Wesleyan ♦ University

EDIT-TIME TACTICS IN IDRIS

by

Joomy Korkut

Faculty Advisor: Daniel R. Licata

A thesis submitted to the faculty of Wesleyan University in partial fulfillment of the
requirements for the degree of Master of Arts

Middletown, Connecticut   **May 2018**

Abstract

Metaprogramming allows users to write programs that write programs. In dependently-typed languages such as Idris, recent work on *elaborator reflection* [16] paved the way for new applications of metaprogramming by showing that a tactic-based proof language can be substituted with a monadic interface that exposes the internal elaborator. The goal of our work is to use elaborator reflection to write editor interaction actions in Idris.

Currently in Idris modes of editors such as Emacs, users can perform actions like type-checking holes, case-splitting, and lemma extraction. Implementations of all of these Idris editor actions are hard-coded in the compiler, and they are written in Haskell. Our work will allow us to rewrite them in Idris as metaprograms and to move them into an Idris library, instead of having them embedded into the compiler.

Furthermore, Idris users can write our own tactics through elaborator reflection and run them from the editor, i.e. in edit-time. This would extend the abilities of the editor interaction mode from the current built-in features to anything that can be done with tactics. In our work, we present the design and implementation of this feature in the Idris compiler.

We also implement an intuitionistic theorem prover tactic, which is meant to be an better alternative to the built-in proof search editor action, and an add-clause tactic that exemplifies how we can move some of the hard-coded features from the compiler to a library.

"What do compilers do? They manipulate programs! Making it easy for users to manipulate their own programs, and also easy to interlace their manipulations with the compilers manipulations, creates a powerful new tool."

Tim Sheard and Simon Peyton Jones [52]

Acknowledgements

First and foremost, I would like to thank Dan Licata, my research advisor. Even though the topic I picked was not very close to the work he does, he agreed to work with me on it and gave me very helpful advice whenever I got stuck. Since this project also had a user interface aspect, his insight as a long-time user of proof assistants shaped my decisions. He has had more impact on my research taste than anyone, and I am indebted to him for his kindness and patience.

I am incredibly fortunate to have met David Christiansen last summer and talked to him about his work on metaprogramming for dependent types. In his dissertation [18] he laid the foundation that makes this project possible, and he later pitched me the idea of using elaborator reflection for interactive editing. I could not have finished this work without him putting up with my incessant stream of questions (my words, not his) or without his extensive comments on my draft.

I would like to express my gratitude to James Lipton and Edward Morehouse for reading and evaluating my thesis. I would also like to thank Cyrus Omar for coining the term "edit-time tactics". Even though he did so in a different context [46, 47], I cannot think of a better way to describe this project. I would like to thank Matthew Wilson for reading my draft many times, and helping me put my project in perspective with our conversations about the history and future of IDEs and programming.

I want to thank Joseph Cutler and Mitchell Riley for the programming languages discussions we had over the year; I am grateful for our little PL community.

Finally, I am thankful to Yulia, Mehmet, Recep, Kivanc, Isin Ekin and my family for their emotional support. This has not been an easy year for me, but I made it through thanks to all of you.

Contents

List of Figures

CHAPTER 1

Introduction

The rising popularity of statically typed functional languages led to the programming methodology of "type-driven development" (hereinafter referred to as TDD), which is a style of development that revolves around the types of missing parts in programs. The expressions users have not written yet, or ones they do not know how to write, can be left as holes in their program, and the program still compiles, with a warning that there are incomplete parts. A hole is kind of expression, and since these languages are statically typed, those holes have types. The parts programmers have already written dictate what the types of the remaining parts should be. These types can guide them when they try to fill the holes to complete the program. In other words, TDD allows writing programs incrementally and top-down. This not only lets them type-check programs at every step, but it also gives them clever hints about the following steps, based on the types and the local contexts.[1] This idea was originally born in the world of proof assistants, and then it was borrowed by more practical functional languages. Interactive editing based on this idea has been used in proof assistants for a long time; the Edinburgh LCF system [29], HOL and Isabelle [44] have allowed users to prove theorems incrementally.[2] Users type in commands called "tactics" that update the proof state by changing the goal, or by creating subgoals. Unlike the ones above, there are tactic-based proof assistants like Coq [12] and Lean [21] that generate a Curry-Howard style proof term at the end; these proof assistants have focused on changing the proof term indirectly. Inspired by these systems, others arose which provide a new

---

[1] Especially as dependent types gradually sneak into mainstream languages, like they already have to Haskell [25] and Scala [3], we predict that TDD will only get more popular.
[2] Propositions and types are distinct in these systems.

interface and type theory that focuses on changing the proof terms directly by incrementally building them up. One of the earliest such proof assistants was the ALF proof editor [36], and the idea was developed further in Epigram [39] and Agda [45]. The ideas that are brewed in these proof assistants continue to inspire more mainstream functional languages.

The traditional programming workflow either depended on saving a file and trying to compile (or run) with every edit, or in the Lisp tradition it depended on the read-eval-print loop (REPL). This has changed with the advancement of integrated development environments (IDEs), programs that have certain functionalities such as code completion, syntax checking, displaying compiler error messages on their corresponding lines, searching in documentation, etc. Through these features, IDEs enable rapid feedback and interaction cycles between the user and checking tools of the language and computation environment, but they are different from TDD. While IDEs can also use types to assist the user, they do not direct the entire development process around types *per se*. TDD is not a program; it is merely a style of programming in which the development process takes the form of a conversation between the type-checker/compiler and the user's editor/IDE. However, this requires certain changes to the compiler, such as being able to type-check incomplete expressions and definitions [11].
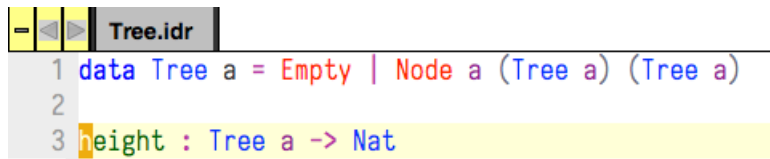
The kind of change that is important for this work is the editor interaction mode (or the IDE mode, as it is called in Idris [10]) that lets the editor talk to the compiler.[3] There are various existing examples of compilers and editors that do this sort of interaction: Proof General [5] and CoqIDE for Coq, the Emacs mode [20] for Agda, the Emacs mode [41] for Idris, jEdit [57] for Isabelle, and the editor mode of Lean.[4] Idris has a special place among these proving languages since it tries to prioritize general purpose

---

[3] To avoid any confusion, we should mention that there are two different parts of an editor interaction mode. The first is a plugin to the editor, often written in the script language of the editor, such as Emacs Lisp or VimL. The second part is a separate program that does the heavy lifting of the editing features that work with the language itself. `ghc-mod` in Haskell and `agda-mode` in Agda would be perfect examples for the second part. We can call these parts the frontend and backend of the editor interaction mode, respectively. When we talk about the language the editor interaction mode is implemented, we mean the language used in the backend, because the language used in the frontend depends on the editor.

[4] Haskell's typed holes in GHC are exciting, their editor interaction features are not as polished as the ones Agda and Idris have, since GHC/Haskell does not come with a built-in editor interaction system.

programming. Compared to mainstream languages that have mature IDEs, Idris still has an unusual standing since it also is a proof assistant [2]. This unique position of Idris provides motivation for editor features on par with mature IDEs for other languages, as well as the TDD-style development workflow via editor actions. Therefore, our work strives to bring the long-standing traditions of proof assistants and IDEs closer together, by introducing the "edit-time tactics" feature. The word "edit-time" is a wordplay on the terms "compile time" and "run time", and it means that we run the tactics when we are still writing our program in the editor. The area of bringing IDE features to proof assistants is not new [8, 35, 43, 54], but it has been focusing on their usefulness as proof tools, not necessarily how they can help mainstream programmers. In contrast, we take tactics from proof assistants and see how they can help type-driven developers.

Before we proceed to describe our work, it is imperative to understand what exactly an editor action is. Editor actions are commands in the editor that make a meaningful change in our code, or ones that give us some information about our code. For example, if we are trying to define a function to compute the height of a binary tree, we can just start by writing the type for the function.
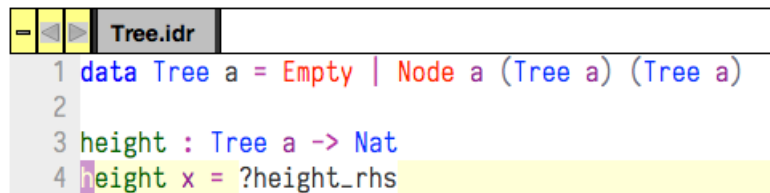
```
Tree.idr
1 data Tree a = Empty | Node a (Tree a) (Tree a)
2
3 height : Tree a -> Nat
```

Note that initially we only declared the type of the function, and nothing else. To get an initial incomplete definition, we can run the editor action "Add initial match clause to type declaration," while the cursor is on the type declaration.
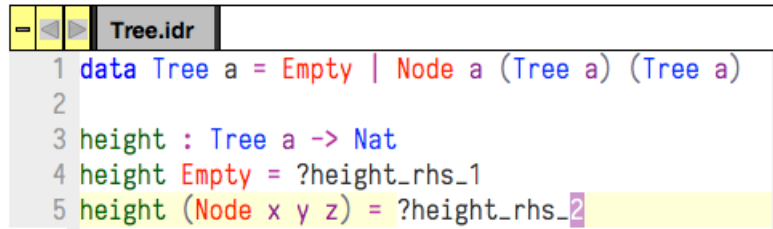
```
Tree.idr
1 data Tree a = Empty | Node a (Tree a) (Tree a)
2
3 height : Tree a -> Nat
4 height x = ?height_rhs
```

Now we have a definition for `height` that takes one argument `x`, and returns `?height_rhs`. This is clearly incomplete; we want to change the return value based on

what x is. So the next step would be to inspect what values x can take. We can place the cursor on x and then run the editor action "Case split pattern variable."
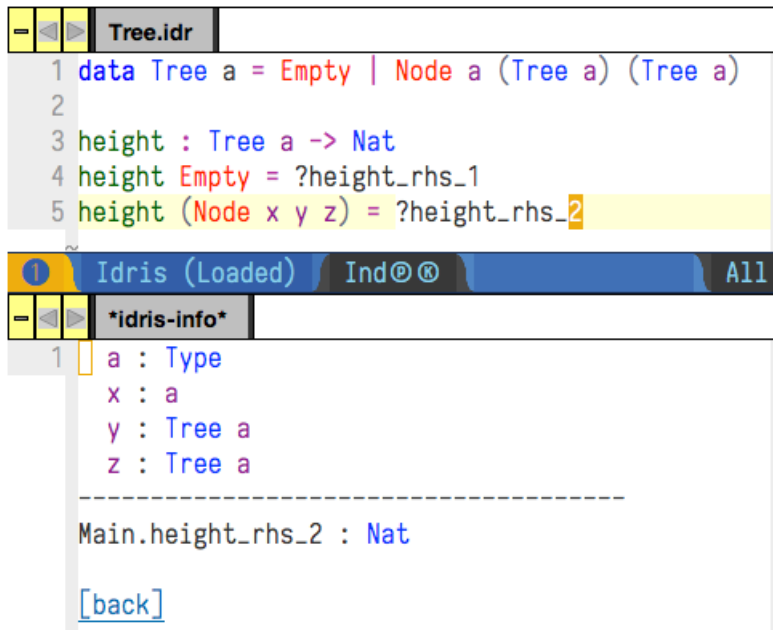
```
Tree.idr
1 data Tree a = Empty | Node a (Tree a) (Tree a)
2
3 height : Tree a -> Nat
4 height Empty = ?height_rhs_1
5 height (Node x y z) = ?height_rhs_2
```

Now we have two holes that we have to complete, namely `?height_rhs_1` and `?height_rhs_2`. When the cursor is on one of the holes, we can run the editor action "Display type" and see what type of expression should replace the hole, and what names are in the local context, i.e. are available to use when writing that expression.
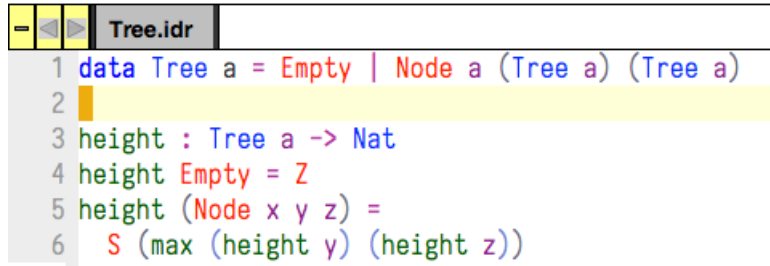
```
Tree.idr
1 data Tree a = Empty | Node a (Tree a) (Tree a)
2
3 height : Tree a -> Nat
4 height Empty = ?height_rhs_1
5 height (Node x y z) = ?height_rhs_2

  Idris (Loaded)    Ind⊛⊗                       All
  *idris-info*
1   a : Type
    x : a
    y : Tree a
    z : Tree a
    ------------------------------------
    Main.height_rhs_2 : Nat

    [back]
```

We complete the function by filling the holes by writing the expressions.

```
Tree.idr
1 data Tree a = Empty | Node a (Tree a) (Tree a)
2 
3 height : Tree a -> Nat
4 height Empty = Z
5 height (Node x y z) =
6   S (max (height y) (height z))
```

This example should show how much editor actions can shape our programming experience.

1.1. Motivation

Now that we are familiar with what an editor action is, we can describe the problem our project solves. The editor actions we reviewed above are features embedded in the compiler. If you want to define a new action, the only way possible is to change the compiler source code, build your own version of the compiler, and then edit the source code of your editor mode to use that feature you added. This is far from ideal: no one should have to fork a compiler just to add a custom editor action. Maintaining a compiler fork and navigating through the compiler source code are usually not in the skill sets of most users. Another drawback is that users would have to learn Haskell and recompile the entire Idris system every time they want to define a custom editor action.

