foetus - Termination Checker for Simple Functional Programs

Andreas Abel*

July 16, 1998

Abstract

We introduce a simple functional language foetus (lambda calculus with tuples, constructors and pattern matching) supplied with a termination checker. This checker tries to find a well-founded structural order on the parameters on the given function to prove termination. The components of the check algorithm are: function call extraction out of the program text, call graph completion and finding a lexical order for the function parameters. The HTML version of this paper contains many ready-to-run Web-based examples.

1 Introduction

Since the very beginning of informatics the problem of termination has been of special interest, for it is part of the problem of program verification for instance. Because the halting problem is undecidable, there is no method that can prove or disprove termination of all programs, but for several systems termination checkers have been developed. We have focused on functional programs and designed the simple language foetus[1], for which we have implemented a termination prover. foetus is a simplification of MuTTI (Munich Type Theory Implementation) based on partial Type Theory (ala Martin

*Theoretical Computer Science, Institute of Computer Science, Ludwigs-Maximilians-University, Oettingenstr. 67, D-80538 Munich, Germany, email: `abel@informatik.uni-muenchen.de`. I want to thank my supervisor Thorsten Altenkirch and Rolf Backofen for his friendly support in technical questions.

[1]In German foetus is an abbreviation of "Funktionale – Obgleich Eingeschränkt – Termination Untersuchende Sprache" ;-). It also expresses that it is derived from MuTTI (this is the German term for *Mum*).

Löf) extended by tuples, constructors and pattern matching. For the syntax see section 2.1.

To prove the termination of a functional program there has to be a well founded order on the product of the function parameters such that the arguments in each recursive call are smaller than the corresponding input regarding this order. We have limited to *structural* orderings. foetus tries to find such an order by collecting all recursive calls of the given function and the belonging behaviour of the function arguments. To handle mutually recursive functions a call graph is constructed and completed.

Section 2 introduces the foetus "body" (syntax and type system). Section 3 provides some examples to intuitively learn the language and see the interpreter and termination checker work. Then in section 4 we explain the "heart" of foetus: the call extractor; we also informally introduce call graph completion and finding of the lexical order: the "brain" of foetus. The latter is formally described in section 5.

2 foetus Language

2.1 foetus Syntax

A foetus program consists of *terms* and *definitions*.

P	::		*empty program*
	|	*term*; P	*term to be evaluated*
	|	*definition*; P	*definition for further use*

When processing input, foetus evaluates the terms and stores the definitions in the environment. "Reserved words" in the foetus language are `case`, `of`, `let` and `in`. Special characterss are `( ) [ ] { } | . , ; = =>`. An identifier may contain letters, digits, apostrophies and underscores. If it starts with a small letter it stands for a variable, else it denotes a constant.

Term syntax. In the following x, x_1, x_2, ... denote variables, C, C_1, C_2, ... constants and u, t, t_1, t_2, ... foetus terms.

t	::	x	*variable*
	$\mid$	$[x]t$	*lambda*
	$\mid$	tu	*application*
	$\mid$	$C(t)$	*constructor*
	$\mid$	case t of $\{\ C_1 x_1 \Rightarrow t_1 \mid \ldots \mid C_n x_n \Rightarrow t_n\ \}$	*pattern matching*
	$\mid$	$(C_1 = t_1, \ldots, C_n = t_n)$	*tuple*
	$\mid$	$t.C$	*projection*
	$\mid$	let $x_1 = t_1, \ldots, x_n = t_n$ in t	*let*
	$\mid$	(t)	*(extra parentheses)*

Definitions. A definition statement has the form $x_1 = t_1, \ldots, x_n = t_n$ (it is a let-term without a "body"). All variables $x_1, \ldots, x_n$ are defined simultanously, thus can refer to each other.

Example. The following foetus program defines addition on natural numbers (spanned by the two constructors `O` "zero" and `S` "successor") and calculates $1 + 1$.

```
add = [x][y]case x of
        { O z => y
        | S x' => S(add x' y) };
one = S(O());
add one one;
```

Note that although `O` is a zero-argument-constructor the syntax forces us to supply a dummy variable `z` within the pattern definition and also empty tuple `()` in the definition of `one`.

2.2 foetus Type System

In the following x, x_1, x_2, ... denote variables, C, C_1, C_2, ... constants, u, t, t_1, t_2, ... foetus terms, τ, σ, σ_1, σ_2, ... foetus types and X, X_1, X_2, ... type variables. $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ denotes the context. The judgement

$$\Gamma \vdash t : \sigma$$

means "in context Γ term t is of type σ".

Type formation.

τ	::	$\sigma \to \tau$	*$\to$-type*
	$\mid$	$\{C_1 : \sigma_1 \mid \ldots \mid C_n : \sigma_n\}$	*labeled sum type*
	$\mid$	$(C_1 : \sigma_1, \ldots, C_n : \sigma_n)$	*labeled product type*
	$\mid$	$\{X\}\tau$	*polymorphic type*
	$\mid$	$\tau\sigma$	*instantiation of polymorphic type*
	$\mid$	Let $X_1 = \sigma_1, \ldots, X_n = \sigma_n$ in τ	*recursive type*

In the formation of a recursive type with Let X_i may only appear strict positiv in σ_i. We define congruence on types $\cong$ as the smallest congruence closed under

$$\text{Let } \vec{X} = \vec{\sigma} \text{ in } \tau \cong \tau[X_1 := \text{Let } \vec{X} = \vec{\sigma} \text{ in } X_1; \ldots; X_n := \text{Let } \vec{X} = \vec{\sigma} \text{ in } X_n]$$

