The Logic of Demand in Haskell

Preface

This paper considers the usage of the pattern matching as a tool to control the logic of demand in a lazy functional programming language Haskell. Some theoretical results are given both with an automatic tools to proove P-logic rules for programm validation.

1 Introduction

Haskell is a lazy functional language but by using pattern matching and irrefutable patterns fine control of computation flow may be achieved. The interaction between lazy semantics and pattern matching is complicated and not well known. Since pattern matching provides fine control on computation process it is impossible to specify the finite set of rules for complete validation. Application of P-logic to the Haskell pattern matching is introduced. It provides a formalization of mixed evaluation and simplifies the validation of programms.

2 A Haskell fragment and its informal semantics

The section provides an overview of Haskell approach to data types and case expressions. A formal semantics is also described.

Data types. The data type declarations are needed to describe data constructors, giving a set of signatures of type constructors. Strictness of constructor argument is explicitly specified by the tag `L` or `S`.

Case expressions. Pattern matching is used in a simple case expressions, a left-hand part of definitions or in an explicit abstractions. Since they all are similar only case expressions are focused. A Haskell case expression is construction of the case descriminator and the list of the case branches. The case expressions are evaluated by repeatedly matching patterns of a branches. If no branch matches an unrecoverable error is raised and the computation flow is aborted. The pattern carry out two tasks: control of

the branch to execute and a binding of the variables. Using variable as a pattern matches every descriminator. Using wildcard, underscore (_), is like the variable pattern but no binding occurs. Data constructors in the patterns have to be applied exactly as described in the data type declaration. Pattern is matched only if the descriminator has exact type and all of the subterms matches. Lazy computation of i-th argument in the cases other than irrefutable patterns described below is implied by non-strict declaration in the data constructor. By using operator ~ the evaluation operation of the irrefutable branch pattern is deferred till the pattern values are needed. Then it is fully evaluated. If it fails no alternative is tried and an unrecoverable error is raised. The Haskell example using the Tree data construction highlights how the deferred computation is working and some of the pitfalls are described.

3 Background

Section 3 describes the Haskell specific type-frame semantics and gives an overview of the adopted simple model of ML polymorphism.

Type-frame semantics. Frame model provides «objects» and axioms of representability and functions from objects to objects in terms of an application operator. Also an environment model condition is described.

A simple model of ML polymorphism. An overview of different models of polymorphism is given. Since article focuses on a fine control of demand in Haskell only ML polymorphism is considered. Othori's ML model is used due to it simplicity. Also it is possible to give the predicate semantics to ML polymorphism.

4 Formal semantics of a Haskell fragment

Section 4 describes the formal semantics of Haskell, static and denotational. Since the article focuses the pattern matching the type semantics is given for the patterns. It consists of a pattern and a record type. Latter is introduced to capture variable bindings but not to extend Haskell. The type frame notion is extended to describe the Haskell semantics. It is done in the context of a domain theory and fits in it.

The Haskell fragment. The Haskell type system is provided with some

details but also with special features ommited. It is a convenient type system for implicit polymorphism. First, the formal rules for the patterns are given using a record types in the type rules.

Simple model of polymorphism for the Haskell fragment. The subsection describes how the simple Othori's model of ground types is extended to the semantics of polymorphic terms. Since the original model defines the meaning of polymorphic expressions in terms of the ground instances and the type-indexed set of denotations it has to be adopted to reflect the semantics of Haskell fragment.

Haskell frames. The subsection consists of the detailed description of the Haskell fragment frame. The first part of the frame is complete partial order (cpo) semantics. Note that CPO semantics for typed λ-calculus is frame model. Every Haskell frame has a bottom element. The definitions of bottom of the compound frames are given in a domain theory terms. Currying and uncurrying operators are defined and equations for them are given. Since data type constructors may be lazy recursive infinite types may be defined. Also strictness may be specified explicitly. The saturation strict on i-th argument property is defined to outline the difference between meanings of the «strictness». Data type constructors are presented in the context of type frames. Also the article adduces the case of curried constructors in the same context. As pattern-matching is computed in Maybe monad, the monadic computation must be considered in the type frame semantics. Different aspects of record type are studied in the context of frame semantics. A combination operator on the result of the monadic computation is defined. The Definition of operators $\diamond$ and [] for the simple types using Maybe monad are given.

