Certification of Haskell programs termination

Intern-ship directed by Frédéric Blanqui

Julien Bureaux

École Normale Supérieure (Paris)

July 17, 2009

Context of this work

- Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. `http://coq.inria.fr/`
- CoLoR is a Coq library on rewriting and termination with tactics for automatically checking the conditions of termination theorems. `http://color.inria.fr/`
- Haskell is an advanced purely functional programming language with non-strict semantics. `http://www.haskell.org/`

Introduction

At the moment CoLoR can only handle rewrite systems. The subject of my intern-ship is to extend it so that it will be able to certify termination proofs for Haskell programs by adapting techniques for TRS.

The main part of this work is to formalize and adapt the techniques described in the article *Automated Termination Analysis for Haskell : From Term rewriting to Programming Languages* written by J. Giesl, S. Swiderski, P. Schneider-Kamp, R. Thiemann in 2006. This article shows how termination techniques for ordinary rewriting can be used to handle the features of Haskell which are missing in term rewriting, especially by using the technique of dependency pairs.

Some differences

What makes adapting term rewriting termination techniques for Haskell not so easy ?

- Haskell has a lazy evaluation strategy
- Defining equations are used in the order they are written
- Haskell has polymorphic types
- In programs with infinite data objects, not all functions are terminating
- Haskell is a high-order language

Part I

Syntax and operational semantics of Haskell

Example of Haskell program

```
data Nats = Z | S Nats
data List a = Nil | Cons a (List a)

p (S Z) = Z
p (S x) = S (p x)

take Z xs = Nil
take n Nil = Nil
take (S n) (Cons x xs) = Cons x (take (p (S n)) xs)

from x = Cons x (from (S x))
```

For instance we could want to prove the termination of the term

```
take n (from x)
```

Some vocabulary

- the symbols **Nats, List** of arity 0, 1 are called type constructors,
- the set of types is the smallest set containing (type) variables, type constructors well-applied to types, and arrow-types $U \to V$ for types U, V.
- The functions symbols **p, take, from** of arity 1, 2, 1 are said defined
- whereas **Z, S, Nil, Cons** of arity 0, 1, 0, 2 are (data) constructor functions,
- the set of terms is the smallest set containing (term) variables, function symbols and *well-typed* application $(u\ v)$ for terms u, v,

Syntax

We consider a subset of Haskell with

- function declaration of the form $f\ l_1 \dots l_n = r$ where $l_1, \dots, l_n$ are patterns with no variables in common.
- no lambda-abstractions ($\backslash t_1 \dots t_n \rightarrow t$ with free variables $x_1, \dots, x_m$ can be replaced with $f\ x_1 \dots x_m$ where f is a new function symbol defined by $f\ x_1 \dots x_m\ t_1 \dots t_n = t$)
- no built-in types. Only user-defined data-structures are permitted, like

```
data Nats = Z | S Nats
data List a = Nil | Cons a (List a)
```

- no **let**, no **cases**, [...]

Evaluation position

Given a defining equation $l = r$ in the program, the evaluation position of t with respect to l $e_l(t)$ is defined, if it exists, as the first position in leftmost outermost order for which l and r are different and such that this position points to a constructor in l, and if the subterm at this position in t is either a defined function symbol or a variable.

Example : if `t = take u (from m)` and we consider the rule left-hand side `take (S n) (Cons x xs)` then we get $t|_{e_l(t)} = u$. For `s = take (S n) (from m)` we get $s|_{e_l(s)} = \mathit{from}\ m$.

If $e_l(t)$ is defined we say that the defining equation whose left-hand side is l is a feasible equation for t.

We now define the evaluation position for a term t by looking for feasible equations and the corresponding position recursively on the term.

1. $e(t) = 1^{m-n}\pi$ if $t = f\ t_1\ \dots\ t_n\ \dots\ t_m$, f is defined, $m > n = arity(f)$, $\pi = e(f\ t_1 \dots t_n)$
2. $e(t) = e_l(t)\pi$ if $t = f\ t_1\ \dots\ t_n$, f is defined, $n = arity(f)$, there are feasible equation for t and the first is $l = r$, $e_l(t) \neq \varepsilon$ and $\pi = e(t|_{e_l(t)})$
3. else $e(t) = \varepsilon$

Example : with the previous program declaration, if
`t = take u (from m)` and
`s = take (S n) (from (p (S m)))`, then $t|_{e(t)} = u$ and $s|_{e(t)} = m$.