Therefore, we want to give users a way to write custom editor actions. Our solution for this is to make use of elaborator reflection [16, 24] in Idris, which is a metaprogramming machinery that allows users to automate the construction of proofs and programs, by reflecting the elaborator monad [10] in the Idris compiler. Christiansen and Brady showed that this mechanism is powerful enough to replace the old tactic language [16] that existed in the previous versions of Idris, which is now deprecated in favor of elaborator reflection.

Elaborator reflection adds a primitive monad `Elab` to Idris itself, in which type-checking and normalizing terms, looking up types and definitions of functions are

monadic actions.[5] Our thesis argues that these actions provide a nice interface with which users can define their custom editor actions. This has the following advantages:

- Implementations of the Idris editor actions mentioned above are built in to the compiler, and they are written in Haskell. Our work will allow us to rewrite them in Idris as `Elab` actions. This way, we can remove these parts from the compiler and move them into an Idris library.

- The abilities of the editor interaction mode are extended from the current built-in features to anything that can be done with tactics. This allows library and DSL authors to provide domain-specific editor actions.

- Defining editor actions with a monadic interface allows us to compose them easily. For instance, if we had case-splitting as an `Elab` action, we could define a tactic to case-split on many arguments at the same time.

- More people can extend Idris; contributing to the Idris standard library or publishing a library of editor actions is much easier than extending the compiler itself.

To see what an edit-time tactic would look like, let's see an actual example in Emacs.

```
 9 %editor
10 prover : TTName -> Elab TT
11 prover = hezarfenTT True
12
13 noContradiction : (p : Type) -> Not (p, Not p)
14 noContradiction = ?a
~
~
 Idris (Loaded)    Ind⊗⊗                    unix | 14:19    Bottom
Editor.a
```

---

[5] These monadic actions are still called tactics, especially if they change the goal queue or the local context, hence the title of this thesis. Note that whenever we use the word "tactic" in the context of Idris, we exclusively refer to the monadic `Elab` actions, not the old tactic language.

In this example, we have defined an editor action `prover` using the tactic we explain in section 5.2, set up an Emacs shortcut for it, and then run it on a hole that we want to fill, using the tactic. We get the following result:

```
 Editor.idr
 9 %editor
10 prover : TTName -> Elab TT
11 prover = hezarfenTT True
12
13 noContradiction : (p : Type) -> Not (p, Not p)
14 noContradiction = \p, c => void (snd c (fst c))
 ~
 ~
 Idris (Not loaded)   Ind®®              unix | 14:18   Bottom
C-c C-p
```

1.2. Contributions

We make the following specific contributions in this thesis:

- We extend the primitive `Elab` monad with the necessary primitive monadic actions that make writing an editor action with elaborator reflection possible (section 4.1).

- We define an Idris interface (or *type class* in Haskell terminology) called `Editorable` for serializing and deserializing Idris expressions. For the reflected type that represents the core language terms of Idris, implementations of this interface are primitives (section 3.2).

- We extend the Idris compiler to track the association between source code and typing contexts (section 4.4).

- The current proof search mechanism in Idris is not particularly advanced. We write an alternative proof search tactic called Hezarfen, a full-blown theorem prover for intuitionistic propositional logic, based on Dyckhoff's LJT [23], and then we show how to use this on holes when we are in the editor (section 5.2).

- We define an add clause tactic that can be run from the editor, which can replace the hard-coded add clause editor action (section 5.1).

Some of the features we implemented in this thesis have already made their way to the Idris compiler, and the rest also will once they are reviewed by the other Idris contributors.

We envision three different audiences for this thesis:

(1) Idris programmers who use editor actions in their editor, who will now have access to more editor actions. For them, reading the applications of edit-time tactics in chapter 5 would be the most helpful.

(2) Advanced Idris programmers who want to write simple editor actions, using the common Idris types and reflected types. More advanced Idris programmers may want to write more complex editor actions that involve data types that they define. They may want to read chapter 3 in order to understand the design of our feature and what should be taken into account when defining such data types.

(3) Compiler developers and contributors for Idris and other dependently-typed languages. They may want to read chapter 4 in order to observe what we needed to change in the compiler, and how they can add this feature to a different language.

Now that we have gotten a taste of edit-time tactics and we know what to look for while reading this thesis, we review the basics of the Idris programming language and its metaprogramming machinery in chapter 2. We discuss the design of the edit-time tactics feature in chapter 3 and their implementation in chapter 4.

Before we move on the other chapters, it might be helpful to take a brief look at the typographic conventions we use in this thesis: when we have code excerpts, we will use a monospace font. Keywords will be written in **black boldface**, types will be blue[6], constant and function names will be green, data type constructors and primitives will be red, bound variables will be purple, holes will be cyan, and comments will be dark gray.[7] We will have code excerpts in different programming languages such as Idris,

---

[6] Links to URLs and different parts of the thesis are also in blue, but not in monospace font.

[7] We are following the color conventions in Conor McBride's Epigram paper [39].

Haskell, and Emacs Lisp in this thesis. For that reason we will always explicitly state what language the code is in, but we will use the same highlighting for all of them.

CHAPTER 2

Background

2.1. Idris

Idris is a dependently typed functional programming language. In simple terms, dependent types allow us to do computation in types, just like we can do computation in terms [18]. Moreover, we can use the computation in types to shape our definitions of computations in terms.

A helpful intuition for functional programmers is to think about how the concept that functions are values was initially a novel idea, and now in dependently typed programming, we promote types to values as well. Thus a function can now take a term as an argument and return a type as a result [7, 32]. For those familiar with Haskell type families [51], this is similar to that, but notice that type families are like functions from types to types, while dependent types allow us to have functions from *terms* to types.

In Figure 2.1, we define a data type of vectors. It is exactly like lists, except now the length is stored within the type. In other words, as we add elements, the length is computed at the type level.

After that, we define a function that zips two vectors. Notice that only two cases are enough to cover all possible paths of this function. If we were to define a `zip` function for lists, we would need four cases: both empty, both non-empty, one empty and one non-empty, and one non-empty and one empty. However, our `zip` function takes two arguments that are of the same length `n`.[1] Therefore, we cannot have a case with one empty and one non-empty, because that contradicts the fact that both vectors are of the same length. Notice that the `zip` function returns a vector of length `n`. In other words, the input vectors and the resulting vector are guaranteed to be of the same length.

---

[1] Both the `Nat` named `n` and the `Type` named `a` here are implicit quantifiers.

FIGURE 2.1. Example of a dependently typed Idris code: vectors

```idris
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 elem
  (::) : elem -> Vect len elem -> Vect (S len) elem

zip : Vect n a -> Vect n b -> Vect n (Pair a b)
zip [] [] = []
zip (x :: xs) (y :: ys) = (x, y) :: zip xs ys
```

In Figure 2.2, we define two similar data types that ensure that the given natural number is even or odd, and compute the number within the type at every step. Later, we give a function that takes a natural number and generates either a value of the type that ensures evenness or one that ensures oddness. This corresponds to a proof that for all $n \in \mathbb{N}$, n is even or n is odd.

FIGURE 2.2. Example of a dependently typed Idris code, parity of natural numbers

```idris
data Even : Nat -> Type where
  EvenZ : Even 0
  EvenSS : Even n -> Even (S (S n))

data Odd : Nat -> Type where
  Odd1 : Odd 1
  OddSS : Odd n -> Odd (S (S n))

total
evenOrOdd : (n : Nat) -> Either (Even n) (Odd n)
evenOrOdd 0 = Left EvenZ
evenOrOdd 1 = Right Odd1
evenOrOdd (S (S n)) = case evenOrOdd n of
                        Left ev => Left (EvenSS ev)
                        Right o => Right (OddSS o)
```

Unlike other dependently typed languages like Agda and Coq, Idris is not total by default. This is because Idris prioritizes general purpose programming rather than theorem proving. However, users can opt in to totality checking either for the entire

module or for specific declarations.[2] We did the latter for `evenOrOdd` by using the keyword **total**. Similarly, we could require the `zip` function from the previous example to be total if we wanted to.[3]

Haskell's type classes and type class instances are called *interfaces* and *implementations* in Idris, respectively. In Haskell there can only be one instance for the same type class and type, but in Idris there can be multiple implementations for the same interface and type. You can name implementations and specify the implementation you want to use by its name when you are writing a function. For our purposes, we will not use multiple implementations.

2.2. Elaborator reflection

Idris programs are elaborated from high-level Idris syntax trees into a core language called `TT`, and then type checked [10]. The implementation of the Idris elaboration in the compiler is written as a Haskell monad called `Elab`. Recent work on elaborator reflection [16] allowed Idris users to access this monad from Idris itself, by implementing a primitive monad `Elab` in Idris itself, that can only be used for metaprogramming in compile time.

2.2.1. Reflected core language types. Since the `Elab` monad in Idris needs to work with core language terms and definitions, we have to use data types in Idris that represent them. We want to have a correspondence between the internal data type that represents the core language syntax in the compiler and the ones defined in Idris.

In this thesis, we will use the term *reflection* to refer to "the capability of converting some piece of concrete code into an abstract syntax tree object that can be manipulated in the same system" [55].[4] In other words, we will define Idris types that let us work with Idris syntax trees within Idris. We will call these types "reflected types" for now. We give a diagram in Figure 4.2 to explain the relationship between the metalanguage

---

[2] Both for functions and data type definitions, since the latter are checked for strict positivity.

[3] Clearly Idris cannot decide whether an arbitrary function is total, since that would solve the halting problem. Instead it acknowledges the ones that are obviously total, and for all the other ones, even if they are actually total, it throws a totality check error.

[4] Note that this is somewhat at odds with the usage of the term "reflection" in normalization by evaluation.

Haskell, the object language Idris, the core language of Idris and the conversions from one to another. The diagram provides an overall look, which is not required to understand this section, but finishing this section is necessary to fully comprehend the diagram. Nevertheless, looking at the diagram before reading this section might be helpful to some readers.

The most important reflected type is called TT, which represents the core language's typed terms, and its definition can be seen in Figure 2.3.

FIGURE 2.3. The reflected type TT in Idris.

```
──── Idris ────
data TT : Type where
  P : NameType -> TTName -> TT -> TT
  V : Int -> TT
  Bind : TTName -> Binder TT -> TT -> TT
  App : TT -> TT -> TT
  TConst : Const -> TT
  Erased : TT
  TType : TTUExp -> TT
  UType : Universe -> TT
```

As a quick summary:

- P creates a variable term from a name, as defined in Figure 2.5, and the type of the variable.

- V creates a de Bruijn variable. (given integer n representing the nth most recently introduced local variable)

- Bind creates any kind of binder (lambda, let etc.) with a term it binds on.

- App creates a function application.

- TConst creates a constant such as an integer, a character, a string etc.

- Erased creates a term that is not known. This is used for erasing the types we do not need later in the compilation.

- TType creates a type of types for a given universe.

- UType creates a uniqueness type for a given uniqueness universe.

This summary is meant to be an overview, so refer to Brady [10] and Christiansen and Brady [16] if this is not perfectly clear. For our purposes, we will mostly be concerned with `P`, `Bind` and `App`.

The other important type that is used in elaborator reflection is `Raw`, which is the type of untyped core language terms, and its definition can be seen in Figure 2.4.

FIGURE 2.4. The reflected type `Raw` in Idris.

```Idris
data Raw : Type where
  Var : TTName -> Raw
  RBind : TTName -> Binder Raw -> Raw -> Raw
  RApp : Raw -> Raw -> Raw
  RType : Raw
  RUType : Universe -> Raw
  RConstant : Const -> Raw
```

The constructors of `Raw` are almost the same ones as `TT`, except a few of them are missing and variables do not have to be annotated with their types. This makes `Raw` terms easier to type by hand, if necessary. Therefore `TT` terms will usually be treated as the output from the type checker, and `Raw` terms will be the input to the type checker.

The `TTName` type is the type of names in the core language, its full definition can be seen in Figure 2.5.

FIGURE 2.5. The reflected type `TTName` in Idris.

```Idris
data TTName : Type where
  UN : String -> TTName
  NS : TTName -> List String -> TTName
  MN : Int -> String -> TTName
  SN : SpecialName -> TTName
```

As a quick summary:

- `UN` represents user-provided variable names without any namespace.

- NS represents variable names with a given namespace. For example, the name Prelude.Bool.True is represented as NS (UN "True") ["Bool", "Prelude"].

- MN represents machine generated names with a hint string and a fresh integer for that hint.

- SN represents special names, which are used for metavariables, implementations etc. We will not deal with them in this thesis.

As a quick way to refer to Idris names, there is a syntactic sugar `{{x}} that would give you the term UN "x". Similarly, there is another syntactic sugar that checks whether a given name exists and lets you refer to an existing name without having to specify its full namespace: `{False} would give you NS (UN "False") ["Bool", "Prelude"].

There are many other types used in the reflection of the core language, but we will not give their definitions here since they are not as common as TT, Raw, and TTName. However, it is useful to at least list the most important ones and describe what they represent.

- TyDecl represents type declarations.
- DataDefn represents data type definitions.
- FunDefn represents function definitions.
- FunClause represents a single clause in a function definition.

2.2.2. Quotations. Writing TT and Raw terms by hand can get tedious, hence there is a quotation syntax that elaborates a given expression into its corresponding TT or Raw term [17]. The syntax `(e), where e is an Idris expression, gives us the typed or untyped core language syntax tree for e. For example, `(not True) gives us the following TT term:

FIGURE 2.6. The TT term we get when we quote not True.

```
──────────────────────────────── Idris ────────────────────────────────
App (P Ref
       (NS (UN "not") ["Bool", "Prelude"])
       (Bind (UN "__pi_arg")
             (Pi (P (TCon 8 0) (NS (UN "Bool") ["Bool", "Prelude"]) Erased)
                  (TType (UVar "./Prelude/Bool.idr" 71)))
             (P (TCon 8 0) (NS (UN "Bool") ["Bool", "Prelude"]) Erased)))
    (P (DCon 1 0)
       (NS (UN "True") ["Bool", "Prelude"])
       (P (TCon 0 0) (NS (UN "Bool") ["Bool", "Prelude"]) Erased))
```

The Raw term for the same expression is a bit smaller:

FIGURE 2.7. The Raw term we get when we quote not True.

```
──────────────────────────────── Idris ────────────────────────────────
(RApp (Var (NS (UN "not") ["Bool", "Prelude"]))
      (Var (NS (UN "True") ["Bool", "Prelude"])))
```

Obviously, we would not want to write terms like these manually every time we want to return the syntax tree for a simple function application. At times like this, quotation saves us, for both expressions and patterns.