($\vec{X} = \vec{\sigma}$ abbreviates $X_1 = \sigma_1, \ldots, X_n = \sigma_n$). Thus we can substitute congruent types:

$$\frac{\Gamma \vdash t : \sigma \quad \sigma \cong \tau}{\Gamma \vdash t : \tau}$$

For ploymorphic types we have rules like in System F:

$$\frac{\Gamma \vdash t : \sigma \quad X \text{ not free type variable in } \Gamma}{\Gamma \vdash t : \{X\}\sigma}\,\mathrm{poly-i}$$

$$\frac{\Gamma \vdash t : \{X\}\sigma}{\Gamma \vdash t : \sigma[X := \tau]}\,\mathrm{poly-e}$$

2.3 Typing rules for foetus terms

We here only briefly introduce the typing rules. For more detailed explanation, read a book about type theorie, e.g. [NPS90].

Lambda abstraction.

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash [x]t : \sigma \to \tau} \to -\mathrm{i}$$

Application.

$$\frac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash tu : \sigma} \to -\mathrm{e}$$

Constructor.

$$\frac{\Gamma \vdash t : \sigma_i}{\Gamma \vdash C_i(t) : \{C_1 : \sigma_1 | \ldots | C_n : \sigma_n\}} \, \{\} - \mathrm{i}$$

Pattern matching.

$$\frac{\Gamma \vdash t : \{C_1 : \sigma_1 | \ldots | C_n : \sigma_n\} \qquad \Gamma, x_i : \sigma_i \vdash u_i : \sigma \text{ for all } 1 \leq i \leq n}{\Gamma \vdash \text{case } t \text{ of } \{C_1(x_1) \Rightarrow u_1 | \ldots | C_n(x_n) \Rightarrow u_n\} : \sigma} \, \{\} - \mathrm{e}$$

Tupels.

$$\frac{\Gamma \vdash t_i : \sigma_i \text{ for all } 1 \leq i \leq n}{\Gamma \vdash (C_1 = t_1, \ldots, C_n = t_n) : (C_1 : \sigma_1, \ldots, C_n : \sigma_n)} \, () - \mathrm{i}$$

Projection.

$$\frac{\Gamma \vdash t : (C_1 : \sigma_1, \ldots, C_n : \sigma_n)}{\Gamma \vdash t.C_i : \sigma_i} \, () - \mathrm{e}$$

Let.

$$\frac{\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash t_i : \sigma_i \text{ for all } 1 \leq i \leq n \qquad \Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash u : \tau}{\Gamma \vdash \text{let } x_1 = t_1; \ldots; x_n = t_n \text{ in } u : \tau} \, \text{let}$$

In foetus type checking is not yet implemented and it is assummed that all terms entered are well typed. Of course, only for well typed terms the termination check produces valid results.

Example. The following well-known example for non-termination passes the foetus termination checker, but it is not well typed.

```
f = [x]x x;
a = f f;
```

foetus output:

```
f passes termination check
a passes termination check
```

3 Examples

3.1 Addition and multiplication

On the natural numbers

$$\text{nat} := \text{Let nat} = \{\texttt{O}()|\texttt{S}(\text{nat})\} \text{ in nat}$$

we define $\texttt{add}, \texttt{mult} : \text{nat} \to \text{nat} \to \text{nat}$:

```
add = [x][y]case x of
        { O z => y
        | S x' => S(add x' y) };
mult = [x][y]case x of
        { O z => O z
        | S x' => (add y (mult x' y)) };
add (S(S(O()))) (S(O()));
mult (S(S(O()))) (S(S(S(O()))));
```

foetus **output:**

```
< =: add -> add
add passes termination check by lexical order 0
< =: mult -> mult
mult passes termination check by lexical order 0
result: S(S(S(O())))
result: S(S(S(S(S(S(O()))))))
```

3.2 Subtraction

We define the predecessor function $\texttt{p} : \text{nat} \to \text{nat}$ and substraction on natural numbers $\texttt{sub} : \text{nat} \to \text{nat} \to \text{nat}$. Note $\texttt{sub}\ x\ y$ calculates $y \dot{-} x$.

```
p = [x]case x of { O z => O z | S x' => x' };
sub = [x][y]case x of
        { O z => y
        | S x' => sub x' (p y) };
sub (S(S(O()))) (S(S(S(S(O())))));
```

foetus **output:**

```
p passes termination check
< ?: sub -> sub
sub passes termination check by lexical order 0
result: S(S(O()))
```

3.3 Division

Division `div` : nat $\to$ nat $\to$ nat can be implemented as follows in functional languages (note that `div x y` calculates $\lfloor \frac{y}{x} \rfloor$):

```
div (x,y) = div'(x,y+1-x)
div'(x,y) = if (y=0) then 0 else div'(x,y-x)
```

`div'` (just like division on natural numbers) terminates if the divisor `x` is unequal 0 because then `y-x` $<$ `y` in the recursive call and thus one function argument is decreasing. But foetus recognizes only direct structural decrease and cannot see that `sub x y'` is less than `y'`. To prove termination of `div'` you need a proof for $x \neq 0 \to \mathsf{sub}\, x\, y < y$ [BG96].

```
p = [x]case x of { O z => O z | S x' => x' };
sub = [x][y]case x of
        { O z => y
        | S x' => sub x' (p y) };
div = [x][y]let
        div' = [y']case y' of
                { O z => O z
                | S dummy => S(div' (sub x y')) }
        in
        (div' (sub (p x) y));
div (S(S(O()))) (S(S(S(S(S(O()))))));
```

foetus **output:**

```
p passes termination check
< ?: sub -> sub
sub passes termination check by lexical order 0
div passes termination check
?: div' -> div'
div' FAILS termination check
result: S(S(O()))
```

Here foetus says `div'` fails termination check, so `div` will not terminate either. `div` would terminate, if `div'` terminated, therefore you get the answer `div passes termination check`.

