measures. This is expressed by the *approx* predicate:

$$\begin{aligned}\mathit{approx} \;::\; &\mathit{scg} \Rightarrow (\alpha,\gamma)\ \mathit{cdesc} \Rightarrow (\alpha,\gamma)\ \mathit{cdesc}\\ &\Rightarrow (\mathit{nat} \Rightarrow \alpha\ \mathit{measure}) \Rightarrow (\mathit{nat} \Rightarrow \alpha\ \mathit{measure}) \Rightarrow \mathit{bool}\end{aligned}$$

$$\begin{aligned}&\mathit{approx}\ G\ C_1\ C_2\ M_1\ M_2 =\\ &(\forall\, i\ j.\ (i \xrightarrow[G]{\downarrow} j \longrightarrow \mathit{step}_<\ C_1\ C_2\ (M_1\ i)\ (M_2\ j)) \wedge\\ &\qquad (i \xrightarrow[G]{\overset{=}{\downarrow}} j \longrightarrow \mathit{step}_\leq\ C_1\ C_2\ (M_1\ i)\ (M_2\ j)))\end{aligned}$$

Now, a control graph $\mathcal{A}$ is a sound description of a given list of call descriptors and measure functions, if between any two calls, either no step is possible, or $\mathcal{A}$ contains the corresponding edge with a size-change graph approximating the call combination[3]:

$$\mathit{sound\text{-}desc} \;::\; \mathit{acg} \Rightarrow (\alpha,\gamma)\ \mathit{cdesc\ list} \Rightarrow (\mathit{nat} \Rightarrow \alpha\ \mathit{measure})\ \mathit{list} \Rightarrow \mathit{bool}$$

$$\begin{aligned}&\mathit{sound\text{-}desc}\ \mathcal{A}\ D\ M =\\ &(\forall\, n{<}|D|.\ \forall\, m{<}|D|.\ \mathit{no\text{-}step}\ D_{[n]}\ D_{[m]} \vee (\exists\, G.\ n \xrightarrow[\mathcal{A}]{G} m \wedge \mathit{approx}\ G\ D_{[n]}\ D_{[m]}\ M_{[n]}\\ &M_{[m]}))\end{aligned}$$

Now, it is straigtforward to prove the following:

Theorem 2. If $\mathit{sound\text{-}desc}\ \mathcal{A}\ D\ M$ and $\mathit{SCT}\ \mathcal{A}$ then $\mathit{mk\text{-}rel}\ D$is wellfounded.

With this theorem, which is basically a formal version of the results in [15], we are able to prove wellfoundedness of a relation, provided we can express it in terms of a list of call descriptors and find an $\mathcal{A}$ which satisfies SCT and is a sound estimation of the relation.

[3] Here, $\mathit{xs}_{[i]}$ denotes the i-th element of the list xs.

3.4 Building size-change problems

It is not hard to build a custom proof tactic to construct $\mathcal{A}$ and prove *sound-desc* $\mathcal{A}$ D M:

- For each pair of calls C_i and C_j, try to prove *no-step* C_i C_j.
- If this succeeds, no edge needs to be added to $\mathcal{A}$.
- If it fails, construct a size-change graph G, by proving as many of the $\mathit{step}_<$ and $\mathit{step}_\leq$ estimations as possible. For each successful proof, the corresponding edge can be added to the G.

For the *"try to prove ..."* steps in the above algorithm, we simply call Isabelle's `auto` tactic, which combines rewriting, classical reasoning and some arithmetic. Of course, other proof methods could easily be plugged in here.

The result for our example function is given in Fig. 2. The $1 \overset{\downarrow}{\rightarrow} 1$ arrow in G_2 is surprising at first: The automated prover discovered that when going from C_2 to C_1, the first argument must get smaller, since for C_1, the first argument must be zero, but before it was nonzero. Also note that there is no arrow from C_1 to itself, which is essential for the termination of the function.

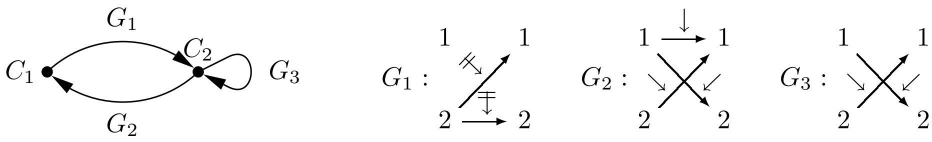

Fig. 2. The control graph and size change graphs for f

4 Implementation Prototype

Finally, an algorithm for checking the predicate SCT_{ex} (cf. §2.4) must be implemented and proved correct. We will present a naive implementation without any optimizations. While this does limit the performance of our system, it is sufficient to explain the ideas and demonstrate the overall approach.

