Edit-Time Tactics in Idris

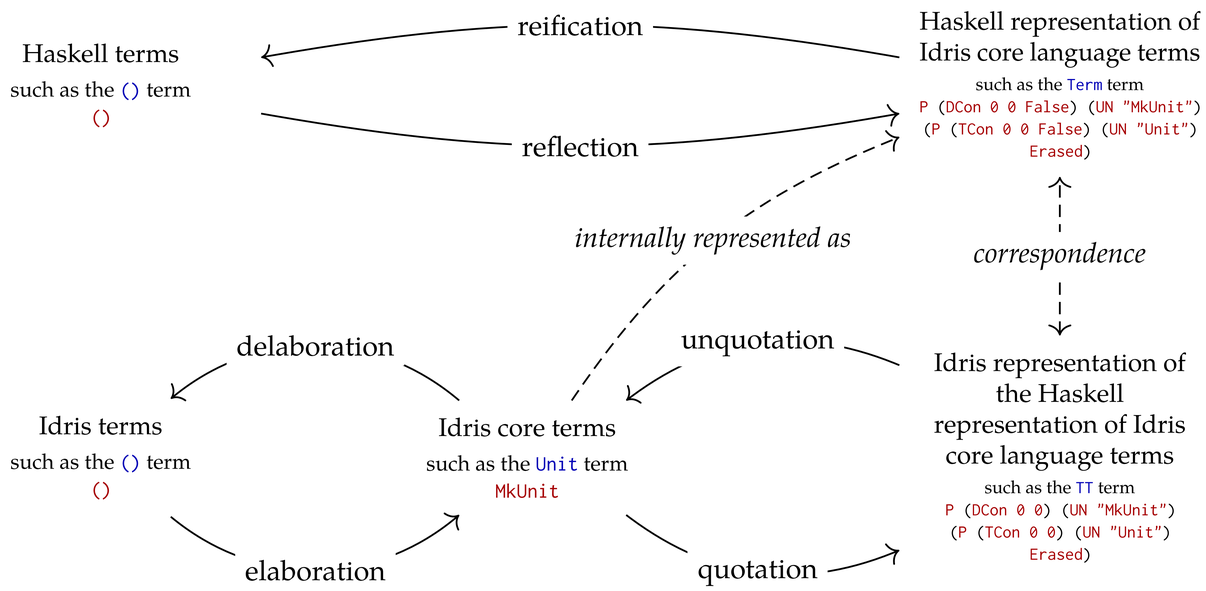
Joomy Korkut
Advisor: Daniel R. Licata
Wesleyan University

Abstract

Metaprogramming allows users to write programs that write programs. In dependently-typed languages such as **Idris**, recent work on **elaborator reflection**[1] paved the way for new applications of metaprogramming by showing that it can be a substitute for **tactic-based proof languages**. The goal of our work is to use elaborator reflection to write editor interaction actions in Idris.

Now we can write new tactics as `Elab` actions and run them from the editor, i.e. in **edit-time**. The old editor interaction mode only had a limited number of built-in actions, like type-checking holes, case-splitting, and lemma extraction. Our work extends its abilities to anything that can be done with tactics. We present the design and implementation of this feature for the Idris compiler.

Crash course on the Idris compiler and Idris metaprogramming



How it works in Emacs

Suppose we have the following snippet:

```
%editor
prover : TTName -> Elab TT
prover = hezarfenTT True

noContradiction : (p : Type)
                -> Not (p, Not p)
noContradiction = ?a
```

We want to fill the hole `?a` automatically, by running the `prover` tactic in the editor. Place the cursor on `?a` and execute the shortcut we assigned in Emacs for `prover`. We get:

```
noContradiction =
  \p, d => void (snd d (fst d))
```

(`prover` uses Hezarfen, which we describe in applications.)

The `Editorable` interface

Editors communicate with the compiler via S-expressions, so users should be able dictate how to convey a value of a given type. We define an interface in Idris, which can specify serialization and deserialization of the terms of a type into an `SExp` and vice versa.

```
interface Editorable a where
  fromEditor : SExp -> Elab a
  toEditor : a -> Elab SExp
```

Usually we can define `Editorable` implementations in Idris itself. But for Idris types that represent the Haskell representation of Idris core terms, such as `TT` and `TyDecl`, we need to define `Editorable` implementations primitively in order to be able to parse, elaborate, delaborate, and pretty print them.

Type-checking with `Editorable`

We want to ascertain during type-checking that an `Elab` action we will use as an editor action is suitable for that task. That requires checking that all components of the type signature have an implementation of `Editorable`. Let's introduce a new function modifier syntax **`%editor`**, which enforces this constraint in compile-time.

If we have a function

```
toy : TTName -> TT -> Elab TT
```

and we declare it an editor action with **`%editor`**, then the types `TTName`, `TT` (from the argument), and `TT` (extracted from the return type) must have implementations of `Editorable`, so that we are sure we know how to deserialize the arguments to `toy` and serialize its result.

Toy example

```
%editor
toy : TTName -> TT -> Elab TT
toy n t = do
  (_, _, ty) <- lookupTyExact n
  case ty of
    `(Nat) => pure t
    _ => fail [ NamePart n
              , TextPart "is not a Nat!"]
```

We can run `toy` from the editor with a hole name and some string representing a `TT` term. If the type of the hole is `Nat`, then `toy` returns the term it was given, otherwise fails.

Idris code cannot tell Emacs to replace the name under cursor with some expression; we have to write 5-10 lines of trivial Emacs Lisp code to glue the `Elab` action to our editor. *(If we are using Emacs, of course.)*

Useful applications

(1) Replace the existing built-in editor actions (written in Haskell as a part of the compiler) with edit-time tactics (written in Idris). In our work, we present a replacement for the "add clause" action.

(2) Write a proof search mechanism better than the built-in one, as an edit-time tactic. In our work, we present **Hezarfen**, a theorem prover tactic that decides intuitionistic propositional logic, generates `TT` proof terms, and simplifies the proof terms as much as possible.

[1] David Christiansen and Edwin Brady. Elaborator reflection: extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ACM, 2016.