3.4 Ackermann function

The not primitive recursive Ackermann function `ack` : nat $\to$ nat $\to$ nat.

```
ack = [x][y]case x of
        { O z  => S(y)
        | S x' => ack x' (case y of
                { O z  => S(O())
                | S y' => ack x y'} ) };
ack (S(S(O()))) (O());
```

foetus **output:**

```
foetus $Revision: 1.0 $
= <: ack -> ack
< ?: ack -> ack
ack passes termination check by lexical order 0 1
result: S(S(S(O())))
```

3.5 List processing

We define lists over type α as

$$\text{list} := \{\alpha\}\text{Let list} = \{\mathtt{Nil}()|\mathtt{Cons}(\mathtt{HD} : \alpha, \mathtt{TL} : \text{list})\} \text{ in list}$$

The well-known list processing functions `map` : $(\alpha \to \beta) \to \text{list}\alpha \to \text{list}\beta$ and `foldl` : $(\alpha \to \beta \to \beta) \to \beta \to \text{list}\alpha \to \beta$ are implemented and testet.

```
nil = Nil();
cons = [hd][tl]Cons(HD=hd,TL=tl);
l1 = cons (A()) (cons (B()) (cons (C()) nil));

map = [f][list]let
        map' = [l]case l of
                { Nil z => Nil()
                | Cons pair => Cons (HD=(f pair.HD),
                                     TL=(map' pair.TL))}
        in map' list;
map ([el]F(el)) l1;

foldl = [f][e][list]let
        foldl' = [e][l]case l of
                { Nil z => e
                | Cons p => foldl' (f p.HD e) p.TL }
        in foldl' e list;

rev = [list]foldl cons nil list;
rev l1;
```

foetus **output:**

```
nil passes termination check
cons passes termination check
l1 passes termination check

map passes termination check
<: map' -> map'
map' passes termination check by lexical order 0
result: Cons(HD=F(A()), TL=Cons(HD=F(B()), TL=Cons(HD=F(C()),
        TL=Nil())))

foldl passes termination check
? <: foldl' -> foldl'
foldl' passes termination check by lexical order 1
rev passes termination check
result: Cons(HD=C(), TL=Cons(HD=B(), TL=Cons(HD=A(), TL=Nil())))
```

3.6 List flattening

The task is to transform a list of lists into a list, so that the elements of the first list come first, then the elements of the second list and so on. Example: `flatten [[A,B,C],[D,E,F]] = [A,B,C,D,E,F]`. The first version `flatten` : $\mathrm{list}(\mathrm{list}\alpha) \to \mathrm{list}\alpha$ works but fails termination check because of the limited pattern matching abilities of foetus, but it is also bad style and inefficient because it builds a temporary list for the recursive call. However, the second version `f` with a mutual recursive auxiliary function `g` : $\mathrm{list}\alpha \to \mathrm{list}(\mathrm{list}\alpha) \to \mathrm{list}\alpha$ passes termination check.

```
nil = Nil();
cons = [hd][tl]Cons(HD=hd,TL=tl);
l1 = cons (A()) (cons (B()) (cons (C()) nil));
ll = (cons l1 (cons l1 nil));

flatten = [listlist]case listlist of
        { Nil z => Nil()
        | Cons p => case p.HD of
                { Nil z => flatten p.TL
                | Cons p' => Cons(HD=p'.HD, TL=flatten
                                (Cons(HD=p'.TL, TL=p.TL))) }};
flatten ll;
```

```
f = [l]case l of
        { Nil z => Nil()
        | Cons p => g p.HD p.TL },
g = [l][ls]case l of
        { Nil z => f ls
        | Cons p => Cons(HD=p.HD, TL=(g p.TL ls)) };
f ll;
```

foetus **output:**

```
nil passes termination check
cons passes termination check
l1 passes termination check
ll passes termination check

?: flatten -> flatten
<: flatten -> flatten
flatten FAILS termination check
result: Cons(HD=A(), TL=Cons(HD=B(), TL=Cons(HD=C(),
TL=Cons(HD=A(), TL=Cons(HD=B(), TL=Cons(HD=C(), TL=Nil()))))))

<: f -> g -> f
f passes termination check by lexical order 0
? <: g -> f -> g
< =: g -> g
g passes termination check by lexical order 1 0
result: Cons(HD=A(), TL=Cons(HD=B(), TL=Cons(HD=C(),
TL=Cons(HD=A(), TL=Cons(HD=B(), TL=Cons(HD=C(), TL=Nil()))))))
```