Typed semantics for the simply-typed Haskell fragment. The denotation semantics is given in this section as a conservative extension of the semantic of ground terms.

5 Logic for the Haskell fragment

The denotation semantics only defines an abstract model, the verification logic is not considered. P-logic is used to verify semantic properties of expressions. This section provides the meaning to the formulas of P-logic by binding them with the formal semantics of the Haskell fragment. Some basic proof rules of P-logic are considered in the context of frame semantics.

Multi-place predicates are not considered since they are too complex and their formal definition of the notion does not fit into the article. The unary predicates and their usage are described both formal and informal. Both strict and lazy P-logic predicates are defined in this section.

Predicates in P-logic. Two ways of the predicate combination in P-logic are described. First, the implicit «lifting» of data type constructors to act as the predicate constructors in P-logic. Second, the «arrow» predicate used to describe the compound predicate that is satisfied by the function-typed expression. Besides two of the atomic unary predicates, the abstract Haskell definition and the typing rules for predicates are provided.

Judgement forms. A judgement form in P-logic is a relation of typing environment and two lists of assertions: the first whose conjunction is an assumption and the second whose disjunction constitutes the conclusion of judgement. As the example property of the standard function map from Haskell prelude is given.

Inference rules for properties of the Haskell fragment. Two types of the inference rules in P-logic are defined: left, when property is on the left side of entailment symbol, and the right one.

Rules asserting an arrow property of the Haskell frame are defined. As the example the increment function for the type Integer is considered and the formal validation rules are provided. Both right and left-introduction rules for the properties of the function application are defined for the strong and weak arrow properties respectively. Lifting data type constructor to the predicate constructor for further applying to the predicates is defined and formal declaration in surface syntax is given. The paper defines the rule asserting difference of resulting predicate constructors lifted from the different data types so the strict type validation may be achieved on the preprocessing stage.

Pattern matching. The algorithms deriving predicates from the Haskell patterns are provided. Derived predicate describes both the control aspect of a pattern and the subterms of matching term. Since patterns may be nested, it is convenient to use the algorithmic calculation of patterns instead of directl formulation of proof rules. Function `pi` implementing desired algorithms of recursive pattern processing and producing pattern predicates is defined and its Haskell code is given. Different examples of the irrefutable patterns are considered, both with the explicit strictness modifiers and without them. The domain of a pattern is defined introducing predicate for a non-deferred match. Both irrefutable (or wildcard) cases and strong matches are covered. Two

rules for the branch computation are defined — one for the match successfull and another for failure. Rules for the case computation are given using `Maybe` monad and the inductive monadic computation of the set of branches. The following examples are considered: one with the match that succeded and another with exception branch computation that generates pattern match failure error (also known as «Non-exhaustive pattern» error).

A semantic interpretation of P-logic. P-logic extends a Haskell by providing the predicate constants and the predicate constructors. The meaning of the simple type is intricated by the meaning of the predicates over underlying frame that interprets the Haskell term type. The article gives the definition and description of the predicate environment. Also the interpretation of the characteristic predicates of sets in a type frame is defined. The paper gives the interpretation of the unversal predicates satisfied by any or only bottom element. The predicate variable is a result of applying the predicate environment map to the name of the variable. The article provides the equation describing the meaning of a predicate formed with the data constructor at a ground instance of the Haskell data type. The function-typed term property characterizing a predicate is also described. The negated and polymorphic predicates are defined too.

Satisfiability and validity of a sequent. A formalization of a well-typed term to satisfy a compatibly typed predicate is given, stating it in the setting of type frame semantics. A ground proposition and a bunch of sequent definitions are declared. The lemma formalizing an assertion that the meaning of a polymorphic predicate cannont depend on the structure of terms of underlaying type is suggested and proved.

