Tool Demonstration: An IDE for Programming and Proving in Idris

Hannes Mehnert
IT University of Copenhagen
hame@itu.dk

David Christiansen
IT University of Copenhagen
drc@itu.dk

Abstract

Dependently typed programming languages have a rich type system, which enables developers to combine proofs with programs, sometimes even eroding the boundary between the activities of proving and programming. This introduces new challenges for integrated development environments that must support this boundary. Instead of reimplementing large parts of a compiler, such as a type checker and a totality checker, we extended the Idris compiler with a structured, machine-readable protocol for interacting with other programs. Furthermore, we developed an IDE for Idris in Emacs, which uses this structured input and output mode, that attempts to combine features from both proof assistant interfaces and rich programming environments such as SLIME. The Idris extension turned out to be generally useful, and has been used in applications such as an IRC bot, an interactive website for trying out Idris, and support for other editors.

1. Introduction

Dependently typed programming unites the previously disparate practices of software development and theorem proving. However, tool support for programming with dependent types typically takes one of these activities to be primary. Additionally, dependently typed programming has unique difficulties, such as the need to define total functions and the need to keep in mind type-level computation when defining terms. Thus, we should expect that good integrated development environments (IDEs) will be at least as valuable for dependent types as they are in traditional programming paradigms. An IDE relies on large parts of a full compiler for a language, such as a parser for highlighting and navigating code, a type checker to report type errors quickly, an interactive proof mode to discharge proof obligations, and so forth. Instead of implementing these parts from scratch and having to maintain these separately, we extended the Idris compiler with a structured input and output mode. This article describes an Idris IDE for Emacs[1], affectionately referred to as `idris-mode`. `idris-mode` is implemented using a high-level communications protocol, which we call Idris's `ideslave` extension. We demonstrate that `ideslave` can support a rich, interactive IDE, but also that it is sufficiently general to support a variety of other tools and interaction models. In our presentation, we will describe both the features of the IDE that we have found useful and the implementation strategy that has allowed the work to be more generally useful.

[1] `https://github.com/idris-hackers/idris-mode`

2. Description

The goal of the Idris language project is to produce a general-purpose programming language with dependent types, in which it is possible to write reliable programs with predictable, fast performance. Idris supports the traditional semantic features of strict functional programming, such as tail-call elimination. Functions and datatypes can be defined similarly to other languages. Additionally, because Idris has full dependent types, the total subset of the language can be used freely in the type system, and pattern-matching recursive definitions can be used to construct proofs in the style of Agda. However, this style of proving is not always the most convenient. Thus, Idris also supports Coq-style tactic proof scripts.

In daily use, Idris requires users to shift their perspective from proofs to programs and back again, repeatedly. Thus, an Idris IDE should support both perspectives concurrently. The Idris IDE for Emacs draws from three primary sources of inspiration:

- The Common Lisp development environment SLIME[2], for writing functional programs in a rich interactive environment;
- Proof General (Aspinall 2000) and Coq IDE, for stepping through tactic scripts;
- and Agda's Emacs mode, for interactive theorem proving with pattern matching.

However, the developers of a compiler are not necessarily in a position to understand what users intend to do with their programming language, and users have their own tool preferences that are separate from compiler writers' taste in editors. Thus, rather than implementing a tightly coupled environment in the style of modern IDEs, we have instead modified the Idris interactive shell to include a machine-readable protocol, similar to that done provided by recent versions of Coq (since 8.4) and by the interaction between SLIME and its connected Lisp compiler.

The queries that are supported by Idris's `ideslave` mode include queries about the state of the compiler, such as *e.g.* requesting the type of a top-level identifier, requesting the built-in documentation associate with a name, or determining which definitions refer to a particular name or are referred to by its definition. Additionally, the Idris read-eval-print loop (REPL) has a number of commands that are intended to be used to implement interactive editing features, such as case-splitting a pattern variable, adding miss-

[2] `https://github.com/slime/slime/`

ing pattern-match clauses to a function definition, and generating `with`-blocks. These features are also supported by the `ideslave` protocol.