Reduction

Then, the evaluation relation is just an interpretation of this specific evaluation strategy. $\rightarrow_H$ is defined by

- If t rewrites to s on position $e(t)$ using the first equation whose left-hand side matches $t|_{e(t)}$ then $t \rightarrow_H s$.
- If $t = c\ t_1\ \ldots\ t_n$ for a constructor c of arity n, $s = c\ t_1\ \ldots\ t_{i-1}\ s_i\ t_{i+1}\ \ldots\ t_n$, and $t_i \rightarrow Hs_i$ for some $1 \leq i \leq n$ then $t \rightarrow_H s$.

For example :

$$p\ (S\ Z) \rightarrow_H Z$$
$$take\ (S\ n)\ (from\ (p\ (S\ Z))) \rightarrow_H take\ (S\ n)\ (from\ Z)$$
$$from\ Z \rightarrow_H Cons\ Z\ (from\ (S\ Z)) \rightarrow_H \ldots$$

Termination

The set of H-terminating ground terms is the smallest set of ground terms t with

- t does not start an infinite sequence of reductions
- if $t \rightarrow_H^* f\ t_1\ \ldots\ t_n$ for a defined symbol f with $n < arity\ f$, and the term t' is H-terminating, then $f\ t_1\ \ldots\ t_n t'$ is also H-terminating
- if $t \rightarrow_H^* c\ t_1\ \ldots\ c_n$ for a constructor c, then $t_1, \ldots, t_n$ are also terminating.

A term t is H-Terminating iff $t\sigma$ is H-terminating for all substitutions σ with $H - terminating$ ground terms.

For example `from` is not H-terminating because `from Z` has an infinite evaluation. However `take u (from m)` is H-terminating since it is so when instantiating u and m with H-terminating ground terms.

Part II

Formalization in Coq

Terms and Types

We currify everything so that we can define the Haskell-signature of a program using the *ASignature* type of CoLoR :

For terms, we declare an applicative algebra with a nullary symbol for each function and a binary symbol **App** for function application

```
Inductive applicative_term : Type :=
  | Symb : S -> applicative_term
  | App : applicative_term.

Definition applicative_arity f :=
  match f with Symb _ => 0 | App => 2 end.
```

For types, an applicative algebra with a binary symbol **Arrow** and a symbol for each type constructor with the corresponding arity

```
Inductive type_symbol : Type :=
  | Constr : S -> type_symbol
  | Arrow : type_symbol.

Definition type_arity T :=
  match T with
    | Constr c -> arity c
    | Arrow -> 2
  end.
```

We also need a function **type_of** from function symbols to types.

Typing relation

An environment E is an association table mapping variables to types. This leads to the following rule

$$\frac{(x, T) \in E}{E \vdash x : T} \; (Var)$$

Denoting by τ_f the type associated to f by **type_of**, for every function symbol f and every type substitution φ we have the rule

$$\frac{}{E \vdash f : \tau_f \varphi} \; (Symb)$$

Finally we give a rule for function application.

$$\frac{E \vdash t : U \to V \quad E \vdash u : U}{E \vdash t \; u : V} \; (App)$$

Coq code for typing

This can be easily implemented in Coq as an inductive predicate :

```
Inductive typing : henv -> hterm -> htype -> Prop :=
  | Tvar : forall E x T,
    MapsTo x T E -> typing E (hvar x) T
  | Tsymb : forall E f phi,
    typing E (hsymb f) (sub phi (type_of_symbol f))
  | Tapp : forall E t U V u,
    typing E t (tarrow U V) -> typing E u U ->
    typing E (happ t u) V.
```

Evaluation position

In order to define the evaluation position of t with respect to l we first search for a candidate position in leftmost outermost order using a recursive function, and if it exists, we check if this position correspond to a constructor symbol. For such a search it is very practical to use the *option types* of Coq.

```
Definition pos_wrt t l :=
  match candidate_pos t l with
    | Some (ps, h) =>
      if hconstructor h then None else Some ps
    | None          => None
  end.
```

Then we define **fpos** the evaluation position corresponding to the first feasible equation in the program (option types are very useful here too).