3.7 Merge sort

With type

$$\mathrm{bool} := \{\texttt{True()}|\texttt{False()}\}$$

we can define `le_nat`: nat $\to$ nat $\to$ bool and `merge` : $(\alpha \to \alpha \to \mathrm{bool}) \to \mathrm{list}\alpha \to \mathrm{list}\alpha \to \mathrm{list}\alpha$ as follows:

```
merge = [le][l1][l2]case l1 of
        { Nil z => l2
        | Cons p1 => case l2 of
                { Nil z => l1
                | Cons p2 => case (le p1.HD p2.HD) of
```

```
                         { True  z => Cons(HD=p1.HD,
                                           TL=merge le p1.TL l2)
                         | False z => Cons(HD=p2.HD,
                                           TL=merge le l1 p2.TL) }}};

le_nat = [x][y]case x of
        { O z  => True()
        | S x' => case y of
                { O z  => False()
                | S y' => le_nat x' y' }};

i = S(O());
ii = S(S(O()));
iii = S(S(S(O())));
iv = S(S(S(S(O()))));
v = S(S(S(S(S(O())))));
l1 = Cons(HD=O(), TL=Cons(HD=iii, TL=Cons(HD=iv, TL=Nil())));
l2 = Cons(HD=i,   TL=Cons(HD=ii,  TL=Cons(HD=v,  TL=Nil())));
merge le_nat l1 l2;
```

foetus **output:**

```
= < <: merge -> merge -> merge
= = <: merge -> merge
= < =: merge -> merge
merge passes termination check by lexical order 1 2
< <: le_nat -> le_nat
le_nat passes termination check by lexical order 0
result: Cons(HD=O(), TL=Cons(HD=S(O()), TL=Cons(HD=S(S(O())),
     TL=Cons(HD=S(S(S(O()))), TL=Cons(HD=S(S(S(S(O())))),
     TL=Cons(HD=S(S(S(S(S(O()))))), TL=Nil()))))))
```

3.8 Parameter permutation: list zipping

The following function `zip` : listα $\to$ listα $\to$ listα combines two lists into one by alternately taking the first elements form these lists and putting them into the result list.

```
zip = [l1][l2]case l1 of
        { Nil z => l2
        | Cons p1 => Cons(HD=p1.HD, TL=zip l2 p1.TL) };
```

```
zip (Cons(HD=A(), TL=Cons(HD=C(), TL=Nil())))
    (Cons(HD=B(), TL=Cons(HD=D(), TL=Nil())));
```

foetus **output:**

```
? ?: zip -> zip -> zip -> zip
< <: zip -> zip -> zip
? ?: zip -> zip
zip FAILS termination check
result: Cons(HD=A(), TL=Cons(HD=B(), TL=Cons(HD=C(),
     TL=Cons(HD=D(), TL=Nil()))))
```

Here in the recursion of `zip` one arguments is decreasing, but arguments are switched. Thus only a even number of recursive calls produces a structural decrease on `l1` and `l2`. foetus does not recognize `zip` to be terminating because not *every* (direct or indirect) recursive call makes the arguments smaller on any structural lexical order.

Of course there are simple orders that fulfill the demanded criteria, like $<$ on $|l1| + |l2|$. Another solution is to "copy" `zip` into `zip'` and implement *mutual recursion* as follows:

```
zip = [l1][l2]case l1 of
        { Nil z => l2
        | Cons p1 => Cons(HD=p1.HD, TL=zip' l2 p1.TL) },
zip'= [l1][l2]case l1 of
        { Nil z => l2
        | Cons p1 => Cons(HD=p1.HD, TL=zip l2 p1.TL) };

zip (Cons(HD=A(), TL=Cons(HD=C(), TL=Nil())))
    (Cons(HD=B(), TL=Cons(HD=D(), TL=Nil())));
```

foetus **output:**

```
< <: zip -> zip' -> zip
zip passes termination check by lexical order 0
< <: zip' -> zip -> zip'
zip' passes termination check by lexical order 0
result: Cons(HD=A(), TL=Cons(HD=B(), TL=Cons(HD=C(), TL=Cons(HD=D(),
TL=Nil()))))
```

3.9 Tuple parameter

This example, an alternative version of `add` : (`X` : nat, `Y` : nat) $\rightarrow$ nat, shows that foetus loses dependency information if you "pack" and "unpack" tuples.

```
add = [xy]case xy.X of
        { O z => xy.Y
        | S x' => S(add (X=x', Y=xy.Y)) };
```

foetus **output:**

```
?: add -> add
add FAILS termination check
```

3.10 Transfinite addition of ordinal numbers

The type of ordinal numbers is

$$\text{ord} := \text{Let ord} = \{\texttt{O}()|\texttt{S}(\text{ord})|\texttt{Lim}(\text{nat} \to \text{ord})\} \text{ in ord}$$

and `addord` : ord $\to$ ord $\to$ ord can be implemented as follows:

```
addord = [x][y]case x of
        { O o => y
        | S x' => S(addord x' y)
        | Lim f => Lim([z]addord (f z) y) };
```

foetus **output:**

```
< =: addord -> addord
addord passes termination check by lexical order 0
```

3.11 Fibonacci numbers

Iterative version `fib` : nat $\to$ nat of algorithm to calculate the fibonacci numbers fib(0) = 1, fib(1) = 1, 2, 3, 5, 8, ... Only the first parameter is important for termination, the second and the third parameter are "accumulators".

```
fib' = [n][fn][fn']case n of
        { O z  => fn
        | S n' => fib' n' (add fn fn') fn};
fib = [n]fib' n (S(O())) (O());
```

foetus **output:**

```
< ? ?: fib' -> fib' -> fib'
< ? ?: fib' -> fib'
fib' passes termination check by lexical order 0
fib passes termination check
```

