Improvements on the "Size Change Termination Principle" in a functional language

Pierre Hyvernat, Christophe Raffalli
Université de Savoie, 73376 Le Bourget-du-Lac Cedex, France

Abstract

We present small improvements of Lee, Jones and Ben-Amram *size change termination principle* specialised to functional programming languages, where destructors come from pattern-matching. This allows for a finer analysis of call-graphs and yields a more precise test for termination. [1]

Introduction

Lee, Jones and Ben-Amram size change termination principle [1] is a simple yet surprisingly strong termination checker for generic programming languages. The ingredients are simply a well-founded order on values, a static analysis of the program to get a "call graph" of the recursive functions and a simple "transitive closure" computation on this graph.

The static analysis should yield a safe description of the potentials calls among the recursive function. The improvements described below were implemented on top of the PML programming language [2, 3], which due to its constraint checking algorithm is able to output a very detailed call-graph even when we use all the power of functional languages to try to hide the calls.

1 The size change principle

The central ingredients for the size-change termination principle are quite simple. Let G be a directed graph on vertices $F = \{f_1, f_2, ..., f_n\}$, with arcs labelled by elements of a *finite* monoid $(P, \circ)$. Let G^+ be the "transitive closure" of G in the following sense:

- G^+ has the same vertices as G,
- for all non-empty directed path $e_1, \dots, e_n$ from f to g in G, there is an edge from f to g in G^+ with label $l_1 \circ \cdots \circ l_n$, l_i being the label of e_i.

Note that because we impose that P is finite, G^+ is also finite.

Suppose now that G is the "call graph" of a set of mutually recursive definitions: vertices are function names and an edge from f to g describes a call to g in the definition of f. The elements of P can be used to describe the relationship between the *parameters* of the calling function f (variables) and the *arguments* of the called function g (terms). An element of P could, for example, tell us that "the second argument of g contains at least two constructors less than the third parameter of f".

With the appropriate monoid operation, the graph G^+ then describes the relation between arguments and parameters in iterated calls. If we have a well-founded order on terms, we have:

Theorem 1. *The following are equivalent [1]:*

- *all infinite directed path $e_1, \dots, e_n, \dots$ in G contain an infinitely decreasing thread,*[2]
- *G^+ contains an idempotent loop with label l (i.e. $l = l \circ l$) meaning "parameter i of f (as the called function) is smaller than argument i of f (as the calling function)", for some i.*

[1] Both authors want to thank Andreas Abel for the fruitful discussions that took place during his visit to Chambéry.

[2] where a thread is simply the trace of a parameter in the sequence of calls

It is interesting to note that this really is an equivalence. The proof is rather simple and uses a simple form of Ramsey's theorem. What is crucial is the finiteness of the monoid describing size relations between parameters and arguments.

The magic of this theorem is that while the first condition may seem uncomputable, the second is rather easy to check.[3] If our programming language only allows non termination from infinite calls to recursive functions, we can thus deduce from the second condition that our recursive definitions are all terminating. Fortunately, PML is such a programming language (see [3]).

2 Detecting impossible cases

The original monoid used by Lee, Jones and Ben-Amram was very simple. It consisted in bipartite graphs that could also be seen as matrices where the coefficient i, j described the relationship between argument j of g and parameter i of f:

- "`<`" meaning "is strictly smaller",
- "`=`" meaning "is smaller (possibly of the same size)",
- "`?`" meaning "I don't know, is possibly much bigger".

Composition was then some kind of matrix multiplication with the obvious composition of coefficients.

While this already achieved quite a lot, this is not really sufficient in a functional programming language where analysis of pattern-matching can give us a lot of information. Consider the following, rather ad-hoc function:

```
val rec f x = match x with
    A[x] -> f B[x]
  | B[x] -> C[]
```

There is a single recursive call, but because of the variant constructors, this call cannot be composed with itself. This function would not pass the original test of [1].

We add information to our monoid in the following way: for a specific call to g from f we consider a list of trees describing the relations between arguments of g and parameters of f. Each tree will describe how an argument of g is built (using variants or tuples) from the different *parts* of the parameters of f. Each such *part* is obtained from a parameter of f by destructing it (projection or pattern-matching).

Definition 1 (full call-tree and full call-information)**.** *We define two sets $\mathscr{L}$ and $\mathscr{T}$ inductively. The set $\mathscr{L}$ corresponds to a sequence of destructors applied to a parameter of the calling function, or a term of unknown size (call to an external function for example):*

- $\texttt{?} \in \mathscr{L}$ *("no information"),*
- $(i) \in \mathscr{L}$ *if* $i \in \mathbf{N}$ *(corresponding to the use of the i-th parameter),*
- $\texttt{-C}w \in \mathscr{L}$ *if* $w \in \mathscr{L}$ *(destruction of a variant: branch of a pattern-matching),*
- $\pi_i w \in \mathscr{L}$ *if* $w \in \mathscr{L}$ *(projection).*

The set $\mathscr{T}$ contains trees of constructors, with leafs containing elements of $\mathscr{L}$:

