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 pattern matching ... FINISH THIS DESCRIPTION

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.

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

ι

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 τ. The type τ 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.

$$\begin{array}{ll} \text{def } f = e & \text{Unrestricted function} \\ \text{total } f = e & \text{Total function} \end{array}$$

3. Expression syntax: The syntax of expressions consists of the traditional constructions for the λ-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?
   - Can we call an unrestricted function inside a total one?