3.12 Non-terminating mutual recursion

The following three functions $\mathtt{f}, \mathtt{g}, \mathtt{h}$: nat $\rightarrow$ nat $\rightarrow$ nat are an artificial example for non-termination that has been designed to show to what extent the call graph has to be completed to assure correct results of the termination checker. Function `h` (here $h(x, y) = 0 \; \forall x, y$) could be any function that "looks into" its arguments, e.g. `add`.

```
h = [x][y]case x of
        { O z  => case y of
                { O z  => O()
                | S y' => h x y' }
        | S x' => h x' y },

f = [x][y]case x of
        { O z  => O()
        | S x' => case y of
                { O z  => O()
                | S y' => h (g x' y) (f (S(S(x))) y') } },

g = [x][y]case x of
        { O z  => O()
        | S x' => case y of
                { O z  => O()
                | S y' => h (f x y) (g x' (S(y))) } };

(* f (S(S(O()))) (S(S(O()))); *)
```

foetus **output:** Note that the combined call $f \rightarrow g \rightarrow f$ still does not prevent termination. But then call graph completion finds $f \rightarrow g \rightarrow g \rightarrow f$ that destroys the lexical order 1 0 that was possible until then.

```
< <: h -> h -> h
< =: h -> h
= <: h -> h
h passes termination check by lexical order 0 1
< ?: f -> g -> g -> f
? ?: f -> f -> g -> g -> f
< =: f -> g -> f
? <: f -> f
f FAILS termination check
? <: g -> f -> f -> g
```

```
? ?: g -> g -> f -> f -> g
< =: g -> f -> g
< ?: g -> g
g FAILS termination check
```

4 Termination Checker Overall Outline

4.1 Function call extraction

The task of foetus is to check whether functions terminate or not. Because the foetus language is functional and no direct loop constructs exist, the only means to form loops is recursion. Therefore out of the program text all function calls have to be extracted to find direct or indirect recursive calls that may cause termination problems.

The heart of foetus is a analyzer that runs through the syntax tree of the given foetus program and looks for *applications*. Consecutive applications are gathered and formed in to a *function call*, e.g. in example 3.1, function `add`. There the two applications `((add x') y)` form the call $\mathrm{add}(x', y)$. As you see in this example "add" is always terminating because in each recursive call the first argument x is decreased. foetus stores with each call information about how the arguments of the call (x', y in the example) relate to the parameters of the calling function (here: x, y), the so-called depedencies (here: $x' < x$, $y = y$). We distinguish three kinds of relations: $<$ (less), $=$ (equal) and **?** (unknown, this includes 'greater').

The abilities of foetus to recognise dependencies are yet very limited. So far only three cases are considered:

1. Constructor elimination.
   Be x, y variables and C a constructor, and $x = C(y)$. It follows $y < x$. This is applied in case constructs (see example above).

2. Projection.
   Be x, y variables, L a label, ρ a relation in $\{<, =\}$ and $y \, \rho \, x$. Here it follows $y.L \, \rho \, x$, i.d. a component is considered as big as the entire tuple.

3. Application.
   Be x, y variables, a a vector of terms (arguments of y), ρ a relation in $\{<, =\}$ and $y \, \rho \, x$. It follows $(ya) \, \rho \, x$.

The rule 3 may have a strange looking, but it can be applied in example 3.10 (`addord`). In the third case $x = \mathrm{Lim}(f)$ we have with rule 1 $f < x$ and with rule 3 $(f z) < x$, therefore `addord` is terminating.

4.2 Call graph

In the end the whole of extracted function calls form the *call graph*. It is a multigraph; each vertex represents a function and each edge from vertex f to vertex g a call of function g within the function of f. The edges are labeled with the dependency information (see above) put in a *call matrix*. The call matrix for the only one call add → add in example 3.1 would be

$$\begin{array}{c|cc} & x & y \\ \hline x' & < & ? \\ y & ? & = \end{array}$$

Note that each row represents one call argument and its relations to the calling function parameters.

Now if a function f calls a function g and the latter calls another function h, f indirectly calls h. The call matrix of this *combined call* $f \to h$ is the product of the two matrices of $g \to h$ and $f \to g$. We get the *completed call graph* if we insert all combined calls (as new edges) into the original graph.

To find out whether a function f is terminating you have to collect all calls from f to itself out of the completed call graph (this includes the direct and the indirect calls). When a *lexical order* exists on the function parameters of f so that every recursive call decreases the order of the parameters, we have proven the termination of f. This order we call *termination order*.

We could call the algorithms of call graph completion and finding a lexical order the "brain" of foetus; it is described more precisely and formally in the next section.

5 Formal Description

5.1 Call Matrix

Be $R = \{<, =, ?\}$ set of the relations "less than", "equal to" and "relation unknown". In the context of "$f(x, y)$ calls $g(a, b)$" $a < y$ means "we know that (call) argument a is less than (input) parameter y", $a = y$ means "a is (at least) equal to y (if not less than)" and $a\,?\,y$ means "we do not know the relation between a and y".

With the two operations $+$ and $\cdot$ defined as in table 1 R forms a commutative rig[2] with 0-element **?** and 1-element $=$. The operation $+$ can be understood as "combining *parallel* information about a relation", e.g. if we

[2]On the WWW I found the English term "rig" for what Germans call a "Halbring". This is probably a play of words: Compared to a "ring" a "rig" misses an "n" as well as inverse elements regarding addition. I cite Ross Moore (see

have $a\,\mathbf{?}\,y$ and $a < y$ we have $a\,(\mathbf{?}+ <)\,y$ and that simplifies to $a < y$. The operation $\cdot$ however is "*serial* combination", e.g. $a < y$ and $y = z$ can be combined into $a\,(< \cdot =)\,z$, simplified: $a < z$. $\mathbf{?}$ is neutral regarding $+$ because it gives you no new information, whereas $<$ is dominant because it is the strongest information. Regarding $\cdot$ the relation $=$ is neutral and $\mathbf{?}$ is dominant because it "destroys" all information. Check the table to see which relation overrides which.