- $\mathscr{L} \subset \mathscr{T}$ *(leafs of the tree),*
- $\texttt{+C}t \in \mathscr{T}$ *if* $t \in \mathscr{T}$ *(construction of a variant),*
- $(t_1, \ldots, t_n) \in \mathscr{T}$ *if* $\forall 1 \leq i \leq n, t_i \in \mathscr{T}$ *(construction of a tuple).*

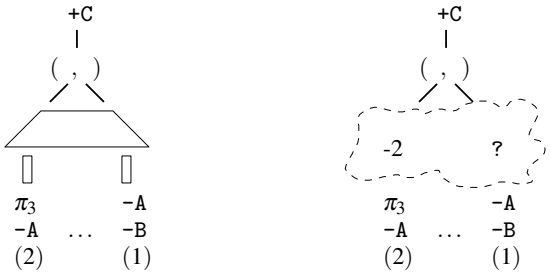

Figure 1: full tree and collapsed tree

A full call-information is simply given by a full call-tree for each argument of the called function.

Composition of full call-information is achieved by pasting appropriate trees on the leafs (*i.e.* pasting the i-th tree on parameter "(i)") and "normalising" by matching consecutive destructors / constructors pairs. Whenever a variant destructor "`-A`" meets with a variant constructor "`+B`" we can abandon the composition: this is an impossible case. Typing should moreover ensure that a projection (record destructor) can never meet a variant constructor and vice and versa.

Composition could make the tree become larger and larger. To ensure finiteness, we need to collapse the trees to have a given *depth*. This should bound both the depth of the constructor part and the length of all the destructor parts. The missing information is summarised by a *size*: "`<`", "`=`" or "`?`" as in the original principle. We actually use a segment of the integers, centered around 0. The original test is thus recovered using a depth of 0 and a size of 1. Of course, the two bounds (depth and size) can be increased to keep more information, at the expense of time and space consumption during the algorithm.

Definition 2 (bounded call-tree and call-information)**.** *The set $\mathscr{T}_{d,s}$ contains bounded trees of $\mathscr{T}$ with a middle "fuzzy" section. See the figure above for a graphical representation of such a tree. More formally, we replace the clause "(i)" by the following:*

- *if $-s \leq n \leq s$ then $(n,i) \in \mathscr{L}_{d,s}$.*

Moreover, if t is an element of this set:

- *the constructor tree of t is of depth at most d,*
- *the destructor sequences of t are of lenght at most d.*

We usually write `?` for the size s, which represents an unbounded increase in size greater that s.

An element of $\mathscr{T}_{d,s}$ is obtained by collapsing the middle section of an element of $\mathscr{T}$ to a "fuzzy section" by counting the deficit destructors / constructors (this is actually subtle if one wants to keep as much information as possible...).

Composition of bounded call-information cannot be obtained by first composing in an "unbounded" manner and then collapsing, because the middle section contains unknown destructors / constructors. We thus need to match destructors of the first call-information with constructors of the second call-information. If constructors or destructors remain after this matching, they need to be incorporated in the middle section of the composition, except if we know the appropriate fuzzy section to be empty, in which case they can be incorporated to the constructor tree of the destructor sequences. To do this, we need an *empty* element distinct from 0, the former meaning no change and the later meaning a constant size.

[3]This problem is however P-space complete. It should be noted that in practise, this isn't a problem...

3 Details on the notion of size

This monoid also allows for a more precise principle. When trying to compare the size of argument i and parameter j for a given call, we have to look at the branches in the i-th tree that reach parameter j. We only need to look at the best of these branches to guarantee termination. This allows us to treat pairs as "morally" distinct arguments, even when they are hidden beneath a variant constructor.

For example, this is what allows the test to tag the following (rather strange) function as terminating. This function computes the sum of a list of integers by keeping an accumulator in the head of the list:

```
val rec sum = fun
    | []  -> Z[]                  | [n] -> n
    | n::Z[]::l -> sum n::l       | n::S[k]::l -> sum S[n]::k::l
```

Note that typing isn't used in the algorithm, and the test doesn't distinguish between "`Z[]`" / "`S[]`" and the list constructors "`Nil[]`" / "`Cons[]`". The single full tree associated with the second recursive call is:

$$\texttt{+Cons}\,(\texttt{+S}\ \texttt{-Cons}\,\pi_1\,(1)\ ,\ \texttt{+Cons}\,(\texttt{-Cons}\,\pi_2\,\texttt{-Cons}\,\pi_1\,\texttt{-S}\ (1)\ ,\ \texttt{-Cons}\,\pi_2\,\texttt{-Cons}\,\pi_2\ (1)))$$

The middle branch also has a deficit of one variant constructor and will thus accounts for termination...

Note that when computing the deficit of size, we only count variant constructors. Record constructors / destructors do not serve any purpose there...

