Science of Computer Programming 124 (2016) 1–19

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Ambiguity and constrained polymorphism

Carlos Camarão [a], Lucília Figueiredo [b], Rodrigo Ribeiro [c,*]

[a] Dep. de Ciência da Computação, Universidade Federal de Minas Gerais, Av. Antônio Carlos 6627, Belo Horizonte, Minas Gerais, Brazil
[b] Dep. de Computação, Universidade Federal de Ouro Preto, ICEB, Campus Universitário Morro do Cruzeiro, Ouro Preto, Minas Gerais, Brazil
[c] Dep. de Computação e Sistemas, Universidade Federal de Ouro Preto, ICEA, João Monlevade, Minas Gerais, Brazil

ARTICLE INFO

Article history:
Received 26 April 2015
Received in revised form 19 February 2016
Accepted 16 March 2016
Available online 30 March 2016

Keywords:
Ambiguity
Context-dependent overloading
Haskell

ABSTRACT

This paper considers the problem of ambiguity in Haskell-like languages. Overloading resolution is characterized in the context of constrained polymorphism by the presence of unreachable variables in constraints on the type of the expression. A new definition of ambiguity is presented, where existence of more than one instance for the constraints on an expression type is considered only after overloading resolution. This introduces a clear distinction between ambiguity and overloading resolution, makes ambiguity more intuitive and independent from extra concepts, such as functional dependencies, and enables more programs to type-check as fewer ambiguities arise.

The paper presents a type system and a type inference algorithm that includes: a constraint-set satisfiability function, that determines whether a given set of constraints is entailed or not in a given context, focusing on issues related to decidability, a constraint-set improvement function, for filtering out constraints for which overloading has been resolved, and a context-reduction function, for reducing constraint sets according to matching instances. A standard dictionary-style semantics for core Haskell is also presented.

1. Introduction

This paper considers the problem of ambiguity in the context of constrained polymorphism.

We use *constrained polymorphism* to refer to the polymorphism originated by the combination of parametric polymorphism and context-dependent overloading.

Context-dependent overloading is characterized by the fact that overloading resolution in expressions (function calls) $e\,e'$ is based not only on the types of the function (e) and the argument (e'), but also on the context in which the expression ($e\,e'$) occurs. As result of this, constants can also be overloaded — for example, literals (like `1`, `2` etc.) can be used to represent fixed and arbitrary precision integers as well as fractional numbers (for instance, they can be used in expressions such as `1 + 2.0`) — and functions with types that differ only on the type of the result (for example, *read* functions can be overloaded, of types *String* → *Bool*, *String* → *Int* etc., each taking a string and generating the denoted value in the corresponding type). In this way, context-dependent overloading allows overloading to have a less restrictive and more prominent role in the presence of parametric polymorphism, as explored mainly in the programming language Haskell.

* Corresponding author.
E-mail addresses: camarao@dcc.ufmg.br (C. Camarão), luciliacf@gmail.com (L. Figueiredo), rodrigo@decsi.ufop.br (R. Ribeiro).

http://dx.doi.org/10.1016/j.scico.2016.03.007
0167-6423/

Ambiguity is however a major concern in context-dependent overloading. The usual meaning of an *ambiguous expression* is, informally, an expression that has more than one meaning, or an expression that can be interpreted in two or more distinct ways.

A formalization of this, with respect to a language semantics definition by means of type derivations, defines that an expression e is ambiguous if there exist two or more type derivations that give the same type and may assign distinct semantics values to e (in the following, $\Gamma \vdash e : \sigma$ specifies that type σ is derivable for expression e in typing context Γ, using the axioms and rules of the type system; $[\![\Gamma \vdash e : \sigma]\!]$ denotes the semantic value obtained by using such axioms and rules):

Definition 1 *(Standard Ambiguity).* An expression e is called *ambiguous* if there exist derivations Δ and Δ' of $[\![\Gamma \vdash e : \sigma]\!]$ and of $[\![\Gamma' \vdash e : \sigma]\!]$, respectively, such that $[\![\Gamma \vdash e : \sigma]\!] \neq [\![\Gamma' \vdash e : \sigma]\!]$, where Γ and Γ' give the same type to every x free in e.