We can also give the type of the expression we want to elaborate, which becomes necessary when Idris cannot infer the type. For True, it is trivial to infer that type is Bool, but for 5, the type can be Int, Integer, Nat, or anything that satisfies the Num interface. Therefore, we have to specify the type when we are quoting. The syntax for that is `(e : t), e.g. `(5 : Nat).

We also can do antiquotation. If we have some variable expression x that has the type TT or Raw, then we can construct a syntax tree using it within the quotation, with the syntax `(not ~x). Note that antiquotation works for expressions, not just variables. The type of expression or variable we have in the antiquotation has to match the type of the quotation. In other words, only TT expressions can be used in an antiquotation in a TT quotation, and *mutatis mutandis* for Raw.

For further information on Idris' quotations, see Christiansen [17].

2.2.3. `Elab` monad. The elaborator reflection [16] feature that has been added to the Idris compiler recently provides a tool for metaprogramming with a monad called `Elab`. This monad is implemented as a primitive and it can only be run during compile time, or in the interactive proof shell.

Elaborator reflection adds a new declaration `%runElab` e to Idris, where e has the type `Elab ()`. This declaration runs the `Elab` action and adds new type declarations, function and data type definitions generated by the `Elab` action generated by e to the context.

Elaborator reflection also adds a new *expression* `%runElab` e to Idris, where e has the type `Elab ()`. The type t of the entire expression is started as the goal of the `Elab` action, and the tactics in e must solve the goal that has the type t. Like the declaration above, this expression also adds the newly generated declarations and definitions to the context.

The `Elab` monad holds a proof state inside, which has a goal type, a proof term that is incrementally built up, a hole queue, a collection of open unification problems, and a supply of fresh names [16]. This state is really held in the Haskell `Elab` monad, though it can be observed from Idris.

Tactics can change the proof state. Here are some examples that do that:

- `claim : TTName -> Raw -> Elab ()`

  Creates a new hole with a given name and a type.

- `fill : Raw -> Elab ()`

  Create a guess to fill the current hole with a term. Fail if the types do not unify [38].

- `solve : Elab ()`

  Try to finalize the guess in the hole. Fail if there is no guess [38].

There are a lot more tactics, which we will not list here. A more thorough list can be found in Christiansen and Brady [16] and Idris documentation.

We also have access to `Elab` actions that do not change the proof state, but give us access to the context or other compiler primitives:

- `check : List (TTName, Binder TT) -> Raw -> Elab (TT, TT)`

  Type-checks a term under a given environment and gives the typed core term version of the `Raw` term and the type of it as a typed core term.

- `normalise : List (TTName, Binder TT) -> TT -> Elab TT`

  Normalizes[5] a typed term under a given environment.

- `lookupTy : TTName -> Elab (List (TTName, NameType, TT))`

  Looks up the type of the given name and returns the ones it finds in a list, in case the name is ambiguous.

Observe that in some of these functions, for inputs we use `Raw`, the untyped core language terms, and results are in `TT`, the typed core language terms. This is because untyped core language terms are easier to write for the tactic users, and type-checking them in the elaborator is easy.

Now let's define a function using elaborator reflection. Take the polymorphic identity function, for example.

FIGURE 2.8. The identity function using elaborator reflection in Idris.

──────── Idris ────────

```
id : (a : Type) -> a -> a
id = %runElab (do intro `{{ty}}
                  intro `{{a}}
                  fill (Var `{{a}})
                  solve)
```

For anyone familiar with Coq, this will look very similar to a normal Coq proof. First we take the type as an argument, and then a value of that type, and we return the same value. Elaborator reflection proofs look a bit more unpolished compared to Coq

---

[5] `normalise` is spelled the British way, since most Idris development happens in the UK.

proofs[6], but it is essentially very similar to Coq tactics, hence the name "tactics" we use to refer to monadic `Elab` actions.

Let's prove a lemma in Idris. This time we want to prove that $(\forall n \in \mathbb{N})\ n = n + 0$, for the standard definition of addition. Since that requires more complex tactics like induction, we will import the Pruviloj[7] library [18].

FIGURE 2.9. A proof that $(\forall n \in \mathbb{N})\ n = n + 0$ using elaborator reflection in Idris.

```
                                  Idris
nPlusZero : (n : Nat) -> n = plus n 0
nPlusZero = %runElab (do intro `{{n}}
                         induction (Var `{{n}})
                         compute
                         reflexivity
                         compute
                         attack
                         intro `{{n1}}
                         intro `{{indHyp}}
                         rewriteWith (Var `{{indHyp}})
                         reflexivity
                         solve)
```

The proof proceeds as follows: we first take in the argument `n`, and then do an induction on `n`. Because of the way induction works in Pruviloj, we have to simplify the goal using `compute`[8]. For the base case, the goal is just proving `0 = 0`. For the inductive step, we have to restructure the goal with `attack` and then reintroduce the input and then introduce the induction hypothesis. Then we rewrite the goal with the induction hypothesis and then the goal becomes trivial. Understanding this proof completely is not crucial for this thesis, but if you want to fully comprehend `attack` and `solve`, you can refer to Christiansen and Brady [16].

---

[6] This is because Coq has the Gallina language for proof terms and Ltac [22] for tactics, therefore, while Idris does not have such a distinction; it only has one language: Idris itself. Therefore the syntaxes for core language terms, quotation and special names look more cluttered. Another reason that elaborator reflection looks more unpolished is that Coq tactics are designed to just inhabit a type, while `Elab` is designed for *programs* where we want to control the precise computational behavior.

[7] Pruviloj is the Idris library included in the Idris distribution. It contains complex tactics written with elaborator reflection.

[8] Its Coq equivalent would be `simpl`.

Finally let's do an example for a declaration with elaborator reflection:

FIGURE 2.10. A type declaration and new definition for n, using elaborator reflection in Idris.

```
                           ─── Idris ───
%runElab (do declareType (Declare `{{n}} [] `(Nat))
             defineFunction (DefineFun `{{n}}
                               [MkFunClause (Var `{{n}}) `(Z)]))
```

The example above first declares that n will have the type Nat. Then it defines it as

n = Z.

Now that we have seen different use cases for elaborator reflection, we can move on to the design of the edit-time tactics feature.

CHAPTER 3

Design

The goal of this project is to allow users to write custom editor actions using elaborator reflection. The editor and the compiler have to communicate to do that; the editor must send the name of the action and the necessary information, and the compiler should then send the result back. In this chapter, we will look at how the built-in editor actions work, and then see what kind of restrictions this brings, and how this shapes the design of edit-time tactics.

3.1. Communication

The current Idris implementation[1] of the editor interaction mode is a part of the Idris compiler, and is written in Haskell. The editor runs an instance of the `idris` executable with the `--ide-mode` flag, which allows socket communication with the program through a machine-readable syntax.[2] To be more precise, the compiler receives S-expressions [40] as input over the socket and sends back S-expressions as output.

Let's revisit the editor actions we looked at in chapter 1. We have the following piece of code, which will complete to the height function on binary trees. The next step is to add the initial function clause.

FIGURE 3.1. Initial `height` function.

```Idris
height : Tree a -> Nat
```

---

[1] This thesis is using Idris 1.2.0.
[2] The Idris mode of Vim works differently, since Vim did not support asynchronous jobs until version 8.0. This should change in the near future.

When we place the cursor on `height` and run the action to add the initial function clause, the editor sends the message on the first line in Figure 3.2 to the compiler, after doing the necessary computation, the compiler responds with the message on the second line.

FIGURE 3.2. The communication for add clause editor action on `height`.

```
———————————————————— S-expression ————————————————————
-> ((:add-clause 4 "height") 8)
<- (:return (:ok "height t1 = ?height_rhs") 8)
```

Let's look at how these messages are formed. The built-in add clause command needs to know the line number after which we are adding a clause, which is 4, and the name of the function we are adding a clause for, which is `height`. Also, for communication purposes, we require that messages have unique IDs, which is what the number 8 we have at the end of the message is. Observe that the response also carries the same number.

The response contains `:ok`, which means the clause adding succeeded, and then a string that contains a line of code. When the editor receives that, it adds this new code to the next line. Note that that last step is done by the front-end of the editor interaction mode, i.e. in Emacs Lisp if we are in Emacs.

The next step in writing the function `height` is to look at the possible cases of the tree, i.e. a case-splitting action. This is the piece of code we have before that action:

FIGURE 3.3. `height` function after adding an initial clause

```
———————————————————————— Idris ————————————————————————
height : Tree a -> Nat
height t1 = ?height_rhs
```

When we place the cursor on `t1` and run the case-splitting action, the communication in Figure 3.4 happens between the editor and the compiler.

FIGURE 3.4. The communication for case-split editor action on `height`.

```
                            ───── S-expression ─────
-> ((:case-split 5 "t1") 11)
<- (:return (:ok
      "height Empty = ?height_rhs_1\n
       height (Node x t1 t2) = ?height_rhs_2\n") 11)
```

The built-in case-split command needs to know the line number on which we are case-splitting, which is 5, and the name of the pattern variable we are splitting, which is `t1`. The response contains `:ok`, which means the case-splitting succeeded, and then a string that contains two lines split by the new line character. When the editor receives this information, it replaces the line the cursor was on before with the new code it received.

We have now seen how exactly the current editor action communication between the editor and the compiler works through S-expressions. Now we want to introduce an S-expression format that can capture *any* of these commands: we want to generalize the ones we have seen so far.

Suppose we have the following `Elab` action that we want to run from the editor: `prover : TTName -> Elab TT`. We will see in detail in section 5.2 how this tactic is implemented. For now let's only look at how it is used. It takes a name of a hole, and returns the proof term to fill the hole with. Let's try to prove a trivial lemma with this tactic.

FIGURE 3.5. Type declaration for a simple theorem `f`, with an incomplete definition.

```
                               ───── Idris ─────
f : Either Unit Void
f = ?q
```

When we place the cursor on `?q` and tell the editor to run the `prover` tactic, we have to convey a couple things to the compiler: We have to tell that we want to run an `Elab` action named `prover`, and we are running this on the hole `?q`, and where exactly the

cursor is (in line and column numbers) when we run this editor action. Here is the communication we want to have for this:

FIGURE 3.6. The communication for the custom `prover` action on the hole `?q`.

```
                               S-expression
-> ((:elab-edit "prover" '("q") 21 5) 22)
<- (:return (:ok "Left ()") 22)
```

The editor tells the compiler that we want to run an `Elab` action named `prover`. But the actions we may want to run take arguments, therefore we pass a list S-expression `'("q")` that corresponds to the arguments `prover` takes. The editor also tells the compiler where the cursor was when the user called the editor action, in this case 21 is the line number and 5 is the column number. Remember from the previous communications that the last number, which is 22 here is the communication ID.

In the response we get from the compiler, we get a piece of code that is supposed to replace the hole. In the next section, we will discuss what determines the kind of S-expression we should send and receive for a given type.

3.2. Types of editor actions

Users of our work will write `Elab` actions that will then be called from the editor via S-expressions that are sent to the compiler. We outlined above that in the S-expression, we need to pass the arguments of the specific `Elab` action we want to call. Since we have to be able to send the arguments in an S-expression and also send the result back in another S-expression, we have limits on what kind of arguments an S-expression can take. In other words, all arguments and the result of an `Elab` editor action have to serializable and deserializable.

Since communication is done via S-expressions, we need to reflect the type of S-expressions to Idris.

FIGURE 3.7. The reflected type SExp in Idris.

```Idris
data SExp : Type where
  SExpList : List SExp -> SExp
  StringAtom : String -> SExp
  BoolAtom : Bool -> SExp
  IntegerAtom : Integer -> SExp
  SymbolAtom : String -> SExp
```

An S-expression is either an atom of string, boolean, integer, or symbol (such as :example-symbol), or a list of S-expressions. For example, the S-expression '("q") will be represented in Idris as SExpList [StringAtom "q"].

Going back to the serialization problem, for the argument and return types or our Elab editor actions, we should specify how they should be converted to SExps. Defining certain functions for a given type sounds like the perfect task for Idris interfaces.

Therefore, we define the interface Editorable in Idris, which tells us how to serialize a given type into an SExp and how to deserialize it. In subsection 3.2.3, we will describe how implementations for the Editorable interface will be used by the compiler during the communication between the editor and the compiler.

FIGURE 3.8. Definition of the Editorable interface.

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

We have two functions fromEditor and toEditor that must be defined for a type that we want to make serializable. fromEditor allows us to deserialize a given S-expression into a value of the type a. toEditor allows us to serialize a value of the type a into an S-expression. Notice that both of these functions return a value in the Elab monad. There are a few reasons for this:

(1) The `Elab` monad captures failure. The `fromEditor` function, which parses S-expressions, should fail if we are given an S-expression that is ill-formed for the type `a`.

(2) The `Elab` monad has an implementation for the `Alternative` interface, which allows us to recover from failure if we need to.

(3) When an `Elab` action fails with an explicit use of `fail : List ErrorReporPart -> Elab a`, the user can give a detailed account of why it failed through the pretty printed errors. This is useful for giving graceful error messages to the user when a custom editor action fails.

(4) Users may want to limit both serialization and deserialization to some of the constructors of a type. Being able to fail for certain constructors gives the users flexibility.

(5) The reason that we specifically need `Elab` instead of any other monad that can fail, is that users may want to type check or normalize terms, or look up information about existing types and functions during serialization and deserialization. This is only available in the `Elab` monad.

3.2.1. `Editorable` implementations in Idris. Now that we have justified why putting `fromEditor` and `toEditor` in the `Elab` monad is necessary, let's see how we can define an `Editorable` implementation for `String`.