4 Building the initial call graph

PML computes the call graph from a set of typing constraints extracted from the program (see [3] for a formal description of the process). Given an uncountable set of formal *type names* $\mathscr{N}$, we annotate each subterm of a program with distinct type names. The only exception is for variables: all occurrences of the same variable have the same annotation. We write type names as Greek letter and annotation as upper-script. The constraints $\mathscr{C}$ are then collected as follows:

- functions: if $(\mathtt{f}^{\alpha}\,\mathtt{u}^{\beta})^{\gamma}$ occurs in the program then $\alpha \subset (\beta \to \gamma) \in \mathscr{C}$ and if $(\mathtt{fun}\,x^{\beta} \to \mathtt{u}^{\gamma})^{\alpha}$ occurs then $(\beta \to \gamma) \subset \alpha \in \mathscr{C}$.
- tuples: if $\pi_i(\mathtt{u}^{\alpha})^{\beta}$ occurs then $\alpha \subset \pi_i\beta \in \mathscr{C}$ and if $(t_1^{\alpha_1},\ldots,t_n^{\alpha_n})^{\beta}$ occurs then $\alpha_1 \times \cdots \times \alpha_n \subset \beta \in \mathscr{C}$.
- variants: if $C[t^{\alpha}]^{\beta}$ occurs then $\mathtt{C}[\alpha] \subset \beta \in \mathscr{C}$ and if $(\mathtt{case}\,t^{\beta}\ \mathtt{of}\ \mathtt{C}_1[x^{\alpha_1}] \to u_1^{\gamma_1} \mid \cdots \mid \mathtt{C}_n[x^{\alpha_n}] \to u_n^{\gamma_n})^{\gamma}$ then $\{\beta \subset C_1[\alpha_1] + \cdots + C_n[\alpha_n], \gamma_1 \subset \gamma, \ldots, \gamma_n \subset \gamma\} \subset \mathscr{C}$.

Those formal constraints are a syntactical version of inclusions between semantical "types".

These extracted constraints are saturated by "deduction", which correspond to computing an approximation of the constraints obtained after reduction of the term. A well-foundedness test is then performed to ensure that loop can only occur in recursive definitions. This is described in [3].

Finally, by traversing the resulting constraints, we can construct the call graph associated to our program. There are basically four steps: collecting the parameters (the function abstraction), collecting the arguments to each calls (the function applications), building the destructor parts of the calls and finally the constructors parts.

This approach allows us to deal with tuples containing functions, tests before function abstractions and other non-standard ways of writing functions and calling them. Those are not frequent but still possible in functional languages, as in the following example which is treated as two separate mutually recursive functions by PML:

```
  val rec f x =
    (fun S[y] -> (f y).2 x x | Z[] -> Z[]),
    (fun y -> (fun S[z] -> (f z).1 y | Z[] -> Z[]))
```

5 Examples and further work

The first example shows what we can gain from the original size-change termination principle:

```
val rec f a b =
  match a,b with
      Node[ Node[a1 , _] , _]   ,  b -> f Node[a1,b] Node[a,b]
    | _,_ -> A[]
```

What is new in this example is that even thought both arguments may increase in global size, the left branch of the first argument decreases. If the depth bound is 2, the corresponding tree will be: $\texttt{+Node}\,(\texttt{-Node}\,\pi_1\,(-1,1),(0,2))$, where the "$-1$" will account for the decreasing branch...

The second example illustrates how the extraction from the typing constraints allow to detect termination when using external function calls in recursive calls. This example is an implementation of subtraction and modulo on unary natural numbers:

```
val pred x = match x with
    S[x] -> x

val rec sub x y = match x, y with
  _, Z[] -> x
| Z[], _ -> raise Undef[]
| _ -> sub (pred x) (pred y)
```

```
val rec mod_aux acc x y' =
  try   let x' = (sub (pred x) y') in
        mod_aux S[acc] x' y'
  with  Undef[] -> acc

val mod x y = match y with
    Z[] -> raise Undef[]
  | S[y'] -> mod_aux Z[] x y'
```

In this example, the call-information for subtraction is $\texttt{-S}(1)$ for argument 1 and $\texttt{-S}(2)$ for argument 2. This means that we detected that the function `pred` removes a `S` on each arguments. The call information for the $\texttt{mod}_\texttt{a}\texttt{ux}$ function is $\texttt{+S}(1)$, $\texttt{-S}(2)$ and (3), which means that we detected that the composition of `pred` and `sub` remove one successor. However, replacing `(sub (pred x) y')` by `sub x S[y']` would fails, because we do not detect that the subtraction will remove at least one successor. Work is being done in that direction...

References

[1] Chin Soon Lee, Neil D. Jones and Amir M. Ben-Amram, *The Size-Change Principle for Program Termination*, ACM SIGPLAN Notices, Volume 36, Issue 3, 2001.

[2] Christophe Raffalli, *The* PML *programming language*, homepage of the project: `http://www.lama.univ-savoie.fr/tracpml`.

[3] Christophe Raffalli, *Realizability for programming languages*, course notes for the "École jeunes chercheurs du GDR IM", 2010. (hal-00474043), `http://www.lama.univ-savoie.fr/~raffalli/pdfs/ejc.pdf`.