# Type based expansion of finite recursion

## Introduction

In this draft, we outline the ideas of a core programming language featuring a type based termination criteria. The main novelty of the language design is its elaboration that unfolds a function call into a term that has a *exaustive* pattern matching ... over the original arguments and an additional counter, limiting the recusrive call expansion to any specific nonnegative number. This unrolling is needed for the purposes of compiling for eBPF, since it does not allow non-terminating functions to run (and that's why it does not allow unbounded loops or general recursion).

## Syntax

1. Type syntax: Our type syntax consist of named types, denoted by variable $C$, which include basic types (int, bool, string, etc) and user defined types. Our type language features the function space, tuples and a coproduct type.

$$\begin{array}{rcll} \tau & ::= & C^{\iota} & \text{Type name} \\ & | & \tau \to \tau & \text{Function name} \\ & | & (\tau, ..., \tau) & \text{Tuples} \\ & | & \tau \oplus \tau & \text{Coproduct} \end{array}$$

$\iota$

2. Top-level definitions: Our language supports 3 basic forms for top-level definitions. The first is type declarations which introduce a new type name which can be understood as a synonym. The syntax for type definitions is as follows:

$$\text{type } C = \tau$$

which introduces the new type name which is equivalent to type $\tau$. The type $\tau$ must not contain any negative occurrence the newly defined type $C$ in order to ensure termination properly.

Besides type definitions, our language features two classes of functions: total and unrestricted.

Unrestricted functions does not limit recursion and does not need to be type checked by our approach. However, such functions need to be elaborated if they call a total function.

Total functions need to be type-checked by our algorithm since they must be terminating.

| | |
|---|---|
| def $f = e$ | Unrestricted function |
| total $f = e$ | Total function |

3. Expression syntax: The syntax of expressions consists of the traditional constructions for the $\lambda$-calculus extended with a case construct and primitives for tuples, coproducts and a special construct for calling a total function.

| | | | |
|---|---|---|---|
| $e$ | ::= | $x$ | Variables |
| | \| | $\lambda x.e$ | Abstraction |
| | \| | $e\ e$ | Application |
| | \| | case $e$ of $\{\overline{p \to e}\}$ | Case construct |
| | \| | $v$ | Constants for primitive types |
| | \| | left $e$ | Coproduct constructor |
| | \| | right $e$ | Coproduct constructor |
| | \| | $(e, ..., e)$ | Tuple constructor |
| | \| | $f[n]$ | Total function call |
| $p$ | ::= | $x$ | Pattern variable |
| | \| | $(p, ..., p)$ | Tuple pattern |
| | \| | $v$ | Constants |
| | \| | left $e$ | Coproduct pattern |
| | \| | right $e$ | Coproduct pattern |

## Some questions to be answered

1. Does it need to have two different classes for functions?
   - How can we call a total function inside a total function?
     - Probably we can just expand them normally passing the current recursion counter. (maycon) But this can grow exponentially large, as I informally show in the draft.
   - Can we call an unrestricted function inside a total one?
     - (maycon) if unrestricted functions only call total ones, they cannot call themselves, thus they cannot be recursive. But they would be trivially terminating, so *a priori* it is no problem to call them inside total functions. If we allow unrestricted functions to call unrestricted ones, we would be allowing unlimited recursion, and would be turing complete (undesirable). But thinking more deeply, if we allow total functions to call unrestricted functions, we would get to define mutually recursive functions between total and unrestricted. See the example below. Given all this, I think we should not have two classes for functions.

```
def f = lambda x . g[x]
total g = lambda x . f x
```

With this example, we shouldn't allow unrestricted calls inside total functions. But I also don't see how we could typecheck and guarantee `g` to be total if it has a call to a unchecked function (`f`).

# Maycon's Draft

Trying to define some basic functions over lists. Our language does not contain polymorphism, so I cannot define `list a` for all `a`. Let's work with an assumed `listC` instead, with constructors `nil` and `cons C listC`. Remember that `C` can be defined as synonym of any specific type, so our definition will work for arbitrary types without negative occurences, but it won't be *general* in the polymorphism sense.

## map

Without polymorphism, let's work with total functions from `C` to `C`, our previously defined synonym.

total map $= \lambda f : C \to C.\lambda x :$ listC. case $x$ of $\{\text{nil} \to \text{nil}, \text{cons } ps \to \text{cons } (f[p])(\text{map } [fs])\}$

Let's expand the execution for a list of ints and successor function. Assume `+1` is written as $\lambda x : \text{int}.x + 1$.

```
map (+1) (cons 1 (cons 2 (cons 3 (cons 4 nil))))
~
case of (cons 1 (cons 2 (cons 3 (cons 4 nil)))) {cons 1 (cons 2 (cons 3 (cons 4 nil))) -> co
~
case (cons 1 (cons 2 (cons 3 (cons 4 nil)))) of {cons 1 (cons 2 (cons 3 (cons 4 nil))) -> co
{cons 2 (cons 3 (cons 4 nil)) -> map [(+1) (cons 3 (cons 4 nil))]})}
...
map (+1) [1, 2, 3, 4] 5
~
map (+1) [2, 3, 4] 4
~
map (+1) [3, 4] 3
~
map (+1) [4] 2
~
map (+1) [] 1
~
evaluation completed
```

The expansion is tiring but possible, no major problems found. We would need the at least the successor of the list size as the recursion counter. But what if the function was to find the number of prime divisors of a number? This function would need to be recursive (potentially total tho), and it would need to expand

inside each expansion step above. The expansion tree can grow exponentially large when recursive functions are used inside our defined recursive functions and require for really large counters. If our argument function is the sum of the results of two recursive functions applied to the same argument, could they expand in parallel? Or will we need an even greater counter? I think we could pass the same counter to both, but we would still need a huge counter, depending (exponentially) on the most expensive function.

Consider `h` the function that takes an integer `x` and return the sum of two total recursive functions `f` and `g`, both applied to `x`.

```
map h [1, 2, 3, 4, 5, 6, 7, 8] counter
```

The counter should be at least the sucessor of the list size multiplied by greater between the minimum possible counter for `f` and for `g`, considering we allow "parallel" expansion. If inside `f` or `g` makes a call to another total recursive function, our counter should be still greater. It has the potential to grow astronomically large.