$+$	$<$	$=$	$\mathbf{?}$
$<$	$<$	$<$	$<$
$=$	$<$	$=$	$=$
$\mathbf{?}$	$<$	$=$	$\mathbf{?}$

$\cdot$	$<$	$=$	$\mathbf{?}$
$<$	$<$	$<$	$\mathbf{?}$
$=$	$<$	$=$	$\mathbf{?}$
$\mathbf{?}$	$\mathbf{?}$	$\mathbf{?}$	$\mathbf{?}$

Table 1: Operations on R

Now we can define multiplication on matrices over R as usual:

$$\cdot : R^{n\times m} \times R^{m\times l} \to R^{n\times l}$$
$$((a_{ij}),(b_{ij})) \mapsto (c_{ij}) = \left(\sum_{k=1}^{m} a_{ik}b_{kj}\right)$$

Why is this a reasonable definition? Assume you have three sets of variables $\{x_1,\ldots,x_n\}$, $\{y_1,\ldots,y_m\}$ and $\{z_1,\ldots,z_l\}$, a matrix $A = (a_{ij}) \,\epsilon\, R^{n\times m}$ reflecting the relations between the x_is and the y_is (i.d. $a_{ij} = \rho \iff x_i\,\rho\,y_j$) and a matrix $B \,\epsilon\, R^{m\times l}$ reflecting the relations between the y_is and the z_is. Then the matrix product $C = AB$ reflects the relations between the x_is and the z_is. Because

$$c_{ij} = a_{i1}\cdot b_{1j} + a_{i2}\cdot b_{2j} + \ldots + a_{im}\cdot b_{mj},$$

we have e.g. $x_i < z_j$ if we know it by intermediate variable y_1 ($a_{i1}\cdot b_{1j} =$ '$<$') *or* by intermediate variable y_2 *or* ... (to be continued).

`http://www.mpce.mq.edu.au/r̃oss/maths/Quantum/Sect1.html#206`):

> A *rig* is a set R enriched with two monoid structures, a commutative one written additively and the other written multiplicatively, such that the following equations hold:
>
> $$a0 = 0 = 0a$$
> $$a(b+c) = ab + ac, \qquad (a+b)c = ac + ab$$
>
> The natural numbers $\mathbb{N}$ provide an example of a rig.
>
> A *ring* is a rig for which the additive monoid is a group. The integers $\mathbb{Z}$ provide an example.
>
> A rig is *commutative* when the multiplicative monoid is commutative.

Definition. A *call matrix* is a matrix over R with no more than one element different from **?** per row.

$$\mathrm{CM}(n,m) := \{(a_{ij}) \,\epsilon\, R^{n\times m} : \forall i \forall j \forall k \neq j (a_{ij} = \textbf{?} \vee a_{ik} = \textbf{?})\}$$

Remark. The reason we define call matrices this way is these are the only ones foetus produces by function call extraction (see section 4.1). Because foetus recognizes only the three described cases of dependecies, a call argument can only depend of *one* function parameter. But multiple dependecies are imaginable, like in

```
f(x,y) = if (x=0) then 0 else let a=min(x,y)-1 in f(a,x)
```

Here the second call argument a is less than both x and y. The next proposition assures that all matrices foetus will have do deal with are *call* matrices.

Proposition. Matrix multiplication on matrices induces a multiplication on call matrices

$$\cdot : \mathrm{CM}(n,m) \times \mathrm{CM}(m,l) \to \mathrm{CM}(n,l)$$

This operation is well defined.

Proof. Be $A = (a_{ij}) \,\epsilon\, \mathrm{CM}(n,m)$, $B = (b_{ij}) \,\epsilon\, \mathrm{CM}(m,l)$, $AB = C = (c_{ij}) \,\epsilon\, R^{n\times l}$ and $k(i)$ the index of the element of the ith row of A that is different to **?** (or 1, if no such element exists). The we have with the rules in rig R

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} = a_{i,k(i)} b_{k(i),j}$$

Now consider the ith row of C:

$$c_i = (c_{ij})_{1\leq j\leq l} = (a_{i,k(i)} b_{k(i),j})_{1\leq j\leq l}$$

Because at most one $b_{k(i),j}$ is unequal to **?**, at most one element of c_i is unequal to **?**. Therefore $C \,\epsilon\, \mathrm{CM}(n,l)$.

5.2 Call Graph

For each $i \,\epsilon\, \mathbb{N}$ we assume a set $F^{(i)} = \{f^{(i)}, g^{(i)}, h^{(i)}, ...\}$ of identifiers for functions of arity i, $\mathcal{F} = \biguplus_{i\,\epsilon\,\mathbb{N}} F^{(i)}$.