We can use Isabelle's code generator to translate the algorithm into ML. The code generator (originally developed by Berghofer [3]) was recently redesigned by Haftmann [9] to generate code for definitions involving type classes. Type classes are compiled into dictionaries as it is done in Haskell compilers.

The code generation framework also supports the execution of functions involving (finite) sets, which are compiled to lists. By using this functionality, it takes just a few steps to produce a working prototype from our specification.

Recall that our definition of graph composition (cf. §2.2) involves existential quantification, which is of course undecidable in general. However, it is easy to make graph composition executable by proving the following equations and making them available to the code generator:

$$\begin{array}{l}
\mathit{joinable}\ ((n,\ e,\ m),\ (n',\ e',\ m')) = (m = n') \\
\mathit{connect}\ ((n,\ e,\ m),\ (n',\ e',\ m')) = (n,\ e{\cdot}e',\ m') \\
\mathit{Graph}\ G{\cdot}\mathit{Graph}\ H = \mathit{Graph}\ (\mathit{connect}\ \text{`}\ \{x \in G \times H.\ \mathit{joinable}\ x\})
\end{array}$$

Note that the bounded comprehension and the image operation (‘) are executable, as they are compiled to an expression involving *map* and *filter*.

The following function, overloaded on the type class of Kleene algebras, computes transitive closures by a simple iteration:

$$\mathit{mk\text{-}tcl}\ A\ X = (\textsf{if}\ X{\cdot}A \leq X\ \textsf{then}\ X\ \textsf{else}\ \mathit{mk\text{-}tcl}\ A\ (X + X{\cdot}A))$$

Note that *mk-tcl* need not always terminate. However, since the SCT problems we consider are always finite, termination can be proved for these cases.

By straigtforward induction, we can prove that *mk-tcl* computes transitive closures of finite graphs:

$$\mathit{finite\text{-}acg}\ A \Longrightarrow \mathit{mk\text{-}tcl}\ A\ A = A^{+}$$

Then the following function checks $SCT_{\textsf{ex}}$:

$$\begin{array}{l}
\mathit{test\text{-}SCT}\ \mathcal{A} = \\
(\textsf{let}\ \mathcal{T} = \mathit{mk\text{-}tcl}\ \mathcal{A}\ \mathcal{A} \\
\ \textsf{in}\ \forall (n,\ G,\ m) {\in} \mathit{dest\text{-}graph}\ \mathcal{T}. \\
\qquad n \neq m \vee G{\cdot}G \neq G \vee (\exists (p,\ e,\ q) {\in} \mathit{dest\text{-}graph}\ G.\ p = q \wedge e = \downarrow))
\end{array}$$

where $\mathit{dest\text{-}graph}\ (\mathit{Graph}\ G) = G$.

We prove that the function is correct:

Theorem 3. $\mathit{finite\text{-}acg}\ \mathcal{A} \Longrightarrow SCT_{\textsf{ex}}\ \mathcal{A} = \mathit{test\text{-}SCT}\ \mathcal{A}$

Note that the bounded universal and existential quantifiers in the definition of *test-SCT* do not prevent code generation: They are translated to the corresponding predicates on lists. Hence, *test-SCT* can be translated to ML and executed.

4.1 Putting everything together

Connecting the results of the previous sections, we obtain a method to formally certify the termination of functions in Isabelle:

- Define the function as usual, and create a list of call descriptors, representing the call relation.
- Assign suitable measures to each call, and, following the steps outlined in §3.4, construct a size-change problem $\mathcal{A}$.
- Apply Thm. 2. It remains to prove $SCT\ \mathcal{A}$.
- Apply Thm. 1. By construction, $\mathcal{A}$ is finite, so it remains to show $SCT_{\textsf{ex}}\ \mathcal{A}$.
- Apply Thm. 3, obtaining an executable goal.
- Evaluate the goal to *True*, either using the simplifier (which is currently only feasible for small examples), or by translating to ML first.

5 Examples

While our SCT implementation handles all forms of structural recursion and their lexicographic combinations, we are most interested in examples, where simpler analyses fail. The following example is adapted from [14]:

$p\ m\ n\ r = (\textsf{if}\ 0 < r\ \textsf{then}\ p\ m\ (r - 1)\ n\ \textsf{else if}\ 0 < n\ \textsf{then}\ p\ r\ (n - 1)\ m\ \textsf{else}\ m)$

Since the argument is permuted in each recursive call, simple size measures or lexicographic combinations are not sufficient to prove termination. The function from §3 is of a similar nature.

A different example shows the ability of the analysis to detect reachability between calls. The function has a boolean argument which eventually becomes *False*, and then the recursion descends on a different argument:

$foo\ True\ (Suc\ n)\ m = foo\ True\ n\ (Suc\ m)$
$foo\ True\ 0\ m = foo\ False\ 0\ m$
$foo\ False\ n\ (Suc\ m) = foo\ False\ (Suc\ n)\ m$
$foo\ False\ n\ 0 = n$

A third example is taken from the WST termination competition problem database [2]. The key observation here is that a recursive call can never occur more than once, which is again detected by the reachability analysis between calls, which yields a control graph with no edges.

$bar\ 0\ (Suc\ n)\ m = bar\ m\ m\ m$
$bar\ (Suc\ v)\ n\ m = 0$
$bar\ k\ 0\ m = 0$

These examples have a certain artificial flavour, as their only reason of existence seems to be to demonstrate termination proofs. So are there also practical examples where size change termination is useful?

The following example comes from a formalization of a descision procedure for equalities in a commutative ring, adapted from similar work in Coq [8] (the Isabelle version was done by Bernhard Häupler). The function adds two polynomials, represented by a datatype with three constructors Pc, Pinj and PX:

$add\ (\textsf{Pc}\ a)\ (\textsf{Pc}\ b) = \textsf{Pc}\ (a + b)$

$add\ (\textsf{Pc}\ c)\ (\textsf{Pinj}\ i\ P) = \textsf{Pinj}\ i\ (add\ P\ (\textsf{Pc}\ c))$

$add\ (\textsf{Pc}\ c)\ (\textsf{PX}\ P\ i\ Q) = \textsf{PX}\ P\ i\ (add\ Q\ (\textsf{Pc}\ c))$

$add\ (\textsf{Pinj}\ x\ P)\ (\textsf{Pinj}\ y\ Q) =$
$(\textsf{if}\ x = y\ \textsf{then}\ mkPinj\ x\ (add\ P\ Q)$
$\ \textsf{else if}\ y < x\ \textsf{then}\ mkPinj\ y\ (add\ (\textsf{Pinj}\ (x - y)\ P)\ Q)$
$\qquad \textsf{else}\ \underline{add\ (\textsf{Pinj}\ y\ Q)\ (\textsf{Pinj}\ x\ P)})$

$add\ (\textsf{Pinj}\ x\ P)\ (\textsf{PX}\ Q\ y\ R) =$
$(\textsf{if}\ x = 0\ \textsf{then}\ add\ P\ (\textsf{PX}\ Q\ y\ R)$
$\ \textsf{else if}\ x = 1\ \textsf{then}\ \textsf{PX}\ Q\ y\ (add\ P\ R)\ \textsf{else}\ \textsf{PX}\ Q\ y\ (add\ (\textsf{Pinj}\ (x - 1)\ P)\ R))$

$$\begin{array}{l}
\mathit{add}\ (\mathsf{PX}\ P_1\ x\ P_2)\ (\mathsf{PX}\ Q_1\ y\ Q_2) = \\
(\textsf{\textit{if}}\ x = y\ \textsf{\textit{then}}\ \mathit{mkPX}\ (\mathit{add}\ P_1\ Q_1)\ x\ (\mathit{add}\ P_2\ Q_2) \\
\ \textsf{\textit{else if}}\ y < x\ \textsf{\textit{then}}\ \mathit{mkPX}\ (\mathit{add}\ (\mathsf{PX}\ P_1\ (x - y)\ (\mathsf{Pc}\ \mathit{0}))\ Q_1)\ y\ (\mathit{add}\ P_2\ Q_2) \\
\qquad \textsf{\textit{else}}\ \underline{\mathit{add}\ (\mathsf{PX}\ Q_1\ y\ Q_2)\ (\mathsf{PX}\ P_1\ x\ P_2)})
\end{array}$$

$$\mathit{add}\ (\mathsf{Pinj}\ i\ P)\ (\mathsf{Pc}\ c) = \underline{\mathit{add}\ (\mathsf{Pc}\ c)\ (\mathsf{Pinj}\ i\ P)}$$

$$\mathit{add}\ (\mathsf{PX}\ P\ i\ Q)\ (\mathsf{Pc}\ c) = \underline{\mathit{add}\ (\mathsf{Pc}\ c)\ (\mathsf{PX}\ P\ i\ Q)}$$

$$\mathit{add}\ (\mathsf{PX}\ Q\ y\ R)\ (\mathsf{Pinj}\ x\ P) = \underline{\mathit{add}\ (\mathsf{Pinj}\ x\ P)\ (\mathsf{PX}\ Q\ y\ R)}$$

In the underlined cases the function just calls itself with permuted arguments. This avoids duplicating the code from other clauses – a sensible programming pattern for commutative functions. However, without an analysis dealing with argument permutation, it is extremely hard to convince Isabelle to accept this definition, which is why the function had to be rewritten in the original version, just for the sake of the termination proof, resulting in significant code duplication.

Note that such duplication does not only concern the function specification, but will turn up again in induction proofs about the function, as the induction rule is generated from the definition. This leads to redundant cases, whose "analogous" proofs have to be copy-and-pasted.

