Demonstration Abstract: Haskell Type Browser

Matthias Neubauer
Universität Freiburg
neubauer@informatik.uni-freiburg.de

Peter Thiemann
Universität Freiburg
thiemann@informatik.uni-freiburg.de

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type Structure*

General Terms

Languages

Keywords

Haskell, Type Inference, Polymorphism, Type Errors, Debugging

1 Introduction

Despite 25 years of experience with ML-style typing and numerous implementations of type inference algorithms for this kind of type system, programmers are still struggling with error messages reported when the inference algorithm fails. Virtually every Haskell programmer can tell stories about type errors where it took hours to identify the actual problem with the program. While initiated functional programmers may accept this as a fact of life, it makes life especially hard for beginners, in fact, too hard for some.

The root of the problem lies in the operational way in which type inference algorithms produce error message. Most algorithms compose the type of an expression by unification of type fragments during a traversal of the expression's syntax tree. A type error occurs whenever a unification fails. Unfortunately, the point of failure may not even be close to the point of the actual mistake in the program. Hence, the error messages often lead to confusion.

In previous work, we have designed a type system based on discriminative sum types with recursive types and annotation subtyping that enables the precise location of the causes of type errors [5]. The resulting system is a conservative extension of the classical Hindley/Milner type system with parametric polymorphism, and has some additional properties which go beyond the classical system: type inference always produces a principal type derivation (even in case of type errors in the classical sense), and type error reports are extracted from the complete type derivation—that is, type errors are independent of the actual algorithm used to compute the type derivation.

We have implemented these ideas in the Haskell Type Browser tool, a system that allows programmers to interactively explore the type structure of both faulty and type correct Haskell modules. Given a Haskell module, the tool generates an interactive XHTML page rendering a highlighted version of the Haskell source code of the module. Additional navigation elements enable the interactive investigation of the module's type structure in various ways.

2 Using The Haskell Type Browser

Running the type browser is a matter of executing the command `htb` *`module`*`.hs`. The command constructs a type derivation and generates an XHTML page, *`module`*`.hs.html`, and an EcmaScript program, *`module`*`.hs.js`, as output. Viewing the XHTML page in a web browser highlights the different type errors and enables the interactive exploration of the module's typing structure. Figure 1 shows the XHTML view for a small sample module with two type errors.

The navigator pop-up window (shown in the upper-right corner) presents an overview of all type errors in the module. It provides buttons to select between errors and to select different degrees of highlighting. Selecting a type error highlights source locations that may be involved in the type error. Different types of highlighting indicate the different roles of source code fragments in the type error. HTB distinguishes between source fragments that produce, consume, or transfer type constructors.

The type window pop-up (topmost window) appears after clicking on a subexpression in the module. It displays the type of the clicked subexpression and offers buttons to manually navigate through the expression's syntax tree.

3 Implementation Notes

The implementation has two parts. The first part is written in Haskell and implements a considerable part of a Haskell compiler front-end. It translates the source code of the explored Haskell module to XHTML and creates a description of the complete typ-

Haskell'04, September 22, 2004, Snowbird, Utah, USA.

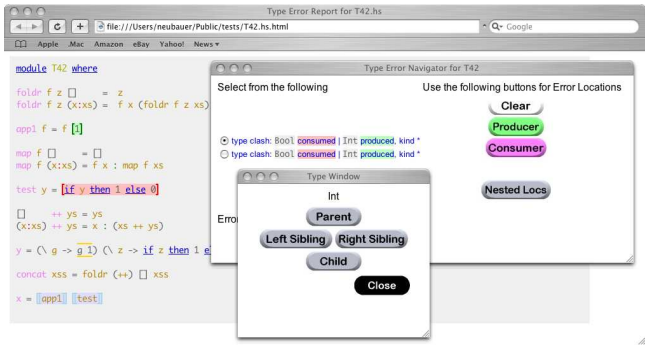

Figure 1. Sample module viewed with the Haskell Type Browser.

ing structure of the module including discriminative sum types and recursive types [5]. The latter information is encoded as one big EcmaScript object literal.

