Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 323 (2016) 235–251

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Type Soundness for Path Polymorphism[⋆]

Andrés Viso[a,1] Eduardo Bonelli[b,2] Mauricio Ayala-Rincón[c,3]

[a] *Consejo Nacional de Investigaciones Científicas y Técnicas – CONICET*
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires – UBA
Buenos Aires, Argentina

[b] *Consejo Nacional de Investigaciones Científicas y Técnicas – CONICET*
Departamento de Ciencia y Tecnología
Universidad Nacional de Quilmes – UNQ
Bernal, Argentina

[c] *Departamentos de Matemática e Ciência da Computação*
Universidade de Brasília – UnB
Brasília D.F., Brasil

Abstract

Path polymorphism is the ability to define functions that can operate uniformly over arbitrary recursively specified data structures. Its essence is captured by patterns of the form $x\,y$ which decompose a compound data structure into its parts. Typing these kinds of patterns is challenging since the type of a compound should determine the type of its components. We propose a static type system (*i.e.* no run-time analysis) for a pattern calculus that captures this feature. Our solution combines type application, constants as types, union types and recursive types. We address the fundamental properties of Subject Reduction and Progress that guarantee a well-behaved dynamics. Both these results rely crucially on a notion of *pattern compatibility* and also on a coinductive characterisation of subtyping.

Keywords: λ-Calculus, Pattern Matching, Path Polymorphism, Static Typing

1 Introduction

Applicative representation of data structures in functional programming languages consists in applying variable arity constructors to arguments. Examples are:

$$s = \texttt{cons}\,(\texttt{vl}\,v_1)\,(\texttt{cons}\,(\texttt{vl}\,v_2)\,\texttt{nil})$$
$$t = \texttt{node}\,(\texttt{vl}\,v_3)\,(\texttt{node}\,(\texttt{vl}\,v_4)\,\texttt{nil}\,\texttt{nil})\,(\texttt{node}\,(\texttt{vl}\,v_5)\,\texttt{nil}\,\texttt{nil})$$

⋆ Work partially funded by the international project DeCOPA STIC-AmSud 146/2012, CONICET, CAPES, CRNS.

1 Email: aeviso@dc.uba.ar

2 Email: eabonelli@gmail.com

3 Email: ayala@unb.br

http://dx.doi.org/10.1016/j.entcs.2016.06.015

These are data structures that hold values, prefixed by the constructor `vl` for "value" ($v_{1,2}$ in the first case, and $v_{3,4,5}$ in the second). Consider the following function for updating the values of any of these two structures by applying some user-supplied function f to it:

$$\begin{array}{llll} \mathsf{upd} = f \rightarrow_{\{f:A\supset B\}} & (\,\mathtt{vl}\,z & \rightarrow_{\{z:A\}} & \mathtt{vl}\,(f\,z) \\ & \mid x\,y & \rightarrow_{\{x:C,y:D\}} & (\mathsf{upd}\,f\,x)\,(\mathsf{upd}\,f\,y) \\ & \mid w & \rightarrow_{\{w:E\}} & w) \end{array} \tag{1}$$

Both $\mathsf{upd}\,(+1)\,s$ and $\mathsf{upd}\,(+1)\,t$ may be evaluated. The expression to the right of "=" is called an *abstraction* and consists of a unique *branch*; this branch in turn is formed from a pattern (f), a user-specified type declaration for the variables in the pattern ($\{f : A \supset B\}$), and a body (in this case the body is itself another abstraction that consists of three branches). Type declarations bind variables in both the pattern and the body. An argument to an abstraction is matched against the patterns, in the order in which they are written, and the appropriate body is selected. Notice the pattern $x\,y$. This pattern embodies the essence of what is known as *path polymorphism* [17,19] since it abstracts a path being "split". The starting point of this paper is how to type a calculus, let us call it CAP for *Calculus of Applicative Patterns*, that admits such examples. CAP may be seen as the static patterns fragment of PPC where instead of the usual abstraction we have alternatives. We next show why the problem is challenging, explain our contribution and also discuss why the current literature falls short of addressing it. We do so with an introduction-by-example approach, for the full syntax and semantics of the calculus refer to Sec. 2.

Preliminaries on typing patterns expressing path polymorphism

Consider these two simple examples:

$$(\mathtt{nil} \rightarrow 0)\,\mathtt{cons} \qquad\qquad (\mathtt{vl}\,x \rightarrow_{\{x:\mathsf{Nat}\}} x+1)\,(\mathtt{vl}\,\mathtt{true}) \tag{2}$$

They should clearly not be typable. In the first case, the abstraction is not capable of handling `cons`. This is avoided by introducing singleton types in the form of the constructors themselves: `nil` is given type $\mathbb{nil}$ while `cons` is given type $\mathbb{cons}$; these are then compared. In the second case, x in the pattern is required to be Nat yet the type of the argument to `vl` in `vl true` is Bool. This is avoided by introducing type application [24] into types: `vl` x is assigned a type of the form $\mathbb{vl}$ @ Nat while `vl true` is assigned type $\mathbb{vl}$ @ Bool; these are then compared.

Consider next the pattern $x\,y$ of upd. It can be instantiated with different applicative terms in each recursive call to upd. For example, suppose $A = B = \mathsf{Nat}$, that v_1 and v_2 are numbers and consider $\mathsf{upd}\,(+1)\,s$. The following table illustrates some of the terms with which x and y are instantiated during the evaluation of $\mathsf{upd}\,(+1)\,s$: