Information and Computation 207 (2009) 341–368

Contents lists available at ScienceDirect

Information and Computation

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

A family of syntactic logical relations for the semantics of Haskell-like languages

Patricia Johann [a,1], Janis Voigtländer [b,*]

[a] Department of Computer and Information Sciences, University of Strathclyde, Glasgow, G1 1XQ, UK
[b] Institut für Theoretische Informatik, Technische Universität Dresden, 01062 Dresden, Germany

ARTICLE INFO

Article history:
Received 13 February 2007
Revised 9 November 2007
Available online 7 November 2008

ABSTRACT

Logical relations are a fundamental and powerful tool for reasoning about programs in languages with parametric polymorphism. Logical relations suitable for reasoning about observational behavior in polymorphic calculi supporting various programming language features have been introduced in recent years. Unfortunately, the calculi studied are typically idealized, and the results obtained for them offer only partial insight into the impact of such features on observational behavior in implemented languages. In this paper, we show how to bring reasoning via logical relations closer to bear on real languages by deriving results that are more pertinent to an intermediate language for the (mostly) lazy functional language Haskell like GHC Core. To provide a more fine-grained analysis of program behavior than is possible by reasoning about program equivalence alone, we work with an abstract notion of relating observational behavior of computations which has among its specializations both observational equivalence and observational approximation. We take selective strictness into account, and we consider the impact of different kinds of computational failure, e.g., divergence versus failed pattern matching, because such distinctions are significant in practice. Once distinguished, the relative definedness of different failure causes needs to be considered, because different orders here induce different observational relations on programs (including the choice between equivalence and approximation). Our main contribution is the construction of an entire family of logical relations, parameterized over a definedness order on failure causes, each member of which characterizes the corresponding observational relation. Although we deal with properties very much tied to types, we base our results on a type-erasing semantics since this is more faithful to actual implementations.

1. Introduction

Typeful programming as identified by Cardelli [1] is currently one of the key approaches to producing safe and reusable code. Types serve as documentation of functionality (even as partial specifications) and can help to rule out whole classes of errors before a program is ever run. Typeful programming is particularly effective for pure functional languages such as Haskell [2], where it comes with powerful reasoning techniques connecting the types of functions to their possible observable behaviors. One such technique is the use of logical relations to reason about polymorphic programs.

* Corresponding author.
E-mail addresses: patricia@cis.strath.ac.uk (P. Johann), janis.voigtlaender@acm.org (J. Voigtländer)
[1] Supported in part by National Science Foundation Grant CCF-0429072.

0890-5401/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2007.11.009

Polymorphism is essential for reconciling strong static typing, which attempts to prevent the use of code in unfit contexts by assigning types that are as precise and descriptive as possible, with the goal of flexible reuse. Of the two kinds of polymorphism identified by Strachey [3] — namely, *parametric polymorphism* and *ad-hoc polymorphism* — we are interested in the former here; see the survey [4] for a refined taxonomy. Parametric polymorphism expresses the requirement that a certain functionality is offered for arbitrary types in a uniform manner. Intuitively, this means that the same algorithm is employed in instantiations of a polymorphically typed function at different concrete types. This intuitive uniformity condition was first formally captured by Reynolds [5] through the introduction of the notion of *relational parametricity*, which in turn rests on the concept of *logical relations* [6,7].

The fundamental idea underlying logical relations is to interpret types as relations (rather than as sets, possibly with additional structure). These relational interpretations are built by induction, starting from specific relations for a language's base types (if any), and obtaining interpretations for compound types by propagating relations along the type structure in an "extensional" manner. The key result to be proved for every logical relation constructed in this way is that every function expressible in the underlying language is related to itself by the relational interpretation of its type. This *parametricity theorem*, or certain generalizations of it, can then be used, for example, to derive useful algebraic laws (so-called "free theorems") about polymorphic functions solely from their types [8], or to establish the semantic correctness of efficiency-improving program transformations [9–13]. But for all such applications, the usefulness in practice depends on a good fit between the semantics of the functional language of interest and that of the typically reduced formal calculus for which parametricity results are proved.

Indeed, the applicability to real programming languages of parametricity results obtained for idealized calculi cannot be taken for granted. Simply assuming such applicability is actually quite dangerous, as can be seen from experience with the *selective strictness* feature of Haskell, a language which is otherwise nonstrict. Denotationally specified via the polymorphic primitive

$$\begin{aligned} &seq :: \forall\alpha\ \beta.\ \alpha \to \beta \to \beta \\ &seq\ \bot\ b = \bot \\ &seq\ a\ b = b \quad \text{if } a \neq \bot \end{aligned}$$

in the language definition [2], and routinely used by programmers to control the time and space behavior of their programs, selective strictness was determined early on to have detrimental effects on parametricity. Nevertheless, reasoning about Haskell programs typically took place as if this were not an issue. In fact, Haskell programs were automatically optimized by a compiler using parametricity-based program transformations whose correctness in the presence of selective strictness was a conjecture at best. In the worst case, this means that the compiler can "optimize" a perfectly functioning program into one that fails to terminate or terminates with a runtime error. Conditions under which this can be avoided were first established in [14,15]. These conditions were derived from a new logical relation for which the parametricity theorem holds even with respect to (a naive, but standardly accepted denotational model of) a sublanguage of Haskell which includes selective strictness. A more thorough account in terms of a polymorphic lambda calculus similar to that used as intermediate language in the Glasgow Haskell Compiler (GHC) was recently given in [16]. With the current paper we further advance a line of research whose ultimate goal is the development of appropriate tools for reasoning about parametricity properties of real programming languages rather than toy calculi.

The first new aspect we consider is that of distinguishing different causes of program failure. The notion of "undefined value" is, in some form, fundamental to any semantic treatment of both fixpoint recursion (which is actually the first challenge when extending relational parametricity from Reynolds' original setting to more realistic languages; it is met by Wadler [8] and Pitts [17]) and selective strictness. For example, the notion of "undefined value" is captured by the notation $\bot$ in the above specification for *seq*, where it stands for a nonterminating computation or a runtime error, such as might be obtained as the result of a failed pattern match. Indeed, it is quite common to conflate these different failure causes into a single denotation or observation, but in practice this is not satisfactory. For example, conflating different failure causes means that a program transformation that is claimed to be semantics-preserving may very well transform a nonterminating program into one that instead terminates with a runtime error, and vice versa, or may confuse different kinds of runtime errors. If this happens automatically in a compiler, a debugging nightmare ensues. And this issue is very real. In particular, Haskell examples similar to the ones in [14,15] can be given for which the classical *foldr*/*build*-fusion rule of Gill et al. [9] exhibits this behavior of transforming one kind of error into another one. So it is completely unclear whether the preconditions (on the arguments to *foldr*) found in earlier work to guarantee total correctness of *foldr*/*build*-fusion in the presence of *seq* still do so when considering different failure causes as semantically different. And even partial correctness of *foldr*/*build*-fusion, in the sense that the program after transformation at least semantically approximates the original one, is no longer guaranteed. While in [14–16] it was established to hold unconditionally, the aforementioned examples show that *foldr*/*build*-fusion may transform arbitrary different failures into each other in either direction. So, no matter how different failure causes are ordered by our notion of semantic approximation, some instance of *foldr*/*build*-fusion will violate that order, and thus not even be partially correct.

Actually, this last observation raises an important question. Assuming we consider different failure causes as semantically different, should there at least be some semantic approximation order between them? For example, one might have the intuition that nonterminating programs are strictly less defined than programs that terminate with a runtime error, and