This is equivalent to defining that an expression e is ambiguous if it prevents the definition of a coherent semantics to e [1, page 286], that is, a semantics defined by induction on the structure of expressions where the semantic value assigned to a well-typed expression is not independent of the type derivation.

Without an explicit reference to a distinct definition, ambiguous refers to the standard definition above.

Detection of ambiguity is usually done at compile-time, by the compiler type analysis phase — in Haskell, by the type inference algorithm. Unfortunately, however, detection of ambiguity can not be based on type system definitions, at least for usual definitions, that allow context-free type instantiations, that is, type instantiations that can be done independently of the context where an expression occurs. This causes a well-known incompleteness problem for usual definitions of Haskell type systems [2–4]. This problem is not the focus of this paper.

This paper concentrates instead on another issue related to ambiguity in Haskell, which has not received attention in the technical literature, namely the relation between ambiguity and overloading resolution in the context of constrained polymorphism, in particular the fact that the possibility of inserting new (instance) definitions disregards that an expression may be disambiguated by occurring in some context where there exists a single instance which can be used to instantiate type variables that do not occur in the simple type component of the constrained type.

Specifically, our contributions are:

- A precise characterization of overloading resolution and ambiguity.
- Discussion of Haskell's open-world definition of ambiguity and proposal of a new definition, called delayed-closure ambiguity, that is distinguished from overloading resolution: in the open-world approach, ambiguity is a syntactic property of a type, not distinguished from overloading resolution, whereas with delayed-closure this syntactic property (existence of unreachable variables in constraints) characterizes overloading resolution, and ambiguity is a property depending on the context where the relevant expression occurs, namely the existence of two or more instances that entail the constraint with unreachable variables. Ambiguity is tested only after overloading resolution.

In Section 2 we present Haskell's definition of ambiguity, called open-world ambiguity. In Section 3 we compare open-world ambiguity with the standard, semantical notion of ambiguity.

Substitutions, denoted by meta-variable ϕ, possibly primed or subscripted, are used throughout the paper. A substitution denotes a function from type variables to simple type expressions. $\phi\ \sigma$ and $\phi(\sigma)$ denote the capture-free operation of substituting $\phi(\alpha)$ for each free occurrence of type variable α in σ, and analogously for the application of substitutions to constraints, sets of types and sets of constraints.

Symbol $\circ$ denotes function composition, and $dom(\phi) = \{\alpha \mid \phi(\alpha) \neq \alpha\}$ and id denotes the identity substitution. The restriction $\phi|_V$ of ϕ to V denotes the substitution ϕ' such that $\phi'(\alpha) = \phi(\alpha)$ if $\alpha \in V$, otherwise α.

A substitution ϕ is more general than another ϕ', written $\phi \leq \phi'$, if there exists ϕ_1 such that $\phi = \phi_1 \circ \phi'$.

Section 4 presents an alternative definition of ambiguity, called *delayed-closure* ambiguity, that specifies essentially that:

1. Ambiguity should be checked when (and only when) overloading is resolved. We identify that overloading is resolved in a constraint on the type of an expression by the presence of unreachable variables in this constraint (overloading resolution is defined formally in Section 2). A type variable that occurs in a constraint is called reachable if it occurs in the simple type or in a constraint where another reachable type variable occurs, otherwise unreachable.
   This is unlike open-world ambiguity, where the existence of any type variable that does not occur in the simple type component of a constrained type implies, in the absence of functional dependencies (see below), ambiguity. For example, type *Coll c e* $\Rightarrow c$ of an *empty* member of a class *Coll c e*, is considered ambiguous in Haskell, since type variable e does not occur in the simple type component of the constrained type *Coll c e* $\Rightarrow c$ (despite being reachable). In delayed-closure ambiguity, types with only reachable type variables are not checked for ambiguity, since overloading is still unresolved and may be resolved later, depending on a program context in which it occurs.
2. Constraints with unreachable type variables may be removed if there exists only a single satisfying substitution that can be used to instantiate the unreachable type variables.