These functions allow us to declare more concisely the evaluation position function **epos**.

```
Function epos {wf subterm t} : position :=
  match unhapps t with
    | cons u tm =>
      if negb (hdefined u) then nil else
        let n := htarity u in let m := length tm in
          match nat_compare n m with
            | Eq =>
              match fpos t R with
                | Some ps =>
                  match subterm_pos t ps with
                    | None => ps
                    | Some x => ps ++ epos x
                      ...
```

Note : we must here use *Function* instead of *Fixpoint* and specify the well-founded order to consider (here subterm). It introduces some new goals to prove in order to complete the definition.

Equivalent definition of Termination

In fact, the definition given by Giesl of H-terminating terms is not well adapted to formalization in Coq because it introduces odd occurrences of the inductive predicate in its own definition. We better deal with a more natural equivalent definition connected to the notion of reducibility : a ground term t of type T is said to be reducible if and only if it does not start an infinite sequence of reductions and either

- T is a base type (that is to say a well-applied type constructor), or
- T is an arrow type $U \to V$ and for all ground term u, if $u : U$ is reducible then so is $(t\ u) : V$.

This leads to the following declaration for ground terms :

```
Fixpoint gRed (T : htype) t : Prop :=
  SN hred t /\ typing henv_empty t T /\
  match T with
    | Fun f Ts =>
      match f,Ts with
        | Constr _, _ => True
        | Arrow, Vcons V _ (Vcons W _ _) =>
          forall v, gRed V v -> gRed W (happ t v)
        | _, _ => False
      end
    | _ => False
  end.
```

We get the notion of termination for all terms by considering substitutions with ground terms.

```
Definition Red E T t := typing E t T /\ forall s,
  wt_sub henv_empty E s -> gRed T (sub s t).
```

The technique of Giesl for proving that a term t is terminating start by expanding a finite graph from this term using certain rules that mime the previous definitions by adding new children :

- the Eval rule links a term to its reduced term,
- the Case rule destructs a variable at the evaluation position by creating a new child for each constructor of the right type replacing the variable
- the VarExp rule applies a new variable when a function is under-applied
- the ParSplit rule splits the parameters of a constructor
- the Ins rule look if the current term can be expressed as an instantiation of a previous one, and it that case links them together

Remark : if one disregards the last rule the graph created would be a tree.

Termination Graph

We start defining generic double-labelled trees and a few functions to deal with positions. The first label of a node is the value of the node, the second is its sort.

```
Inductive tree : Type :=
  Node : A -> B -> list tree -> tree.
```

Following Giesl the sorts of node that we consider are :

```
Inductive node_sort :=
  | EvalNode : node_sort
  | CaseNode : list substitution -> node_sort
  | VarExpNode : node_sort
  | ParSplitNode : node_sort
  | InsNode : position -> node_sort
  | Leaf : node_sort.

Definition graph :=
  tree (hterm * htype * henv) node_sort.
```

Expansion relation

We now define the relation that expresses the fact that a graph G can be expanded to another graph G' (denoted $G \Rightarrow G'$) as an inductive predicate :

```
Inductive expands : graph -> graph -> Prop :=
  | evalrule : forall g t t' ps f ts,
    leaf_pos g ps t ->
    happs f ts = t ->      (* t = f t1 t2 ... tn *)
    hdefined f = true ->
    length ts >= htarity f  ->
    hred t t' ->
    expands g (add_leaf (leaf t') EvalNode ps g)

  | varexprule : forall g ps t f ts,
    let lf := leaf (happ t (fvar t)) in
      leaf_pos g ps t ->
      happs f ts = t ->
      hdefined f = true ->
      length ts < htarity f ->
      expands g (add_leaf lf VarExpNode ps g)
```

```
  | parsplitrule : forall g ps t c ts,
    let add g t :=
      add_leaf (leaf t) ParSplitNode ps g in
    let g' := fold_left add ts g in
      leaf_pos g ps t ->
      happs c ts = t ->
      hconstructor c = true ->
      expands g g'

  | caserule : ...
  | insrule : ...
.
```

If $G_{t,T,E}$ denotes the graph containing as only node the leaf (t, T, E), G is a termination graph for t $(E \vdash t : T)$ if and only if $G_{t,T,E} \Rightarrow^{*} G$.

Conclusion and future work

At this point we have ended the formalization in Coq of Haskell programs, of the Haskell evaluation strategy and of the definition of termination. We also have started to go deeper in the analyse of termination by defining termination graphs and their expansion rules.

But there are still many things to do :

- prove that a node terminates if and only if all its children terminate
- connecting Termination Graphs with Dependency Pairs Problems