Information and Computation 235 (2014) 36–69

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

# Safe typing of functional logic programs with opaque patterns and local bindings ☆

Francisco J. López-Fraguas *, Enrique Martin-Martin, Juan Rodríguez-Hortalá

*Departamento de Sistemas Informáticos y Computación, Facultad de Informática de la Univ. Complutense de Madrid, C/ Prof. José García Santesmases, s/n. 28040 Madrid, Spain*

ARTICLE INFO

*Article history:*
Available online 17 January 2014

*Keywords:*
Functional-logic programming
Type systems
Opaque patterns
Let bindings

ABSTRACT

Type systems are widely used in programming languages as a powerful tool providing safety to programs. Functional logic languages have inherited Damas–Milner type system from their functional part due to its simplicity and popularity. In this paper we address a couple of aspects that can be subject of improvement. One is related to a problematic feature of functional logic languages not taken under consideration by standard systems: it is known that the use of *opaque* HO patterns in left-hand sides of program rules may produce undesirable effects from the point of view of types. We re-examine the problem, and propose two variants of a Damas–Milner-like type system where certain uses of HO patterns (even opaque) are permitted while preserving type safety. The considered formal framework is that of programs without *extra* variables and using let-rewriting as reduction mechanism. The other aspect addressed is the different ways in which polymorphism of local definitions can be handled. At the same time that we formalize the type system, we have made the effort of technically clarifying the overall process of type inference in a whole program.



## 1. Introduction

Type systems for programming languages are an active area of research, no matter which paradigm is considered. In the case of functional programming, most type systems have arisen as extensions of Damas–Milner's [1], for its remarkable simplicity and good properties (decidability, existence of principal types, possibility of type inference, type safety results ...). Functional logic languages [2–4], in their practical side, have inherited almost directly Damas–Milner's types. In principle, most of the type extensions proposed for functional programming could be also incorporated to functional logic languages (e.g. this has been partially done for type classes [5–7]). However, if types are meant to be not only a decoration but are devised to provide safety to programs, then we must ensure that the adopted system has indeed good properties. In this paper we tackle a couple of orthogonal aspects of existing FLP systems that are problematic or not well covered by straightforward adaptations of Damas–Milner typing. One is the presence of so called *higher order (HO) patterns* in programs, an expressive feature allowed in some systems and for which a sensible semantics exists [8]; however, it is known that unrestricted use of HO patterns leads to type unsafety, as recalled below. The second is the degree of polymorphism assumed for local pattern bindings, a matter with respect to which existing FP or FLP systems vary greatly and that is usually not well documented, not to say formalized.

☆ This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, S2009TIC-1465 and UCM-BSCH-GR35/10-A-910502.
* Corresponding author.
*E-mail addresses:* fraguas@sip.ucm.es (F.J. López-Fraguas), emartinm@fdi.ucm.es (E. Martin-Martin), juanrh@fdi.ucm.es (J. Rodríguez-Hortalá).