FIGURE 3.9. `Editorable` implementation for the type `String`.

```idris
implementation Editorable String where
  fromEditor (StringAtom s) = pure s
  fromEditor x =
    fail [ TextPart "Can't parse the"
         , NamePart `{SExp}
         , TextPart (show x ++ "as a")
         , NamePart `{{String}} ]
  toEditor = pure . StringAtom
```

The definition of `fromEditor` above tells us that if an S-expression is a string atom, then we know how to get a `String` from it, otherwise we fail. On the other hand, `toEditor` never fails; if we have a `String` we can always construct a `StringAtom` and put it in the `Elab` monad.[3]

Now, for a more complex example, let's look at Figure 3.10 to see we can make `List`s `Editorable`.

FIGURE 3.10. `Editorable` implementation for the `List` type.

```
——— Idris ———
implementation Editorable a => Editorable (List a) where
  fromEditor (SExpList xs) = traverse fromEditor xs
  fromEditor x =
    fail [ TextPart "Can't parse the"
         , NamePart `{SExp}
         , TextPart (show x ++ "as a")
         , NamePart `{List} ]
  toEditor xs = SExpList <$> traverse toEditor xs
```

The definition of `fromEditor` for lists tells us that if we have a list S-expression, which holds an Idris list `xs` of S-expressions inside, we can apply `fromEditor` to each of the S-expressions in `xs` and deserialize them all, and if they all succeed we can make a list from their results and return that. Similarly, the definition of `toEditor` for lists tells us that if we have an actual list `xs`, we can `traverse` the list to serialize them all, and if they all succeed, we can make an S-expression out of the serialized elements.

Now, let's look Figure 3.11 to see how we can serialize and deserialize one of the types that we expect to use the most in our editor actions, the type of names, `TTName`.

---

[3] Remember that `Editorable` is a `Monad`, which means it also has an `Applicative` implementation, therefore we can use `pure` instead of `return`, and this is the preferred way in Idris.

FIGURE 3.11. `Editorable` implementation for the type `TTName`.

```idris
implementation Editorable TTName where
  fromEditor (StringAtom s) = namify s
    where
      namify : String -> Elab TTName
      namify s =
        case reverse (map pack (splitOn '.' (unpack s))) of
          [] => fail [TextPart "Empty string can't be a TTName"]
          [x] => pure (UN x)
          (x :: xs) => pure (NS (UN x) xs)
  fromEditor x =
    fail [ TextPart "Can't parse the"
         , NamePart `{SExp}
         , TextPart (show x ++ "as a")
         , NamePart `{TTName} ]

  toEditor n = StringAtom <$> stringify n
    where
      stringify : TTName -> Elab String
      stringify (UN x) = pure x
      stringify (NS x []) = stringify x
      stringify (NS x xs) =
        pure (concat (intersperse "." (reverse ("" :: xs)))
             ++ !(stringify x))
      stringify (MN i x) = pure ("__" ++ x ++ show i)
      stringify n'@(SN sn) =
        fail [ TextPart "Don't know how to make"
             , NamePart n', TextPart "into StringAtom"]
```

This is a long code excerpt, and you do not have to follow the code entirely. We will summarize the code below, but this excerpt should demonstrate that we can define one of the most pivotal parts of serialization in Idris itself, without writing any Haskell code.

Here is an overview of what the code in Figure 3.11 does: when we want to deserialize a name, we require it to be a `StringAtom`, and then we split the string `s` on the dots it contains. If there is no dot inside, we make it a `TTName` without a namespace using `UN`, if there are dots inside, we make it a `TTName` with a namespace using `NS`. Similarly, if we

have a `TTName`, we check if we are given a namespace by looking at the constructors of `TTName`. If we are, then we make a `String` the name with dots in between, if we are not, then we merely make `String` from what we have. For the machine generated names, for now, we make up a representation, that we hope will not clash will other names. We fail for the special names. Editor actions will ideally not use names with MN and SN, so this is a temporary fallback solution[4].

Before we move on to the primitive `Editorable` implementations, we should point out that we have not made any function type `Editorable`. One can easily come up with a way to serialize pure and total functions with finite domains by listing the results of the function for each of the inputs. Alternatively, since editor actions are run interactively, we do have access to the source code of functions, so one could explore serialization of functions more seriously, but we will not entertain this thought in this thesis. Remember that there are no `Show` implementations for functions either, for the same reason.

3.2.2. Primitive `Editorable` implementations. The existing elaborator reflection system is enough to define `Editorable` implementations for some of the most common types in Idris. However, communication of Idris terms between the editor and the compiler is not straightforward. The main reason for that there is a colossal gap between the type of terms that our custom editor actions use, and the Idris terms we write in our editor. This gap is because Idris terms are written using the high-level surface language, and `Elab` actions deal with the core language terms. Therefore when we send the code for a term from the editor to the compiler, we will send a code in the surface syntax. Similarly, when the editor receives a piece of code from the compiler, that also has to be in the surface syntax. This is because when the editor takes some code from the file, or puts some code back to the file, it does not know how to turn that into a core language term. This conversion, also called elaboration as explained in section 2.2, can only be done in the Idris compiler.

---

[4] We can `fail` on them too if we wanted to. The reason we choose not to here is that many tactics use `gensym`, which gives back a MN name.

If the editor can only deal with surface-syntax terms, and the `Elab` actions can only work with core language terms, the system we are designing must take care of the conversion between the two. Specifically, the `Editorable` interface that we defined above can be made responsible for this.

When the S-expression received by the compiler contains a string that is supposed to be a piece of Idris code, `fromEditor` should parse the string into a surface-syntax code, and then elaborate that into a core language term. Only after that can we run the `Elab` editor action that we want to execute.

Similarly, when we finish running the `Elab` action, `toEditor` should delaborate[5] the core language code to surface-syntax code, and then it should pretty print it as a string. The resulting string can be sent back from the compiler to the editor in an S-expression.

In subsection 2.2.1, we talked about the core language types that are reflected in Idris. We have already seen the definitions of the types `TT` and `TTName`, and we have touched upon `TyDecl`, `DataDefn`, `FunDefn`, and `FunClause`. We will not concern ourselves with `Raw` right now, since solving the problem for typed core language terms suffices to solve the same problem with untyped one; we can always type-check a `Raw` term into a `TT`, or forget the type of a `TT` term into a `Raw` one.

Notice that we have already given an `Editorable` implementation of `TTName` above, since names in the surface syntax and core language are the same. Elaboration can take care of namespace resolution, but that can be achieved with `Elab` tactics as well.

However, for the other core language types, we should define `Editorable` implementations. As we discussed above, the editor can only send and receive pieces of code in the surface syntax, which means we have to do parsing, elaboration, delaboration and pretty printing in our `Editorable` implementation. It is not possible to write this implementation in Idris, because Idris does not reflect the surface syntax, or parsing or pretty printing with it. Therefore, we will hard-code the implementations of `TT`, `TyDecl`,

---

[5] *Delaboration* is the name in the Idris compiler for converting core language code to surface syntax code. It is meant to be the opposite of elaboration.

`DataDefn`, `FunDefn`, and `FunClause` into the compiler, which allows us to do *direct reflection* [**?**] by making use of the already existing compiler implementations of the steps we listed above.

To achieve this, we will have to extend the existing `Elab` monad with primitives that go through the steps we mentioned above. We will have to define two primitives for each other those types, one for `fromEditor` and one for `toEditor`. Instead of adding two primitives for each of those types, we can only add two primitives and make them polymorphic, and add a constraint on what types it can be used for. This would save us from adding new primitives to `Elab` every time we want to add a primitive `Editorable` implementation.

What we mean by a constraint here is a predicate on types, i.e. a `Type`-indexed type family that describes which types can be used in the primitives for `fromEditor` and `toEditor` we are about to define.

FIGURE 3.12. Definition of the `HasEditorPrim` predicate in Idris.

```Idris
data HasEditorPrim : Type -> Type where
  HasTT        : HasEditorPrim TT
  HasTyDecl    : HasEditorPrim TyDecl
  HasDataDefn  : HasEditorPrim DataDefn
  HasFunDefn   : HasEditorPrim (FunDefn TT)
  HasFunClause : HasEditorPrim (FunClause TT)
```

Now we can use the predicate `HasEditorPrim` to define our two new `Elab` primitives:

FIGURE 3.13. New `Elab` primitives for serialization and deserialization that depend on `HasEditorPrim`.

```Idris
prim__fromEditor : {auto has : HasEditorPrim a} -> SExp -> Elab a
prim__toEditor : {auto has : HasEditorPrim a} -> a -> Elab SExp
```

Notice that their first argument is the predicate `HasEditorPrim` applied to the type `a`, and it is an implicit argument that Idris should try to automatically solve. This saves us from having to pass around a constructor every time we want to use the primitive.

Using these two primitives, the `Editorable` implementation definitions for the core language types all look alike:

FIGURE 3.14. `Editorable` implementation for `TT`, that depends on the new `Elab` primitives.

```
─────────────────────────────── Idris ───────────────────────────────
implementation Editorable TT where
  fromEditor x = prim__fromEditor x
  toEditor x = prim__toEditor x
```

We now know what primitive we want to have and we know what we want it to do. We will discuss the implementation of the `Elab` actinos for `fromEditor` and `toEditor` in the compiler in section 4.3.

3.2.3. Using `Editorable` in the compiler and the `%editor` modifier. The motivation behind the `Editorable` interface is twofold:

(1) to check whether a given `Elab` action is suitable to be used as an editor action.

(2) to use the `fromEditor` and `toEditor` definitions to serialize and deserialize data when the action is run.

The first motivation means all argument and return types should have an `Editorable` implementation. If we want to write a toy editor action `toy : TTName -> TT -> Elab TT`, then the compiler must check if `TTName`, `TT` (from the second argument) and `TT` (from the return type) are all `Editorable`. If any of the argument of return types do not have an `Editorable` implementation, then we should not be able to run the given action from the editor. In fact, to make the distinction clearer, we add a new function modifier syntax `%editor` to Idris, that checks it during type-checking. We explain its implementation in section 4.6.

FIGURE 3.15. Type declaration for the `toy` editor action we want to define.

```
──────────────────────────── Idris ────────────────────────────
%editor
toy : TTName -> TT -> Elab TT
```

When used before the type declaration of a function, Idris makes sure during type-checking that this function's argument and return types all have an `Editorable` implementation. If they do not, then the user gets a type-error that tells them that since they declared that `Elab` action an editor action, they must provide an `Editorable` implementation for all argument and return types, and the error tells them which one is missing.

The second motivation for `Editorable` allows the compiler to communicate with the editor via S-expressions. When the editor sends an S-expression like `((:elab-edit "toy" '("q" "S Z") 35 5) 20)` to the compiler with the intent of running an `Elab` action, the following steps should happen:

(1) The S-expression we are given contains the name of the `Elab` action, namely `toy`. We must lookup the name in order to resolve its namespace and also learn the type of the action.

(2) We check if `toy` has been marked as an editor action using `%editor`. If not, we fail the editor action.

(3) We "collect" the components of the type of the `Elab` action. For `toy`, from the type signature `TTName -> TT -> Elab TT` we obtain the component list with three members: `TTName`, `TT`, and `Elab TT`. Since the last one is the return type for the action, we check the number of arguments we are given by the S-expression, and `'("q" "S Z")` has two elements, has exactly one fewer element than the component list we collected above.

(4) Observe that the argument members of the component list match the arguments list S-expression one to one. The argument components for `toy` are `TTName` and `TT`, and we are given the S-expressions `"q"` and `"S Z"`. If we zip these lists, we get pairs of S-expressions and what type they will have when

they are parsed. Then, Idris parses them using the `fromEditor` function for their respective `Editorable` implementations.

(5) Then we will have a list of terms in the compiler that correspond to a `TTName` and a `TT`. All we have done so far was to check if `toy` was a valid editor action, and then to parse the inputs just so that we can run `toy`. The next step is to run `toy`.

(6) When we are done with that, we know we will get a term of the type `TT`, because `toy`'s type signature ends with `Elab TT`. That means we can use `toEditor` from the `Editorable` implementation of `TT`, which gives us an S-expression.

(7) We can now send that S-expression back to the editor.

3.3. Examples in action

3.3.1. Successful example. We now know how the compiler should work with the `Editorable` interface. To illustrate how it works in reality, let's finish the `toy` editor action we started above.

FIGURE 3.16. Implementation of the `toy` action in Idris.

```
――――――――――――――――――――――― Idris ―――――――――――――――――――――――
%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!"]
```

Here is what we expect from this `Elab` action: we will pass it two arguments: the first one is a name of a hole, and the second one is a core language term. If the type of the hole is `Nat`, then we will return the term that we are given, otherwise we will fail.

To be able to call this editor action from Emacs, we will have to write a bit of Emacs Lisp:

FIGURE 3.17. Necessary Emacs Lisp code to run the toy action.

```
──── Emacs Lisp ────
(defun idris-elab-hole-arg (action args)
  "Run Elab action in editor, replace the hole with the result"
  (interactive)
  (let ((result (car
    (idris-eval
      `(:elab-edit ,action ,args
                   ,(idris-get-line-num) ,(current-column))))))
    (save-excursion
      (apply 'delete-region (idris-hole-start-end))
      (insert result))))

(defun idris-toy ()
  "Run the toy Elab action in editor"
  (interactive)
  (let ((term (read-string "Enter a term:")))
    (idris-elab-hole-arg "toy" `(,(idris-name-at-point) ,term))))
```

The first function, idris-elab-hole-arg, is a general function that can be reused by other hole-based editor actions. It takes a name of an action and a list, and sends them to the Idris executable, also adding the line and column numbers. When it gets a response, it deletes the hole that the cursor is on, and replaces it with the string/code that was just received from the compiler.

The second function, idris-toy, is the little piece of code that runs the toy editor action we defined above. It asks the user to enter some code, and when they do, it uses idris-elab-hole-arg to send to Idris a message that consists of the name under the cursor and that term the user just typed in. Notice that the name under the cursor will be parsed as a TTName, type of the first argument of toy, and the term we typed in will correspond to TT, type of the second argument of toy.

Suppose we want to use this new editor action now.

