Computer Languages, Systems & Structures 44 (2015) 166–180

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Derivation and inference of higher-order strictness types

Sjaak Smetsers [a,*], Marko van Eekelen [a,b]

[a] Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 212, 6525 EC Nijmegen, The Netherlands
[b] Computer Science Department, Open University of the Netherlands, Valkenburgerweg 177, 6419 AT Heerlen, The Netherlands

ARTICLE INFO

Article history:
Received 26 February 2015
Received in revised form
2 June 2015
Accepted 30 July 2015
Available online 10 August 2015

Keywords:
Strictness analysis
Lambda calculus
Typing
Operational semantics
Automated theorem proving

ABSTRACT

We extend an existing first-order typing system for strictness analysis to the fully higher-order case, covering both the derivation system and the inference algorithm. The resulting strictness typing system has expressive capabilities far beyond that of traditional strictness analysis systems. This extension is developed with the explicit aim of formally proving soundness of higher-order strictness typing with respect to a natural operational semantics. A key aspect of our approach is the introduction of a proof assistant at an early stage, namely during development of the proof. As such, the theorem prover aids the design of the language theoretic concepts. The new results in combination with their formal proof can be seen as a case study towards the achievement of the long term PoplMark Challenge. The proof framework developed for this case study can furthermore be used in other typing system case studies.

1. Introduction

In this paper we present a type-theoretic approach to strictness analysis covering both type derivation and type inference.[1] The typing system includes higher-order types as well as user-defined recursive data types. One of our objectives is to formally prove that the higher-order strictness typing is sound with respect to a natural operational semantics. More specifically, we propose to use proof assistants not only for the re-construction of hand-written proofs, but moreover to introduce the tool *during the development* of language theoretic concepts. By introducing the tool in this stage, the consistency of technical concepts can be verified whilst designing them. This is accomplished by checking properties linking these concepts. This paper and the accompanying proof files comprise examples of such concepts and properties. Inaccuracies or mistakes made during the development were most often detected in an early stage, avoiding time consuming and inevitably failing attempts to construct a correctness proof of the main properties. This approach was used for the soundness proof of a non-standard typing system for a simple functional programming language. We combine standard Hindley–Milner typing with strictness information, specifying termination properties of higher-order functions. Strictness information can be used to change inefficient *call-by-need* evaluation into efficient *call-by-value* evaluation. This gain in efficiency lies in the fact that construction of unevaluated expressions (the so-called *closures*) is circumvented.

Combining standard typing with some form of input/output analysis is quite common. We mention a few examples. *Substructural type systems* [24] regulate the order and number of uses of data by ensuring that some values be used at most once, at least once, or exactly once. E.g., linear typing systems (such as uniqueness typing) can be used to identify *unique*

* Corresponding author.
E-mail addresses: S.Smetsers@science.ru.nl (S. Smetsers), M.vanEekelen@cs.ru.nl (M. van Eekelen).

[1] This paper is an extension of an improved version of an earlier conference paper [20] which covers type derivation only.

http://dx.doi.org/10.1016/j.cl.2015.07.004
1477-8424/

objects. These unique objects are suitable for *compile-time garbage collection* which is essential for incorporating destructive updates in functional languages (e.g., see [4,22]). Security-typed languages ([23], for instance) track information flow within programs to enforce security properties such as data confidentiality and integrity. This information can be used to prevent unintentional information leaks. Ref. [21] describes a dedicated typing system for predicting the heap space usage of first-order, strict functional programs. This information can be used in several ways, most notably to ensure that a program allocates sufficient free memory.

Another view on this paper is that it reports on a case study of computer aided verification of theories about syntactic objects. Syntactic theories such as *operational semantics* and *type systems* play an important role in the (static) analysis of computer programs and the construction of reliable implementations of programming languages. The usability and reliability of syntactic techniques can undoubtedly be improved by using automated proof assistants. This need is recognized by many researchers. Most notably, the PoplMark Challenge [1] calls for experiments on verifications of metatheory and semantics using proof tools. The concrete proposal is to formalize existing proofs of properties of type systems with different proof assistants. The long term goal is far more ambitious. It envisions "…a future in which the papers in conferences such as Principles of Programming Languages (POPL) and the International Conference on Functional Programming (ICFP) are routinely accompanied by mechanically checkable proofs of the theorems they claim."

The contribution of our work is threefold. The first contribution is the formalization of a non-standard typing system for strictness analysis of functional programs covering both the derivation system and the type inference algorithm. A first-order version of this typing system was presented in [5]. Compared to traditional strictness analyzers, it has two main advantages. Firstly, it can be combined with ordinary typing: the compiler does not require an additional analysis phase. Secondly, it avoids fixed point computations, resulting in a much more efficient implementation. In this paper we augment first order typing with function types, in effect making it fully higher-order. Compared to common strictness analyses, the resulting system has an additional advantage: it permits the specification and derivation of strictness properties *between* the function arguments. We prove that our system is *sound* with respect to a given natural operational semantics. Thereafter, we discuss the extension of the system needed to deal with recursive data-types.

Secondly, it can be seen as a methodological experiment. We assess the usability of theorem provers for formalizing complex semantical issues, not only after the manual construction of the proofs, but especially during the development of basic theory. The complexity of the typing system in our case study is of a similar level as that of the PoplMark challenge. However, the main proof methods are not known in advance, as is the case in the challenges, but are to be developed during the proof process.

Finally, the PVS formalization can be used as a framework for developing other metatheoretical concepts. The framework can be used as a basis for developing other type based analyses together with their formal soundness proof, living up to the ambition of the long term PoplMark Challenge.

1.1. Overview

Section 2 introduces the core language used in this paper. Basic aspects of strictness are treated in Section 3 including semantic interpretations of (higher-order) strictness types. The derivation rules for deriving higher-order strictness types are given in Section 4. The soundness proof of this typing system is treated informally in Section 5. The formal proof is described in Section 6. Type inference is covered in Section 7. Recursive data structures are dealt with in Section 8. The paper concludes with a discussion of related work in Section 9 and with concluding remarks and ideas for future work in Section 10.

2. Extended lambda calculus

In this section we introduce the core functional language used throughout the paper. This language captures essential aspects such as basic values, abstraction, application, data constructors and destructors, and recursion.

2.1. Syntax

Definition 1. Let $\mathbb{V} = \{x, y, z, x_0, x_1, \ldots\}$ be an infinite set of term variables

- The set Λ of (lambda) expressions is defined by the following abstract syntax:

$$\Lambda ::= \mathbb{V} \mid \square \mid \lambda\mathbb{V}.\Lambda \mid \Lambda\,\Lambda \mid \langle \Lambda, \Lambda \rangle \mid \mathbf{fst}\,\Lambda \mid \mathbf{snd}\,\Lambda \mid$$
$$\mathbf{inl}\,\Lambda \mid \mathbf{inr}\,\Lambda \mid \mathbf{case}\,\Lambda\,\mathbf{of}\,\Lambda\,\mathbf{or}\,\Lambda \mid \mu\mathbb{V}.\Lambda.$$

- The set of free variables of M is denoted by $\mathrm{FV}(M)$. Let $\vec{x} = (x_1, \ldots, x_n)$. We write $\Lambda^{\vec{x}}$ for the set of $\lambda-$terms closed by $\vec{x}$, i.e., $\{M \in \Lambda \mid \mathrm{FV}(M) \subseteq \vec{x}\}$. We write Λ^{o} instead of $\Lambda^{()}$ (expressions with no free variables, so called *closed expressions*).