0890-5401/$ – see front matter 
http://dx.doi.org/10.1016/j.ic.2014.01.004

| | | |
|---|---|---|
| $t \in Pat$ | Set of patterns | Page 39 |
| $e \in Exp$ | Set of expressions | Page 39 |
| $fv(e)$ | Set of free variables | Page 39 |
| $\mathcal{C}$ | One-hole context | Page 39 |
| $ftv(\sigma)$ | Set of free type variables | Page 39 |
| $\theta \in PSub$ | Data substitution | Page 39 |
| $\pi \in TSub$ | type substitution | Page 40 |
| $\mathcal{A}$ | Set of assumptions | Page 40 |
| $\oplus$ | Union of set of assumptions | Page 40 |
| $vran(\pi)$ | Variable range of a substitution | Page 40 |
| $\succ$ | Generic instance relation | Page 40 |
| $\succ_{var}$ | Variant relation | Page 40 |
| $\vdash$ | Basic typing relation | Fig. 5, page 40 |
| $wt_{\mathcal{A}}(e)$ | Well-typed expression wrt. $\vdash$ | Page 41 |
| $\vdash^{\bullet}$ | Extended typing relation | Fig. 7, page 41 |
| | Opaque variable | Definition 1, page 41 |
| $critVar_{\mathcal{A}}(e)$ | Critical variables of $e$ | Definition 2, page 41 |
| $wt_{\mathcal{A}}(\mathcal{P})$ | Well-typed program | Definition 3, page 42 |
| $\Psi(e)$ | Elimination of compound patterns | Fig. 8, page 43 |
| $\rightarrow^{l}$ | Let-rewriting relation | Fig. 9, page 43 |
| $\Vdash$ | Basic type inference relation | Fig. 10, page 45 |
| $\Vdash^{\bullet}$ | Extended type inference relation | Fig. 11, page 45 |
| $\Pi^{\bullet}_{\mathcal{A},e}$ | Typing substitution | Definition 4, page 46 |
| $\mathcal{B}(\mathcal{A},\mathcal{P})$ | Type inference of a program | Definition 5, page 46 |
| $e^{\circ} \in Exp^{\circ}$ | Simple expressions | Page 49 |
| $\vdash^{\circ}$ | Relaxed typing relation | Fig. 13, page 50 |
| $wt^{\circ}_{\mathcal{A}}(\mathcal{P})$ | Well-typed relaxed program | Page 51 |

**Fig. 1.** Summary of notation.

The rest of the paper is organized as follows. Fig. 1 presents the summary of notation. Sections 1.1 and 1.2 make an introductory discussion to the two mentioned aspects. Section 2 contains some preliminaries about FL programs and types. In Section 3 we expose the type system and prove its soundness wrt. *let-rewriting*, an operational reduction semantics for FL programs presented in [9]. Section 4 contains a type inference relation, which lets us find the most general type of expressions. Section 5 presents a method to infer types for programs. In Section 6 we examine some practical limitations of the type system and propose a variant to overcome them. In Section 7 we further discuss some other aspects of the two presented type systems. Finally, Section 8 contains some conclusions and points to future work.

### 1.1. Higher order patterns

In our setting patterns appear in the left-hand side of rules or let-bindings. Some of them can be HO patterns, if they contain partial applications of function or constructor symbols. The use of HO patterns has practical interest—see e.g. [8,10,11] for illustrating examples—and is natural in a setting having an intensional view of functions, where different descriptions of the same 'extensional' function can be observably distinguished.[1] This somehow non-typical behavior does not emerge by the allowance of HO patterns itself; as it is known [9], it stems from the mere combination of HO-functions, lazy evaluation and call-time choice semantics for non-determinism, a cocktail which is present in current FLP systems, whether or not they support HO-patterns. However, HO patterns can be a source of problems from the point of view of the types. In particular, it was shown in [13] that unrestricted use of HO patterns leads to loss of *type preservation*, an essential property for a type system expressing that evaluation does not change types. The following is a crisp example of the problem.

**Example 1** *(Polymorphic casting).* (See [14].) Consider the program consisting of the rules $snd\,X\,Y \rightarrow Y$, *and true* $X \rightarrow X$, *and false* $X \rightarrow$ *false*, with the usual types inferred by a direct adaptation of the classical Damas–Milner algorithm. We can extend the program with the functions $unpack(snd\ X) \rightarrow X$ *and cast* $X \rightarrow unpack(snd\ X)$, whose inferred types will be $\forall\alpha,\beta.(\alpha \rightarrow \alpha) \rightarrow \beta$ and $\forall\alpha,\ \beta.\alpha \rightarrow \beta$ respectively. Then it is clear that the expression $and(cast\ 0)\ true$ is well-typed, because *cast* 0 has type *bool* (in fact it has any type), but if we reduce that expression using the rules of *cast* and *unpack* the resulting expression *and* 0 *true* is ill-typed because 0 has not type *bool*, as is required by the context.

[1] By saying that $e, e'$ represent the same 'extensional' function we mean that $e\ x$ and $e'\ x$ behave the same, for each argument $x$. We remark that some authors [12] have suggested the possibility of other notions of extensionality for which FLP languages would respect extensionality.