FIGURE 3.18. Example `Nat` declaration to run `toy` on.

```
─ Idris ─────────────────────────────
n : Nat
n = ?q
```

We can place the cursor on ?q and then run the `idris-toy` function in Emacs, possibly through a keyboard shortcut we assign to it. By definition of `idris-toy`, Emacs asks us to enter a term, suppose we write S Z and press enter. The following communication happens in the background with the compiler:

FIGURE 3.19. Communication to run `toy` on the hole ?q.

```
─ S-expression ─────────────────────
-> ((:elab-edit "toy" ("q" "S Z") 35 5) 20)
<- (:return (:ok "1" nil) 20)
```

We send the `Elab` action name `toy` and the list of arguments as an S-expression. Idris sends back the result of the `Elab` action when run with the arguments ?q as the name, and the core language term for S Z, which is `Prelude.Nat.S Prelude.Nat.Z`. Since the type of our hole is `Nat` here, `toy` just returns the same term we give it, but it pretty prints it as 1.

By the definition of `idris-elab-hole-arg`, the result 1 replaces the hole ?q in the file. The resulting code is this:

FIGURE 3.20. End result of running `toy` on the hole ?q.

```
─ Idris ─────────────────────────────
n : Nat
n = 1
```

3.3.2. What `Editorable` precludes. Suppose we wanted to write a polymorphic editor action, that merely returned its argument back.

FIGURE 3.21. A polymorphic editor action `poly` in Idris that is supposed to fail.

```Idris
%editor
poly : a -> Elab a
poly x = pure x
```

First of all, observe that the type declaration above actually is `{a : Type} -> a -> Elab a`, which means there is an implicit argument in the beginning that has the type `Type`.[6]

The problem with this is that, the editor has to tell the compiler what type of result we want to get at the end. We do not yet have a way to communicate a value of the type `Type` between the editor and the compiler, i.e. we cannot define an `Editorable` implementation for `Type` in Idris itself, because Idris does not allow us to pattern match on types, since that breaks parametricity and creates problems with type erasure [15]. We will not explore adding such an `Editorable` implementation as a primitive. We will instead disallow polymorphic editor actions.[7] When we try to compile `poly`, we get the following error:

FIGURE 3.22. Error message that shows why `poly` fails.

```Idris error message
You declared poly to be an editor action, but there's no
Language.Reflection.Editor.Editorable implementation for Type
```

Here is another custom editor action we want to preclude:

---

[6] `Type` is the type of types in Idris, similar to `*` in Haskell and `Set` in Agda, but with a caveat: Haskell's `*` has the type `*` [25], while Agda has universe polymorphism and Idris has cumulativity. This means Agda and Idris manage to avoid Girard's paradox, but Haskell does not even attempt that, "... not that there's anything wrong with that." [13]

[7] We leave polymorphic editor actions to future work. We argue that monomorphic editor actions suffice in almost all use cases in the editor.

FIGURE 3.23. A higher-order editor action `funAction` in Idris that is supposed to fail.

```
———————————— Idris ————————————
%editor
funAction : (Nat -> Nat) -> Elab ()
funAction x = pure ()
```

We do not have a way to serialize and deserialize functions of the type `Nat -> Nat`, so therefore we should not allow `funAction` to be declared an editor action.[8] When we try to compile `funAction`, we get the following error:

FIGURE 3.24. Error message that shows why `funAction` fails.

```
————————— Idris error message —————————
You declared funAction to be an editor action, but there's no
Language.Reflection.Editor.Editorable implementation for Nat -> Nat
```

We have now seen the motivations behind the `Editorable` interface and why we defined it this way. We have also demonstrated that this definition is useful for serialization and deserialization of the data used in editor actions, and that for the unserializable kinds of data, such as functions, `Editorable` acts as a gatekeeper. In the next chapter, we will look at how this design is implemented in the Idris compiler in Haskell.

---

[8] Remember that we cannot inspect a function's body; we can only apply arguments to a function.

CHAPTER 4

Implementation

4.1. Additions to the Idris standard library

Out of the types we skimmed through in subsection 2.2.1, `TT`, `Raw`, and `TTName` reside in the `Language.Reflection` module, and `TyDecl`, `DataDefn`, and `FunDefn` live in `Language.Reflection.Elab`. This is because Idris only has quotation of terms, as we reviewed in subsection 2.2.2, so the definitions we will need in quotations are in `Language.Reflection`. The ones that are not needed by quotation but are needed for elaborator reflection are in `Language.Reflection.Elab`.

Out of the types we defined in section 3.2, `SExp` should live in `Language.Reflection.Elab` since `Elab` will depend on it. However, the new interface `Editorable` should live in `Language.Reflection.Editor`.

Remember from Figure 3.13 that we added two new `Elab` primitives to our language, namely `prim__fromEditor` and `prim__toEditor`. The way `Elab` primitives work in Idris is that there is a constructor in the `Elab` data type for each of them. Therefore we need to add those constructors and then define `prim__fromEditor` and `prim__toEditor` in terms of them.

FIGURE 4.1. New constructors for the new `Elab` primitives, and function definitions based on them.

```idris
export
data Elab : Type -> Type where
  -- the constructors we had before
  Prim__FromEditor : {a : Type} -> HasEditorPrim a -> SExp -> Elab a
  Prim__ToEditor : {a : Type} -> HasEditorPrim a -> a -> Elab SExp

export
prim__fromEditor : {auto has : HasEditorPrim a} -> SExp -> Elab a
prim__fromEditor {has = x} sexp = Prim__FromEditor x sexp

export
prim__toEditor : {auto has : HasEditorPrim a} -> a -> Elab SExp
prim__toEditor {has = x} y = Prim__ToEditor x y
```

We make the additions in Figure 4.1 in `Language.Reflection.Elab`. Notice that **export** for `Elab` does not export the constructors for the module, and we do not want to export all the primitive actions without any limits; we want to make `Elab` more abstract. For that purpose, we define `prim__fromEditor` and `prim__toEditor` that we actually can export. Compared to the types of the constructors we defined, types of the functions we defined are made more user-friendly by making the `HasEditorPrim` argument automatically inferrable by proof search, which is a helpful feature of Idris usable via the **auto** keyword.

We will inspect in how these new constructors will be used in the compiler, but first we have to clarify some terminology.

4.2. Reflection and reification in the compiler

Before we delve into further details of the implementation, we should compare and contrast certain terminology we will use in this chapter. The graph in Figure 4.2 describes the relationship between the different kinds of languages and representations and the spells out the specific names for moving from one to another.

FIGURE 4.2. The relationship between reflection, reification, quotation, unquotation, elaboration and delaboration.

Haskell terms

such as the `()` term

`()`

reification

Haskell
representation of
Idris core language
terms

such as the `Term`[1] term
`P (DCon 0 0 False) (UN`
`"MkUnit") (P (TCon 0 0`
`False) (UN "Unit")`
`Erased)`

reflection

internally represented as

correspondence

delaboration

Idris terms

such as the `()` term

`()`

unquotation

Idris core terms

such as the `Unit` term

`MkUnit`

Idris representation
of the Haskell
representation of
Idris core language
terms

such as the `TT` term
`P (DCon 0 0) (UN`
`"MkUnit")`
`(P (TCon 0 0) (UN`
`"Unit") Erased)`

elaboration

quotation

In the previous sections, we have informally used the word "reflection" to refer to the act of providing an Idris representation for a Haskell type we use in the compiler. That was handwavy but still accurate: In the graph we defined reflection as moving from a Haskell term to the Haskell representation of an Idris term. Since this holds for any Haskell term, you can also take a Haskell term that is itself an internal Haskell representation of some Idris syntax. When you reflect that, you get the internal representation of an Idris representation of the Haskell representation of an Idris term. Our previous usage of "reflection" was only a special case of the word, but in this section, we will use the generalized definition.[2]

---

[1] In the compiler source code in Haskell, both `Term` and `Type` are aliases for `TT Name`. This should not be confused with the type of reflected names, `TTName`, defined in Idris. In the compiler, the type of typed core language terms, `TT`, is indexed by the type of names in that term. The type of names in the compiler is called `Name`, therefore `TT Name` in the compiler is the type of core language terms with the usual names. We will stick with `Term` and `Type` in this thesis to avoid this confusion.

[2] We should clarify that there is no infinite regress here. When we quote an Idris term, we still get an Idris term, of the type `TT`, which then can be quoted again and again if necessary. However, all of these are

Let's give an example and see what we mean by this. We defined the Idris type SExp in Figure 3.7. When we will define fromEditor and toEditor in section 4.3, we will need to convert Haskell terms of the Haskell type SExp into the internal representation of an Idris term of the Idris type SExp, and vice versa.

The conversion from the former to the latter is reflection. We write the function reflectSExp in Figure 4.3.

FIGURE 4.3. The function to reflect Haskell terms of the type SExp to the internal representations of Idris terms of the Idris type SExp.

```
                              ─── Haskell ───
reflectSExp :: SExp -> Raw
reflectSExp (StringAtom s) =
  RApp (Var (tacN "StringAtom")) (RConstant (Str s))
reflectSExp (SymbolAtom s) =
  RApp (Var (tacN "SymbolAtom")) (RConstant (Str s))
reflectSExp (BoolAtom b) =
  RApp (Var (tacN "BoolAtom")) (reflectBool b)
reflectSExp (IntegerAtom i) =
  RApp (Var (tacN "IntegerAtom")) (RConstant (BI i))
reflectSExp (SexpList l) =
  RApp (Var (tacN "SExpList"))
       (reflectList (Var (tacN "SExp")) (map reflectSExp l))
```

Observe that in the reflectSExp, we are returning a Raw term, a core language term that is untyped. This mainly for convenience: it is easier to write the syntax trees for untyped terms. We can always type-check them later.

To get a Haskell term of the type SExp from an internal representation of an Idris term that has the Idris type SExp, we have to write a function reifySExp that reifies a given Term, which you can see in Figure 4.4:

---

still representable in Idris. Similarly, a Haskell term can be reflected again and again, and the Term type is enough to express this.

FIGURE 4.4. The function to reify the internal representations of Idris terms of the Idris type SExp to Haskell terms of the type SExp.

```Haskell
reifySExp :: Term -> ElabD SExp
reifySExp (App _ (P _ n _) x)
  | n == tacN "StringAtom"  = StringAtom  <$> reifyString x
  | n == tacN "SymbolAtom"  = SymbolAtom  <$> reifyString x
  | n == tacN "BoolAtom"    = BoolAtom    <$> reifyBool x
  | n == tacN "IntegerAtom" = IntegerAtom <$> reifyInteger x
  | n == tacN "SExpList"    = SexpList    <$> reifyList reifySExp x
reifySExp tm = fail ("Not an SExp: " ++ show tm)
```

Observe in `reifySExp` that we are returning `ElabD SExp`.[3] This is because reification, unlike reflection, can fail. This function is designed to reify the internal representation of S-expressions. If the `Term` it receives does not correspond to one, reification should fail.

Now that we know what reflection and reification mean in the context of compiler development, we can move on to how they are used in implementing primitive implementations for `Editorable`.

4.3. Primitive `Editorable` implementations

In subsection 3.2.2 we discussed why core language types like `TT`, `TyDecl`, `DataDefn`, `FunDefn`, and `FunClause` must have primitive implementations of the `Editorable` interface. Recall that for each of these types, the editor can only send a piece of code to the compiler as a string which contains code in the surface language, however these types are in the core language. There are many steps in between that we are missing, such as parsing, elaboration, delaboration and pretty printing.

Now let's see how we implemented this in the Idris compiler. Tactic evaluation in the compiler happens in `runElabAction`, which is defined in the compiler source code. Inside that, there is a helper function `runTacTm` that takes a typed term corresponding

---

[3] `ElabD` is an alias for the `Elab'` monad combined with some special state, this does not concern us in this thesis.

to a value in the `Elab` monad of Idris, which should correspond to a `Elab` primitive constructor applied to arguments, i.e. when the `Elab` action term is in a normal form. Then we can check which primitive constructor we are dealing with and behave accordingly.

FIGURE 4.5. The `runElabAction` function in the compiler, how `Elab` action terms are run under the hood.

```haskell
runElabAction :: ElabInfo -> IState -> FC -> Env
              -> Term -> [String] -> ElabD Term
runElabAction info ist fc env tm ns = do tm' <- eval tm
                                         runTacTm tm'

  -- some helper functions ...

  runTacTm :: Term -> ElabD Term
  runTacTm tac@(unApply -> (P _ n _, args))
    | n == tacN ”Prim__Solve”
    = do ~[] <- tacTmArgs 0 tac args
         solve
         returnUnit
    -- other cases for other constructors
```

In Figure 4.6 we can observe how `runElabAction` is defined in the compiler. It takes many arguments, such as information for the elaborator, the internal state of the compiler at that point, the source location of the action, the environment under which the `Elab` term should be evaluated to a normal form, the term that represents the `Elab` action term, and a namespace. For our purposes, we do not have to worry about all of these.

In the definition of `runElabAction` we evaluate `tm` to a normal form of the `Elab` action, and then we check in `runTacTm` which form we want run.

If we have a non-neutral normal form of the `Elab` action, that means when we decide that what we have is a global variable term, which is a constructor, represented by `P`, or it is a series of applications to a variable term that is a constructor. We can handle both situations with the `unApply :: Term -> (Term, [Term])` function, which dissects

multiple curried function applications into the the term that should be a function and the list of argument terms passed to it.[4]

Figure 4.6 demonstrates how an example `Elab` action term is treated by `runTacTm`. For the primitive version of the `solve` action, which is represented by the `Prim__Solve` constructor in `Elab`, we first make sure that there are no given arguments, and then run `solve :: Elab' aux ()` in the internal elaborator monad. If these all succeed, we return the reflected unit term back.

Now, in order to add more `Elab` primitives, we have to implement similar cases for `Prim__FromEditor` and `Prim__ToEditor` in `runTacTm`, which we started to do in Figure 4.6.

---

[4] A term that is not a function application will be treated as a function application with no arguments passed.