Output from the Idris REPL at the command line is semantically colored, helping users to distinguish between defined constants, type constructors, data constructors, and bound variables. The output from `ideslave` extends this, providing semantic information such as documentation summaries, types, and fully-qualified names. In Emacs, this information is shown in tooltips. Additionally, these annotations are used to make every machine-rendered Idris term into a collection of hyperlinks to documentation and type information, and terms occurring inside the documentation are no exception.

While `ideslave` makes no assumptions about the interaction mode of a theorem proving interface, it is straightforward enough to allow for preliminary support for a style of interaction similar to Proof General, where buffers are shown containing the current proof goal and the tactics that have been executed thus far. Users can then step forwards and backwards through the tactic script, in an experience somewhat reminiscent of debugging imperative programs. Stepping through a tactic proof is sometimes the only way to understand why it works, especially if it has not been carefully designed for maintainability.

Perhaps unusually for dependently-typed languages, `idris-mode` supports compiling and executing programs from within the IDE. Users need only select a menu item, and the executing program is attached to an Emacs process buffer. Additionally, `idris-mode` supports expected features such as hopping from an error message to its origin in an Idris source buffer.

The `ideslave` interaction feature was intended to allow multiple text editors to support Idris well with the minimum possible effort. Presently, it is used for an Idris plugin for the Atom text editor[3]. However, it has also proven to be useful for other tools, such as an IRC bot[4] that can normalize terms and look up documentation in Idris's IRC channel and a Web page that allows users to interact with an Idris compiler through their browsers[5]. While we did not design the `ideslave` protocol with these applications in mind, it turned out to be general and flexible enough to be reused by the developers of the mentioned applications.

3. Related Systems

Proof General (Aspinall 2000) is a widely-used interface to a variety of proof assistants. Proof General's interaction model is one of stepping through a proof script, one statement at a time, with the intermediate proof states shown at each step. In many ways, it resembles debuggers for imperative languages. While this model is quite pleasant for working with tactic-based systems such as Coq, it does not provide strong support for defining new functions. Function definitions must typically be typed in as a single unit, and then they are sent to the underlying proof assistant as a whole unit.

Agda mode for Emacs takes a different approach, reflecting that an Agda file is not an imperative script for constructing terms, but instead it is a collection of terms and definitions. In Agda mode, the entire buffer should be type-correct at all times, rather than accepting definitions one at a time. Strong support is provided for working with definitions. The right-hand sides of definitions are allowed to contain *holes*, the contents of which are exempted from type checking. The editor's interactive tools allow users to type-check the contents of a hole and insert them, case-split a pattern variable, attempt to automatically fill a hole, and view the type of a hole's contents and the expected type.

Isabelle/jEdit (Wenzel 2012) supports many of the features that we intend to have in `idris-mode`, such as type-directed completion of names that takes the context into the account, code navigation, and interaction with a tactic-based theorem prover. Unlike Proof General, Isabelle/jEdit allows the modification of tactic proofs at any point, rather than "locking" the region that the prover has accepted. This style of interaction represents an alternative, superior approach to writing tactic-based proofs. However, the implementation complexity is far higher than a system in the style of Proof General, and tactic proofs are a small enough part of the daily practice of Idris development that it seems unlikely to be worth the investment of effort.

4. Conclusion and Future Work

We designed and implemented an `ideslave` extension for the Idris compiler (Brady 2013) that provides a machine-readable protocol for interacting with the REPL. Additionally we develop an interactive development environment for the editor Emacs which is used by at least a dozen developers. Furthermore, the `ideslave` extension is used to provide an interactive REPL in the Idris IRC channel, a plugin for the Atom editor, and a web-based Idris editor.

In the future we plan to provide more features such as source code navigation, including cross references and jumping to the definition of a function; semantic highlighting à la Agda in which source files are decorated in the same manner as compiler output. Furthermore, we want to incrementally parse and elaborate source files so that the interactive features work on buffers that do not yet type check, but are under development.

References

D. Aspinall. Proof General: A generic tool for proof development. In S. Graf and M. I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000. ISBN 3-540-67282-6.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

M. Wenzel. Isabelle/jEdit — a Prover IDE within the PIDE framework. arXiv:1207.3441, 2012.

[3] `https://github.com/fangel/atom-language-idris`

[4] `https://github.com/idris-hackers/idris-ircslave`

[5] `http://www.tryidris.org/console`, source `https://github.com/puffnfresh/tryidris`