The second part is written in EcmaScript[1] [1] and implements the interactive type browser environment. It runs in a regular web browser after being loaded together with the annotated XHTML output of the Haskell source code. The EcmaScript type browser interprets the type structure object emitted by the first part and adds suitable navigation elements to the web page.

The type browser applies to one module at a time and performs type inference according to our type discipline [5]. One problem in the implementation was the import of type, class, and instance information through the module system. Instead of implementing the Haskell module system, we adapted the compilation manager of a current version of GHC [2] to export the initial typing environment for a module instead of compiling it. We then use GHC's information as additional input for our own type inference system.

The abstract syntax, the parser, and the pretty printer for Haskell modules are adapted from an existing Haskell library [4]. We changed the Haskell syntax library such that the abstract syntax also registers the exact source location of every lexeme of the original source code which allows us to reconstruct the layout of the original source code for the final XHTML output.

The type inference engine is inspired by Jones's work *Typing Haskell in Haskell* [3] and extended in two ways. First, the handling of typing constraints (*e.g.*, type class constraints) follows the HM(X) approach for type inference with constraint types [6]. So far, we added support for one instance of several constraint systems, Haskell 98 type classes. Second, type terms are represented as (mutable) term graphs instead of (immutable) tree structures. This allows us to share common subterms instead of copying terms several times. Also, recursive types are just cycles in the term graphs. Unification is accomplished by applying a union-find algorithm on term graphs. To handle the peculiarity of discriminative sum types (that is, types with several, different type constructors on top), each type variable can carry a sorted list of type constructors instead of just a single type constructor.

The XHTML output page for the source code features unique `id` attributes for each subexpression of the module. In addition, syntactic categories, such as keywords, expressions or variables, are annotated by separate `class` attributes [7]. This way, an EcmaScript program can dynamically change the appearance of parts of the Haskell module.

The second part, the interactive type browser environment, is implemented in EcmaScript and is started after loading the XHTML output and the object literal representing the typing structure into a web browser. After analyzing the typing structure, it opens the navigator pop-up window displaying type errors and installs event handlers on each XHTML node of the source code. If the user selects a specific type error, appropriate source locations are highlighted in the source code window. Clicking on a source location triggers an event handler that opens a second type of pop-up window, the type exploration window.

4 Conclusion

Our preliminary experiments with admittedly small examples demonstrate that the idea of a type browser based on discriminative sum types appears to work. Problem spots that we are currently focusing on is the adequate visualization of recursive types, intelligent navigation in large source files, and the display of flow information.

We are planning to release a first public version of our tool in the near future. The broader application, especially to real world Haskell code, will show whether our approach is generally helpful to understand the type structure of Haskell modules and to locate type errors.

5 References

[1] ECMAScript Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, Dec. 1999. ECMA International, ECMA-262, 3rd edition.

[2] The Glasgow Haskell compiler. `http://www.haskell.org/ghc/`.

[3] M. P. Jones. Typing Haskell in Haskell. In E. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Technical Reports, 1999. `ftp://ftp.cs.uu.nl/pub/RUU/CS/techreps/CS-1999/1999-28.pdf`.

[4] S. Marlow, S. Panne, and N. Winstanley. hsparser: The 100% pure Haskell parser, 1998. `http://www.pms.informatik.uni-muenchen.de/mitarbeiter/panne/haskell_libs/hsparser.html`.

[5] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In O. Shivers, editor, *Proc. International Conference on Functional Programming 2003*, pages 15–26, Uppsala, Sweden, Aug. 2003. ACM Press, New York.

[6] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[7] XHTML 1.0: The extensible hypertext markup language. `http://www.w3.org/TR/xhtml1`, Jan. 2000.

[1] EcmaScript is the standardized language behind JavaScript.