FIGURE 4.6. Adding cases for `Prim__FromEditor` and `Prim__ToEditor` in `runTacTm` in the compiler.

```Haskell
runTacTm :: Term -> ElabD Term
runTacTm tac@(unApply -> (P _ n _, args))
  -- other cases for other Elab primitives
  | n == tacN "Prim__FromEditor"
  = do ~[ty, hasEditorPrim, arg] <- tacTmArgs 3 tac args
       ty' <- eval ty
       arg' <- eval arg
       case ty' of
         P _ tyN _ | tyN == reflm "TT" ->
           -- now we want to convert an S-expression to TT
         P _ tyN _ | tyN == tacN "TyDecl" ->
           -- now we want to convert an S-expression to TyDecl
         -- other cases for other core language types
  | n == tacN "Prim__ToEditor"
  = do ~[ty, hasEditorPrim, arg] <- tacTmArgs 3 tac args
       ty' <- eval ty
       arg' <- eval arg
       case ty' of
         P _ tyN _ | tyN == reflm "TT" ->
           -- now we want to convert a TT into an S-expression
         P _ tyN _ | tyN == tacN "TyDecl" ->
           -- now we want to convert a TyDecl into an S-expression
         -- other cases for other core language types
```

Observe that we left most parts in here blank, let's zoom in to the `TT` cases of `Prim__FromEditor` and `Prim__toEditor`. The actual code blocks for them refer to many helper functions that are difficult to follow; we will instead explain what they do in detailed prose. We will use the variable names in Figure 4.6, so it would be helpful to look back at the code excerpt when a name is unclear.

For the `TT` case of `Prim__FromEditor` that we left as a comment before, we have to do the following steps:

(1) We have an `Term` named `arg'` representing the S-expression passed to the `Prim__FromEditor`, we have to reify this using `reifySExp` and get a Haskell term with the Haskell type `SExp`.

(2) Remember from subsection 3.2.2 that we should receive a string atom S-expression. We fail if the `SExp` we get in the previous step is not a string atom. Otherwise we have access to the string `s`.

(3) We parse the string `s` into a abstract syntax tree term `pterm` of the surface syntax, which is represented by the type `PTerm` in the compiler. We therefore obtain the variable `pterm`.

(4) Elaboration cannot resolve namespaces in the parsed surface syntax terms. Using the current context, we write a function `resolveNames :: Context -> PTerm -> Either Err PTerm` that traverses the the surface language abstract syntax tree using Uniplate [42], and then at the `PRef` terms, i.e. variable references, we search the context to find the matching fully namespaced name. If there is a unique matching name, we change the name. If there are multiple matching names, we return an error that there is ambiguity in the name. If there are no finds, we leave it as is and let elaboration handle the rest. We apply the current context and `pterm` to this function and obtain `pterm'`, the surface syntax term with resolved names.[5]

(5) Elaborate `pterm'` into the core language and get `t`.

(6) Now, remember that our function `fromEditor` must return `Elab TT` in this context. Therefore in the Haskell implementation, we want to return the Haskell representation of the Idris representation of the Haskell representation of the given code. In other words, we have to reflect `t`, which is a `Term`, and then get a `Raw` term, which is a Haskell representation of the Idris representation of `t`. We can call the reflected term `reflected`.

(7) `runTacTm` requires us to return a `Term`, therefore we type-check `reflected` and get a Haskell term of the type `Term`, which we can call `tmReflected`.

(8) When we return a core syntax tree, we want to return it in normal form, therefore we normalise `tmReflected` and return it.

---

[5] We can do better when there is a known context and type, e.g. at a hole. Then we could elaborate in a local synthesized context to use types for overloading. This is a possible direction for future work.

For the `TT` case of `Prim__ToEditor` that we left as a comment before, we have to do the following steps:

(1) We have an Idris term `arg'` representing the `TT` term passed to the `Prim__ToEditor`, we have to reify this using `reifyTT` and get a Haskell term with the Haskell type `Term`. In other words, we are given the Haskell representation of the Idris representation of the Haskell representation of a term. We want to get the Haskell representation of that term, using reification.

(2) We now have a `Term` term named `v`. We delaborate and resugar `v` and get the surface syntax version of it. We name the delaborated version `pterm`, which has the type `PTerm`.

(3) We pretty print `pterm` and get a string `s`.

(4) With `s`, we can create `StringAtom s`, which is a Haskell term with the Haskell type `SExp`. Using `reflectSExp`, we can get the Haskell representation of an Idris term that has the Idris type `SExp`, which we call `tm`.

(5) When we return a core syntax tree, we want to return it in normal form, therefore we normalise `tm` and return it.

The primitive implementations for the other types, namely `TyDecl`, `DataDefn`, `FunDefn`, and `FunClause`, are not significantly different from `TT`. Hence we will not go over them.

In the next section, we will see how we will deal with the issue of local contexts when we elaborate terms we receive from the editor.

4.4. Extensions to elaboration and the internal Idris state

In subsection 3.2.3, we talked about how we wanted to deserialize S-expressions into the arguments to the `Elab` action we want to run. As we seen in the previous section, in order to deserialize an S-expression into a term of the type `TT`, we have to parse, elaborate and reflect. During elaboration, we cannot have any unbound variables, otherwise we get an error.

Nevertheless, the editor can send to the compiler code pieces that contain local variables. Handling the global context is easy, but when it comes to a local context, e.g. if the code snippet we received uses a variable whose binding is introduced by **let**, **case**, or a lambda elaboration is doomed to fail. We need to find a way to get the local context at a given position, and then pass it to elaboration.

If we restricted `Elab` editor actions to only holes, this would not have been a problem, since holes already provide a way to query the local context at them. For other names and terms, however, we do not have a way.

To solve this problem, we extended the elaboration state with an interval map in which keys are source locations, i.e. a pair of line and column numbers, and values are local contexts, i.e. environments. We use the finger tree implementation of interval maps in Haskell [31].

For those who are unfamiliar, an interval map is a data structure that maps intervals to values. Every entry consists of the interval between two keys, and a value associated with the interval. One can query the map with a single key, and get the values in the map which are mapped by the intervals that the key is in.

During elaboration, we still have access to the source locations of all terms. Therefore we can register the interval between the start and end locations of the term in the source code, and then map it to the local context. Following the terminology we used in the compiler, we will call this map the **source map**.

This task required significant refactoring, since the source map needs to be preserved during elaboration of different parts, and also after elaboration. Elaboration itself cannot change the internal Idris state `IState`; however, it has its own state `ElabState`. When elaboration finishes, we then update `IState` with the additions.

In the next section, we will see how exactly `fromEditor` and `toEditor` are used in the IDE mode of the compiler to communicate with the editor.

4.5. Extensions to the Idris IDE mode

Idris' IDE mode has an S-expression parser, which looks at the shape of an S-expression and creates a IDE mode command value based on it. We extend this parser by adding a line that says that if we see a symbol atom `:elab-edit`, a string atom for the `Elab` action name, an S-expression list, and two integer atoms for line and column numbers (in that order), then we add a new IDE mode command in the compiler.

These commands are passed around for a while in the compiler, however eventually there are separate functions that deal with each editor command. We write the function for edit-time tactics under the name `elabEditAt` in Figure 4.7.

FIGURE 4.7. The function `elabEditAt` that runs editor actions in the compiler.

```haskell
elabEditAt
  :: FilePath    -- ^ The file name in which the Elab action is run
  -> String      -- ^ The name of the action
  -> (Int, Int)  -- ^ The line and column number the action is run on
  -> [SExp]      -- ^ The arguments to the action
  -> Idris ()
elabEditAt filename nameStr pos args = ...
```

Once again the actual code block for `elabEditAt` refers to many helper functions that are difficult to follow; we will instead explain what it does in detailed prose. This will read like a much more detailed version of our description in subsection 3.2.3 about how `Editorable` works in the compiler.

Here are the steps `elabEditAt` takes when it is called:

(1) The variable `nameStr`, which has the Haskell type `String`, holds the name of the `Elab` action we want to run. However, we need to parse this string into a `Name`, in order to get any namespace information that is provided inside `nameStr`. If there is any namespace information, we use `NS`, otherwise we use `UN`. We call the result `name`, which has the type `Name`.

(2) Now we have `name`, yet we do not know whether it really refers to a existing `Elab` action. If it does what is its type? We consult the context and find out its type, which we can call `ty`, which has the Haskell type `Type`. From the context, we also resolve the name of the `Elab` action if we need to, we call the resolved name `ns`.

(3) Using the resolved name `ns`, we look up the flags declared for this function. We check if the it has been declared an editor action using the **%editor** modifier. If not, we do not proceed.

(4) We are given a list of `SExp`s, which is the list that the editor passes to the compiler to apply the editor action they want to call. However, we have to check whether this list contains the right number of arguments, and for that we need to know how many arguments the editor action should have. For that, we can write a function `collectTypes` that takes a the `Type` `ty` and gives us a list of `Type`s, which are the components of `ty`, i.e. curried arguments and also the return type.

FIGURE 4.8. The function `collectTypes` to dissect a type signature to its components.

──────── Haskell ────────

```
collectTypes :: Type -> [Type]
collectTypes (Bind _ (Pi _ _ t1 _) t2) = t1 : collectTypes t2
collectTypes t = [t]
```

We apply `ty` to `collectTypes` and name the result `collected`, which is a list of `Type`s.[6]

(5) We check if the length of `collected` is exactly one more than the length of `args`, because the last element in `collected` is the return type of the `Elab` action.

---

[6] Observe that this throws away the binding structure of the Π-type, which mean **%editor** actions cannot have dependent types. As we mentioned before about polymorphic editor actions, we believe monomorphic non-dependently-typed editor actions suffice for almost all editor actions. We leave further exploration for future work.

(6) Using the interval map we defined in section 4.4 and the line and column number the cursor in on, which are given to `elabEditAt` as arguments, we query the local context at the cursor location, which we call `env`.

(7) We zip `collected` and `args` and get a list of pairs of `Type` and `SExp`. For each of the pairs, we have to do the following and get a result for each pair of (`argTy`, `sexp`):

    (a) We construct an Haskell representation of an Idris term that calls `fromEditor`. Remember that `fromEditor` is polymorphic and also has an interface constraint; therefore, constructing this as a `Term` is not trivial. As a shortcut, we can construct a `PTerm` that explicitly applies the polymorphic type, which will be `argTy` in this case.

    (b) We elaborate the `PTerm` we construct above, which gives us `tm`, the Haskell representation of an Idris term that has the type represented by `argTy`, but in the Idris `Elab` monad.

    (c) Since `tm` is the Haskell representation of an `Elab` action, we should run it using `runElabAction`, under the local context `env`. This gives us `tm'`, the Haskell representation of a term that has the type represented by `argTy`.

    (d) We return `tm'` as the result for the pair (`argTy`, `sexp`).

(8) We make a list of all the `tm'`s returned for each pair.

(9) Remember that this list consists of the results parsing each S-expression into what type they represent. Therefore, its elements are actually Haskell representations of argument for the `Elab` action we want to call. Hence, we construct a term by applying the elements of the list to the variable reference to `ns`, which we can call `app`.

(10) Now we have a Haskell representation of an `Elab` action, because we applied all the arguments required. From the last element of `collected`, which is supposed to be the return type, we can find out what type the term we expect when we run `app`. The last element of `collected` is a Haskell representation of

an Idris type in the shape of `Elab a`, for some `a`. We extract the Haskell representation for the Idris type `a` from it. We name the extracted representation `lastTyInElab`.

(11) Using `runElabAction` and `lastTyInElab`, we run `app`. This gives us `res`, the Haskell representation of a term that has the Idris type that `lastTyInElab` represents.

(12) We want to serialize this term before returning it as an S-expression. The only way we know how to serialize is through the `toEditor` function in `Editorable`, but remember that `toEditor` is defined in Idris. We thus construct a Haskell representation of an Idris term that calls `toEditor`, similar to how we constructed an application term for `fromEditor` before in item 4a. For convenience in implementation resolution, we first construct a `PTerm`, then elaborate it. We call the elaborated term `tm`.

(13) Since `tm` is the Haskell representation of an `Elab` action, we should run it using `runElabAction`. We get the Haskell representation of a term that has the Idris type `SExp`.

(14) We reify the Haskell representation of that term into a Haskell term that has the Haskell type `SExp`, using `reifySExp`, and get `resSExp`.

(15) We send a message to the client/editor and report that the action is successful, and the result is `resSExp`.

This concludes the IDE mode section of implementing edit-time tactics.

4.6. Extensions to the type-checker for the `%editor` modifier

The last remaining part of the implementation is an additional restriction to type-checking.

If a function is declared an editor action with the **`%editor`** modifier, we want to find out during type-checking if it is viable to use them as an editor action, and an `Elab` action is *viable* if the components of its type have implementations of the `Editorable`

interface.[7] We discussed this idea in the `toy` example in subsection 3.2.2, and also have seen the type errors you can get in subsection 3.3.2. Now let's see how we implement this feature.

To the `elabType'` function in the compiler, we add the following:

(1) We use `collectTypes` that we defined in Figure 4.8 to get the components of the type signature, we will call this list of `Type`s `collected`.

(2) The last element in `collected` is supposed to be return type. We extract `lastTyInElab`, which is the Haskell representation of the Idris type in the Idris `Elab` monad, the same way we did in item 10.

(3) We replace the last element in `collected` with `lastTyInElab`, and name the new list `toCheck`.

(4) For every element `ty` in `toCheck`, we do the following:

(a) We want to find out if there is an `Editorable` implementation for the Idris type `ty` represents. The easy way to do that is to construct a `PTerm` of a `fromEditor` application and then elaborate it, as we did in before, which should resolve implementation constraints, or give an error if it cannot.

(b) If there is no error, we move on to the next `ty` silently. If there is an error, that implies that there was no `Editorable` implementation for the Idris type `ty` represents. Then we send an error message like the ones in subsection 3.3.2 back to the editor.

This concludes the implementation of edit-time tactics in the Idris compiler. Remember that the part we described here is only for the compiler; it is just one step of how edit-time tactics work. For the communication between the editor and the compiler, we have to write some editor language code, i.e. Emacs Lisp, that would send and receive messages. We showed in Figure 3.17 what that would look like.