6 Soundness of P-logic

Soundness of a logic means that all of its inference rules are coherent with its semantics. An inference rule asserts a propositional implication of a consequent judgement from zero or more antecedent judgement forms.

Soundness of inference rules. An inference rule is sound if the implication it states is valid for a model of the logic. An implication is valid if it is true of a model under all type-compatible assignments to variables.

A reference frame model. The section describes interpreter for the Haskell fragment which later will be used to check the soundness of P-logic rules. It relies on Maybe monad to control the flow among alternate match as Haskell

has explicit monads. Also frame set of Haskell types is defined. It is a set of representations of the saturated applications of its data constructors to elements of the frame sets of their argument types.

Finite models for Haskell types. Type constructions of the Haskell fragment are considered to show how each type or type construction can be represented by a finite type in which to model some rule of P-logic.

Modeling predicates. The way to check the soundness of the rule with simulation of all type–compatile value assignment is given. Notes about Univ and UnDef predicates are also supplied.

Automated model checking of inference rules. Checking the soundness of polymorphic inference rules with interpreter given in previous section is described. Different steps of this process such as predicate assignment or choosing of a type instance are provided in more or less details.

7 Related Work

As part of the Programatica project at the Pacific Software Research Center a formal basis for reasoning about Haskell programs and automated tools for mechanizing such reasoning are devoloping. Previous work on Miranda by Simon Thompson lacks ability to work with partialy undefined or lazy data types hense Miranda itself provides no way to declare them.

Among verification verification tools Sparkle for lazy functional language Clean is outlined. The Sparkle logic has a notation to express an undefined value but does not provide modalities.

Stratego language from `http://stratego-language.org/` suggests algrebra for constructing complex data types by the constructor congruences. This concept may be used in P-logic applied to the predicate constructors and predicate congruences.

Larsen introduces weak-strong modality analogous to the one of P-logic to discriminate *must* and *may* transitions in a process algebra. This analysis was generalized by Huth, Jagadeesan and Schmidt. They provide a semantic interpretation of the modality in a more general framework.

Since all programming logics must confront the issue of undefined values induced by computational environment and some other reasons partial logics that deal with undefinedness have been studied by many of groups. Among them are Owe, Gumb & Lambert, Cheng & Jones, Gries & Schneider, Konikowska and an excellent overview by Farmer.

8 Conclusion

The language fragment which concerns the paper is the patter-matching part of Haskell that has impact on demand. A formalization of the denotational and axiomatic semantics of Haskell pattern-matching is given. Unlike to ML with eager semantics the Haskell pattern-matching is much more complex. Pattern-matching is essentially an eager activity, and is thus harmonious with ML's eager semantics.

The first part of this paper reports on part of a semantics for the whole of Haskell. The denotational semantics was chosen because it allows evaluation of P-logic expressions and sufficiently expressive to specify while language whith varying levels of formality operationally, denotationally, or informally: type classes and overloading, polymorphism, polymorphic recursion, and mixed evaluation. A pure domain-theoretic approach is sufficient in terms of expressivenes but lacks the abstraction for a standart semantics. So frame semantics as a suitably abstract foundation for Haskell was suggested. The representation independence is extremely useful in the proof of soundness, allowing to use model-checking over finite models of types for many rules. Another virtue of frames as a semantic basis for Haskell is their close connection to the semantics of ML polymorphism. Overloading and polymorphic recursion can be neatly expressed in Ohori's setting.

P-logic is a verification logic for all of Haskell, but the article has only provided here the part relative to Haskell's fine control of demand. P-logic allows to formulate properties more precisely using both the weak and strong modalities.

The proof rules of P-logic are sufficiently subtle to validate in mind so the paper suggests the way to mechanize the most detailed parts of a soundness proof. The meta–theory that supports this automatic soundness checking is one of the contributions of the paper.

Appendix

The appendix consists of the detailed proof of two lemmas on associating and binding predicates to the pattern.