Definition. We form the set of *calls* as follows

$$\mathcal{C} = \{(f^{(n)}, g^{(m)}, A) : f^{(n)} \in F^{(n)}, g^{(m)} \in F^{(m)}, A \in \mathrm{CM}(m, n)\}$$

On calls we define the partial operation *combination of calls*

$$\circ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$$

$$\big((g^{(m)}, h^{(l)}, B), (f^{(n)}, g^{(m)}, A)\big) \mapsto (f^{(n)}, h^{(l)}, BA)$$

Meaning: If g calls h with call matrix B and f calls g with call matrix A, then f indirectly calls h with call matrix BA. $\circ$ cannot be applied to calls that have no "common function" like g, therefore it is partial. $\circ$ can be expanded to sets of calls

$$\circ : \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{C}) \to \mathcal{P}(\mathcal{C})$$

$$(C, C') \mapsto \{c \circ_{\mathcal{C}} c' : c \in C, c' \in C', (c, c') \in \mathrm{Dom}(\circ_{\mathcal{C}})\}$$

Here we combine each call in C with each call in $\hat{C}$ to which $\circ_{\mathcal{C}}$ is applicable and form a set of the combined calls. $\circ_{\mathcal{P}(\mathcal{C})}$ is a total function.

Definition. A *call graph* is a graph (V, E) with vertices $V = \mathcal{F}$ and edges $E \subset_{\text{finit}} \mathcal{C}$. A call graph is *complete* if

$$E \circ E \subseteq E$$

Definition. The *completion* of a call graph (V, E) is a call graph (V, E') such that

(1) (V, E') is complete,

(2) $E \subseteq E'$ and

(3) for all E'' satisfying (1) and (2) we have $E' \subseteq E''$.

Proposition. The completion of a call graph (V, E) is the call graph (V, E') such that

$$c \in E' \iff \exists n > 0, c_1, \ldots, c_n \in E : c_1 \circ \ldots \circ c_n = c$$

Proof.

(1) Be $c \in E' \circ E'$. Then there are $d, e \in E'$ with $c = d \circ e$. Because (V, E') is complete, we have

$$\begin{array}{ll} d = d_1 \circ \ldots \circ d_n & d_1, \ldots, d_n \in E \\ e = e_1 \circ \ldots \circ e_m & e_1, \ldots, e_m \in E \end{array}$$

Thus $c = d_1 \circ \ldots \circ d_n \circ e_1 \circ \ldots \circ e_m \in E'$.

(2) $E \subseteq E'$ is trivial with $n = 1$.

(3) Be (V, E'') complete and $E \subseteq E''$. This gives us $E \circ E \subseteq E''$ from which we gain by induction

$$\underbrace{E \circ \ldots \circ E}_{n-\text{times}} =: E^n \subseteq E'' \text{ for all } n.$$

Now be $c \in E'$. That implies $c = c_1 \circ \ldots \circ c_n$ $(c_i \in E)$ for a suitable n. Hence $c \in E^n \subseteq E''$. $q \cdot e \cdot d$

Proposition. (Completion algorithm) Be (V, E) a call graph, (V, E') its completion and $(E_n)_{n \in \mathbb{N}}$ a sequence of sets of calls defined as follows:

$$\begin{array}{rcl} E_0 & = & E \\ E_{n+1} & = & E_n \cup (E_n \circ E) \end{array}$$

Then there is a $n \in \mathbb{N}$ so that

$$E' = E_n = E_{n+1} = E_{n+2} = \ldots$$

(Obviously the E_n grow monotonously.)

Proof. First we show by induction that $E_n \subseteq E'$ for all $n \in \mathbb{N}$: It is obvious that $E_0 \subseteq E'$. Now be $E_n \subseteq E'$ and $c \in E_{n+1} \setminus E_n$. Then $c \in E_n \circ E$, therefore $c = d_1 \circ \ldots \circ d_n \circ e$, $d_1, \ldots, d_n, e \in E$. It follows $c \in E'$, $E_{n+1} \subseteq E'$.

Second: Because we have a finit set of starting edges E and therefore a finit set of reachable vertices and also a finit set of possible edges between two vertices (limited by the number of different call matrices of fixed dimensions) the E_is cannot grow endlessly. Thus an $n \in \mathbb{N}$ exists with $E_n = E_{n+1}$.

Third: We show that $E' \subseteq E_n$ for that particular n. Be $c \in E'$. Then there exists an m such that $c = d_1 \circ \ldots \circ d_m$, therefore $c \in E_m$. Now if $m \leq n$ then $E_m \subseteq E_n$, otherwise $m > n$ and hence $E_m = E_n$, in both cases $c \in E_n$. $q \cdot e \cdot d$

5.3 Lexical Order

Definition. Be (V, E) a complete call graph and $f^{(i)}$ a function of arity i. We call

$$E_{f^{(i)}} := \{\Delta(C) : (f^{(i)}, f^{(i)}, C) \,\epsilon\, E\} \subset R^i$$

the *recursion behaviour* of function $f^{(i)}$. (Δ takes the diagonal of square matrices).

Each row of this set represents one possible recursive call of $f^{(i)}$ and how the orders of all parameters are altered in this call. The diagonals of the call matrices are taken because we want to know only how a parameter relates to its old value in the last call to $f^{(i)}$. $E_{f^{(i)}}$ of course is a *finite* subset of R^i.

In the following we identify lexical orders on parameters with permutations $\pi \,\epsilon\, S_n$ of the arguments. Often not all of the parameters are relevant for termination; these are not listed in the lexical order and can appear in the permutation in any sequence.

In example 3.11 (`fib'`) only the argument 0 has to be considered to prove termination, the order of argument 1 and 2 are irrelevant and therefore both permutations

$$\pi_1 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix}$$

and

$$\pi_2 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 2 & 1 \end{pmatrix}$$

are valid continuations of the lexical order "0".