In the next chapter, we will see real applications of edit-time tactics.

---

[7] Encoding this as a universe could be a starting point for dependently-typed editor actions.

CHAPTER 5

Applications

5.1. A tactic to replace the built-in "add clause" action

We have seen in chapter 1 how the "Add initial match clause to type declaration" editor action works. When the cursor is on the type signature of a function that does not have any clauses, we can run this editor action and get an initial clause for the function.

In this section we implement this editor action for top-level type declarations without implicit arguments or interface constraints, using edit-time tactics. The Idris code we need to write is in Figure 5.1.

FIGURE 5.1. Implementation of the edit-time tactic for "add clause".

```idris
collectTypes : TT -> (List TT, TT)
collectTypes (Bind _ (Pi ty _) t) =
  let (xs, t') = collectTypes t in
  (ty :: xs, t')
collectTypes t = ([], t)

%editor
addClause : TTName -> Elab (FunClause TT)
addClause n =
  do (_, _, ty) <- lookupTyExact n
     ty' <- normalise !getEnv ty
     let (argTys, retTy) = collectTypes ty'
     argNames <- traverse (const fresh) argTys
     let lhsUntyped = foldl RApp (Var n) (map Var argNames)
     env <- getEnv
     (lhsTyped, _) <- check env lhsUntyped
     holeName <- fresh
     let rhs = Bind holeName (GHole retTy) (V 0)
     pure (MkFunClause lhsTyped rhs)
```

The `collectTypes` function we write in Idris is very similar to the one we defined in Figure 4.8 for the compiler implementation. This one, however returns a pair of the list of inputs and the output type.

The `addClause` tactic only takes one input, which is the name of the function we will add an initial clause for. Using this name, we look up the type of that function, normalize the type, and get its components using `collectTypes`. We name the list of input types `argTys`, and the output type `retTy` For each of member of `argTys`, we generate a new name using `fresh`.[1] We later use these names and make them into variable terms, using `Var`, and create a function application using all of these names. This application is supposed to represent the left-hand side in our final definition. The right-hand side is a hole term that has the type `retTy`. This concludes our type declaration term.

The Emacs Lisp code we write for this is the same as the existing add-clause editor action, so we only have to change the part that Emacs sends a message to the compiler. Our message now should refer to `addClause` instead of the built-in add-clause editor action.

Let's see this `Elab` action at work in Figure 5.2.

FIGURE 5.2. Example function to run `addClause` on.

─── Idris ───
```
example : (name : String) -> (age : Nat) -> IO ()
```

If we put the cursor on `example` and execute the Emacs Lisp code somehow, which is often done via a shortcut, we will get the result in Figure 5.3.

FIGURE 5.3. Result of running `addClause` on the `example` function.

─── Idris ───
```
example : (name : String) -> (age : Nat) -> IO ()
example a b = ?c
```

[1] `fresh` is defined in Hezarfen, not in a standard library. The standard way of creating fresh names is `gensym`, but we wrote wrapper function `fresh` that does not generate `MN`, and gives variables readable names, usually one-letter.

This concludes the rudimentary replacement the the existing add-clause section. For simplicity purposes, we have not covered the type signatures with implicit arguments or interface constraints. One can write a tactic to handle those cases as well.

5.2. Theorem prover for intuitionistic propositional logic

In this section, we describe the tactic Hezarfen[2], which can decide intuitionistic propositional logic theorems.[3] This tactic will be based on Dyckhoff's LJT [23] and its Haskell implementation Djinn [6], which generates Haskell expressions that have a given type.

Djinn is a standalone program that takes commands interactively, and when it generates an expression it prints it on the screen. Instead, we want to design Hezarfen as a library that provides an `Elab` action that can be used as a tactic in proofs, and also as a custom editor action that helps us when the built-in proof search mechanism does not suffice.

As a part of this library, we want to define the `Elab` action we used in section 3.1, with the type `prover : TTName -> Elab TT`. This tactic takes the name of the hole it is supposed to fill, and gives back a `TT` term in the `Elab` monad.

Since Hezarfen's proof terms are sent back to the editor and put back into the source code, we should aim to make our proof terms as simple as possible. Hence, we should implement both proof term generation and simplification.

5.2.1. Proof term generation. In Hezarfen, define two types `Context` and `Sequent` to help us represent the proof rules as manipulations of the goal type.

FIGURE 5.4. Definitions of `Context` and `Sequent` for Hezarfen.

```Idris
data Context = Ctx (List (TTName, Raw)) (List (TTName, Raw))
data Sequent = Seq Context Raw
```

---

[2] The name is pronounced [hɛzaɾfæn], like "has are fan", and it means polymath in Turkish. Tactic source code is available at http://github.com/joom/hezarfen.
[3] Similar to Coq's tauto tactic.

A context is two lists of pairs that consist of names mapped to Raw terms that represent the type of the term that the name refers to. The reason we want to have two lists is that we want to distinguish between the *consumed* and *unconsumed* bindings. Once we use up an entry in the second list, we delete it from the second list. But we may want to add new bindings, which we do on the first list. Suppose we have $(C \vee D) \supset B$ in our context. When we are checking the premises we want to remove it from the second list and add $C \supset B$ and $D \supset B$ to the first list. In Dyckhoff's presentation, this rule looks like:

$$\frac{C \supset B,\; D \supset B,\; \Gamma \Longrightarrow G}{(C \vee D) \supset B,\; \Gamma \Longrightarrow G}$$

But in Hezarfen's source code, this rule is written as in Figure 5.5.

FIGURE 5.5. The "Either implies" case in Hezarfen

```Idris
breakdown' : Sequent -> Elab Tm
breakdown' goal = case goal of
  -- numerous previous cases
  Seq (Ctx g ((n, `((Either ~d ~e) -> ~b)) :: o)) c =>
    let (n1, n2, newgoal) = !(appDisjImplL (Ctx g o) (d, e, b, c)) in
    let (l1, l2) = (!fresh, !fresh) in
    pure $ RBind n1 (Let `(~d -> ~b)
                      (RBind l1 (Lam d) (RApp (Var n)
                        `(Left a=~d b=~e ~(Var l1))))))
            $ RBind n2 (Let `(~e -> ~b)
                      (RBind l2 (Lam e) (RApp (Var n)
                        `(Right a=~d b=~e ~(Var l2))))))
              !(breakdown False newgoal)
```

Understanding this code fully is not necessary; our goal is to give the greater picture. This piece of code checks our second list in the context to see if there is a name with the type (Either d e) -> b, for some d, e and b. If there is, using this name n, we can create two functions, one with the type d -> b and the other with the type e -> b. We generate two fresh names so we can name these functions, and then we create a proof

term, in which we generate lambda bindings for these two functions we define in terms of `n`. The rest of the proof proceeds recursively.

The remaining rules of proof term generation proceed similarly. For more detail on what the generated proof terms are, see Dyckhoff's paper or Hezarfen's source code.

5.2.2. Simplification. In `Hezarfen.Simplify`, we define a function `reduce` that simplifies a given `Raw` term into another `Raw` term in the `Elab` monad.[4] A rudimentary implementation of this function that does not include all the simplification steps is given in **??**.

FIGURE 5.6. Rudimentary implementation of `reduce` in Hezarfen.

```
─────────────────────────── Idris ───────────────────────────
reduce : Raw -> Elab Raw
reduce t = case t of
  -- Eta reduction:  (\x => f x) becomes f
  RBind n (Lam b) (RApp t' (Var n')) =>
    if n == n'
      then reduce t'
      else pure $ RBind n (Lam b) !(reduce (RApp !(reduce t') (Var n')))

  -- (id x) becomes x
  RApp (RApp (Var `{id}) c) x => reduce x

  RBind n b t' => pure $ RBind n b !(reduce t')
  RApp t1 t2 => pure $ RApp !(reduce t1) !(reduce t2)
  _ => pure t
```

In the full implementation, we do more complex simplifications, such as simplifying `(\x => g (f x))` into `(g . f)`, removing unused **let** bindings, substituting a **let** binding in the body if the binding is only used once, etc.

To fully simplify a `Raw` term, we repeatedly apply it to `reduce` until fixpoint, which should be improved in future work.

There is also recent work on writing code generating theorem provers that are more modular and efficient since they depend on an intermediate proof representation that is

---

[4] We depend on the `Elab` monad for fresh name generation.

later reconstructed [53]. Hezarfen directly deals with the untyped core language syntax tree, both for the input terms that represent types, and for the proof term it returns, and this causes some overhead in our tactic.

CHAPTER 6

Related work

6.1. In Haskell

Template Haskell [52] is the main metaprogramming mechanism in Haskell. It is similar to elaborator reflection in the sense that metaprograms are defined in a monad called `Q`, which allows metaprograms to create fresh names and look up definitions. Template Haskell metaprograms generate expressions and definitions, which are among the capabilities of the `Elab` monad in Idris. However, there are significant differences; quotations in Template Haskell return values in the `Q` monad, and Template Haskell does not try to reflect the elaboration infrastructure of Haskell.[1] Neither does it hold an internal proof state that can be changed by monadic actions, nor it does try to provide an alternative way to implement tactics in Haskell.[2]

Brian McKenna worked on expanding the definitions generated by Template Haskell to source code, which then is pretty printed and put back into the source code in Emacs using YASnippet.[3]

On the IDE feature side of things, Alan Zimmerman and Matthew Pickering developed `ghc-exactprint`[4], which is a library that helps IDE and tooling development by providing a way to automatically refactor Haskell programs without changing a part of the program unintentionally. As they put it, their library respects "the identity

---

[1] However, Haskell metaprogramming using the GHC core language has been discussed in the GHC developers mail list, with credit to Idris: http://mail.haskell.org/pipermail/ghc-devs/2015-November/010402.html

[2] That being said, Siva Somayyajula has a rudimentary implementation a tactic monad in Haskell based on the `Q` monad: http://github.com/ssomayyajula/elab

[3] His tweet with screenshots can be found at http://twitter.com/puffnfresh/status/935274097642057728 and the project that enables this feature can be found at http://hackage.haskell.org/package/th-pprint.

[4] It can be found here: http://hackage.haskell.org/package/ghc-exactprint

refactoring", which is non-trivial if your system allows many different kinds of transformations [48]. There is also the Haskell IDE Engine project that aims to integrate many Haskell tools based on the GHC API to the editor workflow, by providing a backend for editor modes.[5]

6.2. In Agda

There is a line of work on bringing more automated theorem proving, proof automation and tactics, or metaprogramming to Agda. Lindblad and Benke (2006) introduced a term search algorithm called Agsy, a proof search mechanism that aims to save users' time by automating parts of the proof that are straightforward but tedious to write [34]. Agda has a derivative of this mechanism implemented as a part of its compiler. Kokke and Swierstra (2015) used the Agda's prior reflection system to define a new proof search mechanism in Agda itself [33]. The Hezarfen tactic we discussed in section 5.2 is not as advanced as their `auto` function, yet in their paper, they discussed a feature similar to edit-time tactics as future work:

> "In the future, it may be interesting to explore how to integrate proof automation using the reflection mechanism better with Agda's IDE. For instance, we could create an IDE feature which replaces a call to `auto` with the proof terms that it generates. As a result, reloading the file would no longer need to recompute the proof terms." [33]

In this thesis, we generalized their suggestion to all tactics, and specified how the editor/IDE and the compiler should communicate with each other in order to successfully call a tactic with inputs of the correct types.

There is also work on "proof by reflection" in Agda, which is different from our usage of the word "reflection" so far.

> "Reflection is an overloaded word in this context, since in programming language technology reflection is the capability of converting

---

[5] The project can be found here: http://github.com/haskell/haskell-ide-engine, and Alan Zimmerman's talk at the Haskell Implementors' Workshop 2017 can be found here: http://youtu.be/-pjQcG94CxM

some piece of concrete code into an abstract syntax tree object that can
be manipulated in the same system. Reflection in the proof technical
sense is the method of mechanically constructing a proof of a theorem
by inspecting its shape." [55]

We have been concerned with the first meaning of "reflection" in this thesis, how-
ever the work on the second meaning of this word is still relevant to proof automa-
tion, and their ideas can be reused in our edit-time tactics. Work by van der Walt and
Swierstra showed compelling examples of proof by reflection in Agda, such as a proof
mechanism for boolean tautologies [56].

6.3. In Coq

Coq has a metaprogramming mechanism called `template-coq`[6] that is based on
Malecha's term reification [37]. Recently a typed version of this system is also intro-
duced [4]. However, we are not aware of any work on using template metaprograms in
Coq to write new features for the editor.

Aside from this, there is a large body of work on proof automation, proof engineer-
ing and tactic languages in Coq. Coq's original tactic language is Ltac [22], which is
separate from its Coq's term language Gallina. However, alternatives to Ltac have been
developed, such as Mtac [59] and MetaCoq [58]. Especially Mtac, which is a tactic lan-
guage for Coq that facilitates custom proof search by providing a monadic interface,
has inspired further research in the area, including Idris' elaborator reflection [16].

Chlipala's *Certified Programming with Dependent Types* [14] has emerged as the canon-
ical introductory textbook for proof engineering; it explains the basics of tactic program-
ming and even delves into proof search and proof by reflection. Note that we use the
word reflection in the proof technical sense, as mentioned in the quote above.

---

[6] It can be found here: https://github.com/Template-Coq/template-coq

6.4. In Lean

Lean [21], which has a tactic metaprogramming system [24] similar to Idris' elaborator reflection, also allows running tactics in edit-time, and it does not require writing any code for the editor mode frontend.[7] The type of these editor actions can be seen in Figure 6.1.

FIGURE 6.1. Definition of `hole_command` in Lean.

```
meta structure hole_command :=
  (name  : string)
  (descr : string)
  (action : list pexpr → tactic (list (string × string)))
```

They provide the following documentation for `hole_command`:[8]

> "The front-end (e.g., Emacs, VS Code) can invoke commands for holes
> {! ... !} in a declaration. A command is a tactic that takes zero or
> more pre-terms in the hole, and returns a list of pair (`s`, `descr`) where
> `s` is a substitution and 'descr' is a short explanation for the substitution.
> Each string `s` represents a different way to fill the hole. The frontend is
> responsible for replacing the hole with the string/alternative selected
> by the user. This infrastructure can be used to implement auto-fill
> and/or refine commands. An action may return an empty list. This is
> useful for actions that just return information such as: the type of an
> expression, its normal form, etc."