With our prototype, we could automatically prove termination of *add*.

6 Discussion

6.1 Scope of the method

It is hard to describe the class of problems that can be solved by our tool. While SCT itself is well-understood, the success of the overall method also depends on the quality of the estimations in the size change graphs, which is again determined by the capabilities of Isabelle's `auto` tactic.

While this makes it hard to predict if the method will succeed on a given problem, the advantage is that `auto` can make use of lemmas already present in the current theory. Most static analyses would have a hard time speculating and proving these lemmas, especially when induction is required. In Isabelle, such lemmas can be provided by the user and then used by automated tools.

This shows how interactive theorem proving and automated methods can benefit from each other, when combined: The user can help establishing difficult lemmas and SCT, with its strengths in combinatorics, provides the automated path analysis.

6.2 Practical applications

Manolios and Vroon tested their system against a large corpus of ACL2 definitions, and observed an impressive gain in automation.

Interestingly, when looking at the function definitions in the current Isabelle distribution and the Archive of Formal Proofs [1], most of the definitions can already be handled by a much simpler search for lexicographic orderings [6]. SCT does solve all these problems, but its real strengths are not used.

A possible explanation could be that users, knowing about Isabelle's limitations in that area, tried to avoid function definitions that would require a difficult manual termination proof, and used other modeling techniques like inductive relations instead. It remains to see whether this changes when SCT becomes generally available in Isabelle. But the *add* function discussed above already shows the potential of SCT.

6.3 Efficiency

Especially in the light of the PSPACE-hardness result, efficiency is a concern. In our setup, there are two critical operations:

First, in order to approximate the size-change problem, many proof goals must be generated and tried by the automated prover, one for each possible edge in each size-change graph. For the *add* function this takes about 2 minutes on a 1GHz laptop. As an improvement, one can implement a more efficient tactic, which is specialized on the kind of inequalities that actually occur, or add a heuristic to filter out subgoals that are likely to be unprovable.

Second, computing the transitive closure can take long. However, our minimalistic implementation represents graphs very inefficiently using sets (implemented by lists). First experiments showed that a better representation (e.g. matrices, implemented by quadtrees) leads to a significant speedup and we plan to integrate such an algorithm soon. Note that the proofs of the metatheory (§2) need not be changed.

7 Conclusion

By formalizing the size-change principle, we made an important termination criterion available for Isabelle. The implemented algorithm is only a proof-of-concept, but it should not be difficult to develop and integrate a more efficient implementation.

Recursive functions are an important application, but they are not the only one: It would be interesting work to apply SCT to other sorts of termination problems. One example is the framework for hoare-logic style verification of imperative programs [19], where termination proofs for loops and recursive procedures currently also need a user-specified well-founded relation. In fact, due to the modularity of SCT, much of the present work should be reusable with little or no change.

Acknowledgements. I want to thank Tobias Nipkow for fruitful discussions and helpful comments on a previous draft of this paper, and Amine Chaieb for pointing me to the *add* example. The anonymous referees provided valuable feedback.

References

1. Archive of Formal Proofs. http://afp.sourceforge.net/.
2. Termination Competition. http://www.lri.fr/~marche/termination-competition/.
3. S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *TYPES*, LNCS 2277, pages 24–40. Springer, 2000.
4. Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Texts in theoretical comp. science. Springer, 2004.
5. F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In A. Geser and H. Søndergaard, editors, *Eighth International Workshop on Termination, WST'06, Seattle, WA, USA*, 2006.
6. L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL, 2007. Manuscript.
7. M. Gordon and T. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.
8. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In J. Hurd and T. F. Melham, editors, *TPHOLs*, LNCS 3603, pages 98–113. Springer, 2005.
9. F. Haftmann and T. Nipkow. A design for a generic code generator for the Isabelle/HOL system, 2007. Manuscript.
10. J. Harrison. The HOL Light theorem prover. `http://www.cl.cam.ac.uk/users/jrh/hol-light`.
11. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.
12. D. Kozen. On Kleene algebras and closed semirings. In Rovan, editor, *Proc. Math. Found. Comput. Sci.*, LNCS 452, pages 26–47, Banska-Bystrica, Slovakia, 1990. Springer.
13. A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, LNAI 4130, pages 589–603. Springer, 2006.
14. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Prog. Languages*, pages 81–92, 2001.
15. P. Manolios and D. Vroon. Termination analysis with calling context graphs. In T. Ball and R. B. Jones, editors, *CAV*, LNCS 4144, pages 401–414. Springer, 2006.
16. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002.
17. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *CADE*, Springer LNCS 607, pages 748–752, 1992.
18. C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, New York, 1994.
19. N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *LPAR*, LNCS 3452, pages 398–414. Springer, 2004.
20. M. Wenzel. Using axiomatic type classes in Isabelle, 2000. Isabelle documentation.