Note: In the following we abbreviate the notation of permutations to $\pi_1 = [012]$ and $\pi_2 = [021]$.

Definition. (1) Be B the recursion behaviour of function $f^{(n)}$. We call the permutation $\pi \,\epsilon\, S_n$ a *termination order* for $f^{(n)}$ if

$$\forall r \,\epsilon\, B \exists 1 \leq k \leq n : r_{\pi(k)} = `<' \wedge (\forall 1 \leq i \leq k : r_{\pi(i)} = `=')$$

This definition is a very wide one. In most cases you will look for more special termination orders:

Definition. (2, inductive) Be B the recursion behaviour of a given function. We call the permutation $\pi \,\epsilon\, S_n$ a *termination order* on B if $|B| = 0$

or

$$\exists r \,\epsilon\, B : r_{\pi(0)} = <$$
$$\wedge \; \nexists r \,\epsilon\, B : r_{\pi(0)} = \mathbf{?}$$
$$\wedge \; \pi'_0 \,\epsilon\, S_{n-1} \text{ termination order on } B' := \{r'_{\pi(0)} : r_{\pi(0)} \neq <\} \subset R^{n-1}$$

whereas $\pi'_i = [k_0 \ldots k_{i-1}k_{i+1} \ldots k_{n-1}] \,\epsilon\, S_{n-1}$ given $\pi = [k_0 \ldots k_{n-1}] \,\epsilon\, S_n$ and $r'_i = (k_0, \ldots, k_{i-1}, k_{i+1}, \ldots, k_{n-1}) \,\epsilon\, R^{n-1}$ given $r = (k_0, \ldots, k_{n-1}) \,\epsilon\, R^n$.

The algorithm implemented in foetus searches termination orders like in definition (2); it is a one-to-one transfer of this definition. Every termination order matching definition (2) also matches definition (1) and it can easily be shown that if there is a termination order of type (1) there also exists on of type (2).

Example 5.1 *Be* $E = \{(=, <, ?), (=, =, <), (=, <, =)\}$ *the given recursion behaviour. Then* $\pi_1 = [012]$ *is a type (1) termination order on* E *and* $\pi_2 = [120]$ *is of both types.*

6 Implementation

foetus has been implemented in SML 97. We have used the new Standard ML Basis Library to ensure a safe and possibly optimized handling of standard data structures like lists etc. The parser for the foetus terms has been created with ML-Lex and ML-Yacc. The ML implementation currently used is Standard ML of New Jersey, Version 109.32.

`foetus.lex`	foetus language token specification for `ml-lex`
`foetus.grm`	foetus language grammar for `ml-yacc`
`aux.sml`	auxiliary functions
`closure.sml`	terms and environment
`foetus.sml`	values, evaluation function `hnf`, printing
`matrix.sml`	polymorphic matrices with necessary operations
`simpledeps.sml`	simple implementation of dependecies
`analyse.sml`	static analysis of foetus code
`check.sml`	termination check via call graph
`top.sml`	top level environment
`load.sml`	loader and foetus parser

Table 2: foetus source files

7 Conclusion

We have seen that foetus and its "brain", the call graph completion and finding a lexical order on the function arguments, contributes to automated termination proofs. Of course, in its current state it is no more than a toy to gather experience on his subject. Some improvements have to be done: foetus should be able to recognize more kinds of dependencies (see section 4.1).

- Let assignments. The use of `let`-constructs to save values within functions is discouraged because foetus stores no relations concerning them; it performs no symbolic evaluation during analyzation. For example:

```
case list of
        {Cons pair => let
                hd = pair.HD,
                tl = pair.TL in ...
```

  foetus does not know that `hd` $<$ `list` and that `tl` $<$ `list`. At least such simple assignments (for code shortening) should be handled.

- Tuple handling. foetus should trace the dependencies not only of the whole tuples but also of their components. At the moment you cannot define functions with one tuple as parameter instead of separate parameters and still expect a termination proof (see example 3.9).

- Function results. The reason that foetus cannot prove termination of `div` (see example 3.3) is that it does not know $x \neq 0 \to (y - x = 0 \vee y - x < y)$. But this could be shown for the `sub` function by induction and result in a dependency foetus could use [BG96].

Furthermore the call graph completion algorithm could be adopted to prove termination of parameter permuting functions like `zip` (see example 3.8).

If foetus has "grown older" in the described manner it could be "born into" one of the "adult" program verfication systems or theorem provers like ALF, Isabelle, LEGO or MuTTI **;-)**.

References

[BG96] Jürgen Brauburger and Jürgen Giesl. Termination analysis for partial functions. *In Proceedings of the Third International Static Analysis Symposium (SAS'96), Aachen, Germany, Lecture Notes in Computer Science 1145, Springer-Verlag*, 1996.

[Gie97] Jürgen Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning 19: 1–29*, 1997.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin Löf's Type Theory: An Introduction.* Clarendon Press, Oxford, 1990.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Sli96] Konrad Slind. Function definition in higher order logic. In *Proceedings of TPHOLs 96 (LNCS 1125)*, 1996.

[Sli97a] Konrad Slind. Derivation and use of induction schemes in higher-order logic. In *Proceedings of TPHOLs97 (LNCS 1275)*, 1997.

[TTu97b] Alastair Telford and David Turner. Ensuring Streams Flow. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, 6th International Conference, AMAST '97, Sydney Australia, December 1997*, volume 1349 of *Lecture Notes in Computer Science*, pages 509–523. AMAST, Springer-Verlag, December 1997.