In comparison to the edit-time tactics mechanism presented in our work, Lean's system is very restrictive. It only allows editor action that run on holes, but our system allows any kind of editor action as long as the user writes the necessary glue code in the editor mode language. We already showed in Figure 3.17, what the glue code to fill a hole would look like in Emacs Lisp. Another downside of Lean's system is that

---

[7] No Emacs Lisp if you are using Emacs.
[8] From the source code of Lean 3.4.1.

editor actions can only have a single type, as opposed to our system, which allows any kind of `Elab` action as long as the components of the a type all have an `Editorable` implementation. Our system lets users write more expressive custom editor actions.

6.5. Others

We should also think about the prospects for building editor interactions into a compiler from the start, so let's take a look at the existing work on languages that are designed with a priority on editor interactions.

Building editor interactions in a compiler from the start is not a new idea, both Idris and Agda have done this already. They did not, however, take metaprogrammable editor interactions into account, and that is what our work brings to Idris. We believe a path through Racket, a language-oriented programming [26, 27] language would be an interesting take on building a language around its editor interactions. DrRacket [28], Racket's IDE, makes writing editor interaction easy for the languages defined in Racket. This not only eliminates a lot of boilerplate code, but it also allows using Racket itself to define new editor actions. There are already dependently-typed languages defined in Racket: one example is Cur[9] [9], a proof assistant with powerful metaprogramming tools. There is also Pudding[10], a proof assistant in development that uses Racket for specifications, proof automation, code extraction and also extensions to the proof assistant itself. Another one is Pie[11] [19], a minimal language used for educational purposes. We believe there is potential for stronger editor interaction for these languages through metaprogramming.

Another path that is worth exploring more is structure editors. In the proof assistant world, The Alfa proof editor [30] has established a proof interface based on structure editor manipulating proof trees. More recently and for a simpler type theory, the Hazel project [46, 47] explored what a language designed around its editor would look like. Specifically, they designed a structure editor and a type theory to deal with incomplete

---

[9] It can be found here: http://github.com/wilbowma/cur
[10] It can be found here: http://github.com/david-christiansen/pudding
[11] It can be found here: http://github.com/the-little-typer/pie

programs in this setting. They also coined the term "edit-time" to mean when the user is writing a program in the editor, and suggested "edit-time tactics" as future work[12], by which they meant a separate language in which users can define editor actions, and a library of predefined editor actions that the users can compose.

Another question we should answer is how feasible it is to implement delaboration in such a way that the compiler could respond directly with surface-syntax terms that fit in the current binding context. Currently there is no way in Idris to write an editor action that returns a surface-syntax term.[13] The way elaborator reflection is defined in Idris forces us to deal with core language terms only, and for the rest we depend on the built-in delaboration. There is also no reflected Idris type that represents the surface syntax, since the surface syntax can change quite often, maintaining its Idris representation would be difficult, not to mention with every change it would likely break users' code that depends on it. Therefore, adding an Idris representation of the surface-syntax is not planned.

Apart from Idris, it is possible to design a language that lets the users define editor actions that return surface-syntax terms. We see two possible ways to do this:

(1) Not having a core language and surface-syntax distinction. This is not ideal if you have a large programming language, then the type-checking, evaluation, etc. have to be extended every time we want to add a new syntax. Not to mention that lacking features like implicit arguments is bad language ergonomics; elaboration is needed to resolve the implicit arguments [49].

(2) Having a reflected type in your language that represents the surface-syntax terms, exposing the delaboration mechanism in your metaprogramming mechanism, and allowing splicing surface-syntax terms into programs. We are not aware of any work that does this.

---

[12] We learned this from their slides and also personal communication with Cyrus Omar and Ravi Chugh.
[13] The only way around returning surface-syntax directly from an editor action is to return a `String` that consists of the code, but that is inelegant and we would like to avoid that.

CHAPTER 7

Conclusion

7.1. Future work

7.1.1. Proof simplification. The ability to run tactics as editor actions has a consequence that we have not explored much in this thesis. Idris tactics generate proof terms at compile time, but their compilation can take a long time for complex tactics[1], not to mention that the implementation of elaborator reflection in Idris has significant performance issues, as shown by Ebner et al. [24]. Yet we still want to utilize complex tactics to generate proofs or terms. Using edit-time tactics, one would run a tactic once from the editor, generate the proof term and serialize and send that to the editor and put it back in the file. If we think of the differences between the traditions of writing the proof terms directly and writing tactics, the former more common in Agda and Idris and the latter in Coq, this work will constitute a one way bridge between the two, by making use of the elaborator reflection to create proof terms in the editor in a smarter and quicker way.

The problem with that approach is that the generated proof terms can be (and often are) gigantic and hideous, especially if generating a minimal proof term is not a priority for the tactic we are using. If there was a generic mechanism to simplify and minimize the generated proof terms, and even write them in a way that makes use of dependent pattern matching, then this could be a more usable consequence of this work. Ideally, we would want the artifact we are handing in to the reader of our proofs to look just like what it would be if we had not used this system. We leave that for future work.

---

[1] Similar problems arise in Coq as well. For example, theorems that use the famous omega tactic that decides Presburger arithmetic [50] take a long time to compile, and it usually generates a huge proof term.

However, even without proof simplification, edit-time tactics still could be a last resort solution to long compile times for tactics.

7.1.2. Writing an editor action frontend in Idris. We explained in chapter 1 that this thesis focuses on writing the backend of an editor action in Idris, and that we still had to write some Emacs Lisp (if we are using Emacs). However, Idris supports many different code generation targets [1] seamlessly.

For example, since compiling to JavaScript is built-in, we can use JavaScript code generation to write the editor interaction frontend for Visual Studio Code and Atom.

There are also experimental projects on compiling Idris to Emacs Lisp[2] and VimL (Vimscript)[3]. These projects are not mature enough yet, but we believe they have the potential to inspire different applications of metaprogramming, especially if the Idris modes of these editors are written in Idris via their respective code generation targets.

7.2. Final words

In this thesis, we extended the capabilities of the editor interaction mode of Idris by allowing users to define new editor actions in Idris itself. We did so through a metaprogramming technique that was introduced to Idris recently by Christiansen and Brady [16].

Editors communicate with the compiler via S-expressions, so we gave users the power to dictate how a value of a given Idris type should exactly be communicated; through the `Editorable` interface users are now able to define how a received S-expression should be parsed by the compiler, and how the compiler should send the result as an S-expression. To achieve this, we reflected the `SExp` type to Idris, and extended elaborator reflection by adding new `Elab` primitives, with which we defined the `Editorable` implementations for Idris types representing the Haskell representation of Idris core language terms.

---

[2] Steven Shaw's work on compiling Idris to Emacs Lisp: http://github.com/steshaw/idris-elisp

[3] Oskar Wickström and Soham Chowdhury's work on compiling Idris to VimL: https://github.com/owickstrom/idris-vimscript

Using this feature, we showed a simple `toy` example, and then the `addClause` example that can replace an existing built-in editor action, and Hezarfen, which is meant to be a better proof search mechanism than the built-in one. We believe there is potential to replace even more of the built-in editor actions with edit-time tactics, such as case-splitting and lifting a hole into a lemma. We can also add new general edit-time tactics, such as renaming a binder, renaming a function within a file, pruning unused arguments in a function, etc.

We also believe that as more decision procedures are coded up in Idris, edit-time tactics can become a more popular feature. Especially library and DSL authors can ship custom editor actions for their package, which would allow library users to write code more easily with that library or DSL.

Hopefully our work will bring dependently-typed languages one step closer to the state-of-the-art IDEs, and even give them an edge by allowing the reuse of the existing metaprogramming mechanisms and tactic engineering efforts to write editor actions.

Bibliography

[1] Code generation targets Idris 1.2.0 documentation. http://docs.idris-lang.org/en/latest/reference/codegen.html, . Accessed: 2018-04-23. 68

[2] Frequently asked questions Idris 1.2.0 documentation. http://docs.idris-lang.org/en/latest/faq/faq.html, . Accessed: 2018-04-23. 3

[3] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *ACM Sigplan Notices*, volume 49, pages 233–249. ACM, 2014. 1

[4] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Typed template Coq–certified meta-programming in Coq. 2018. 63

[5] David Aspinall. Proof General: A generic tool for proof development. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 38–43, 2000. 2

[6] Lennart Augustsson. Announcing Djinn. http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747, 2005. Accessed: 2017-10-04. 57

[7] Henk Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1(2):125–154, 1991. 10

[8] Yves Bertot. The CtCoq system: Design and architecture. *Formal aspects of Computing*, 11(3):225–243, 1999. 3

[9] William J Bowman. Growing a proof assistant. *Higher-Order Programming with Effects*, 2016. 65

[10] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. 2, 5, 12, 14

[11] Edwin Brady. *Type-Driven Development with Idris*. Manning Publications, 2017. ISBN 1617293024. 2

[12] Pierre Castéran and Yves Bertot. Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions., 2004. 1

[13] Larry Charles. Seinfeld s4e17, The Outing., Feb 1993. 37

[14] Adam Chlipala. Certified programming with dependent types, 2011. 63

[15] David Christiansen. Coding for types: Universe pattern in Idris. http://youtube.com/watch?v=AWeT_G04a0A, 2015. Curry On Prague. 37

[16] 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. ii, 5, 12, 14, 17, 19, 63, 68

[17] David Raymond Christiansen. Type-directed elaboration of quasiquotations: A high-level syntax for low-level reflection. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*, page 1. ACM, 2014. 15, 17

[18] David Raymond Christiansen. *Practical Reflection and Metaprogramming for Dependent Types*. PhD thesis, IT University of Copenhagen, Software and Systems Section, 2016. iv, 10, 19

[19] David Thrane Christiansen and Daniel P. Friedman. The little typer. Unpublished, 2018. 65

[20] Catarina Coquand, Makoto Takeyama, and Dan Synek. An Emacs interface for type directed support constructing proofs and programs. In *European Joint Conferences on Theory and Practice of Software, ENTCS*, 2006. 2

[21] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015. 1, 64

[22] David Delahaye. A tactic language for the system Coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 85–95. Springer, 2000. 19, 63

[23] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992. 7, 57

[24] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):34, 2017. 5, 64, 67

[25] Richard A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. PhD thesis, University of Pennsylvania, 2016. 1, 37

[26] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket manifesto. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. 65

[27] Daniel Feltey, Spencer P Florence, Tim Knutson, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Languages the Racket way. *Language Workbench Challenge*, 2016. 65

[28] Robert Bruce Findler. DrRacket: The Racket programming environment, 2013. 65

[29] Mike Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction*, pages 169–186, 2000. 1

[30] Thomas Hallgren. The proof editor Alfa. http://www.cse.chalmers.se/hallgren/Alfa, 1998. 65

[31] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of functional programming*, 16(2):197–217, 2006. 49

[32] Simon Peyton Jones and Erik Meijer. Henk: a typed intermediate language. *TIC*, 97, 1997. 10

[33] Pepijn Kokke and Wouter Swierstra. Auto in Agda. In *International Conference on Mathematics of Program Construction*, pages 276–301. Springer, 2015. 62

[34] Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in Agda. In *International Workshop on Types for Proofs and Programs*, pages 154–169. Springer, 2004. 62

[35] Zhaohui Luo. Developing reuse technology in proof engineering. In *Proceedings of AISB95, Workshop on Automated Reasoning*. Citeseer, 1995. 3

[36] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *International Workshop on Types for Proofs and Programs*, pages 213–237. Springer, 1993. 2

[37] Gregory Michael Malecha. *Extensible Proof Engineering in Intensional Type Theory*. PhD thesis, Harvard University, 2014. 63

[38] Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, 2000. 17

[39] Conor McBride and James McKinna. The view from the left. *Journal of functional programming*, 14(1):69–111, 2004. 2, 8

[40] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960. 21

[41] Hannes Mehnert and David Christiansen. Tool demonstration: An IDE for programming and proving in Idris. *Proceedings of Vienna Summer of Logic, VSL*, 14, 2014. 2

[42] Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 49–60. ACM, 2007. 47

[43] Anne Mulhern, Charles Fischer, and Ben Liblit. Tool support for proof engineering. *Electronic Notes in Theoretical Computer Science*, 174(2):75–86, 2007. 3

[44] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002. 1

[45] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007. 2

[46] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, 2017. iv, 65

[47] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. Toward semantic foundations for program editors. In *Summit on Advances in Programming Languages (SNAPL 2017)*, 2017. iv, 65

[48] Matthew Pickering. Announcing ghc-exactprint: A new foundation for refactoring tools, Jul 2015. URL http://mpickering.github.io/posts/2015-07-23-ghc-exactprint.html. 62

[49] Robert Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks, Antibes*, volume 4, 1990. 66

[50] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991. 67

[51] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. *ACM Sigplan Notices*, 43(9):51–62, 2008. 10

[52] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002. iii, 61

[53] František Silvaši and Martin Tomášek. Fully automatic modular theorem prover with code generation support. In *Informatics, 2017 IEEE 14th International Scientific Conference on*, pages 332–338. IEEE, 2017. 60

[54] Laurent Théry, Yves Bertot, and Gilles Kahn. *Real theorem provers deserve real user-interfaces*, volume 17. ACM, 1992. 3

[55] Paul van der Walt. Reflection in Agda. Master's thesis, 2012. 12, 63

[56] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*, pages 157–173. Springer, 2012. 63

[57] Makarius Wenzel. Isabelle/jEdit-a prover IDE within the PIDE framework. In *AISC/MKM/Calculemus*, pages 468–471. Springer, 2012. 2

[58] Beta Ziliani. Introducing MetaCoq: A safe tactic language for Coq. 63

[59] Beta Ziliani, Derek Dreyer, Neelakantan R Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: a monad for typed tactic programming in Coq. *Journal of Functional Programming*, 25, 2015. 63