Extensible Type-Directed Editing

Joomy Korkut
Wesleyan University
Middletown, Connecticut, USA
ckorkut@wesleyan.edu

David Thrane Christiansen
Galois, Inc.
Portland, Oregon, USA
dtc@galois.com

Abstract

Dependently typed programming languages, such as Idris and Agda, feature rich interactive environments that use informative types to assist users with the construction of programs. However, these environments have been provided by the authors of the language, and users have not had an easy way to extend and customize them. We address this problem by extending Idris's metaprogramming facilities with primitives for describing new type-directed editing features, making Idris's editors as extensible as its elaborator.

CCS Concepts • **Software and its engineering** → *Functional languages*; *Language features*;

Keywords Metaprogramming, dependent types, editors

ACM Reference Format:
Joomy Korkut and David Thrane Christiansen. 2018. Extensible Type-Directed Editing. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '18), September 27, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3240719.3241791

This paper uses colors in the example code.

1 Introduction

Rich type systems give programmers a way to express their intentions as types, statically ruling out many incorrect programs. But rich types are useful for much more than preventing mistakes: the information provided by informative types can be used by programming tools to guide program construction, automating away tedious details and freeing programmers to concentrate on the parts of their problem that require human creativity.

Type-driven programming environments are necessarily built according to language developers' assumptions about how programmers will use them. These assumptions, however, can never hold for all members of a diverse community working on a variety of problems. Unfortunately, the interactive features of Idris and Agda are presently built in to their respective compilers, and skill in dependently typed programming does not imply the ability to extend the implementation of dependently typed languages and maintain those extensions so that they continue to work as compilers are improved.

The Idris elaborator [7] translates programs written in Idris into a smaller core type theory, called `TT`. The elaborator is written in Haskell, making use of an elaboration monad to track the complicated state that is involved. The high-level Idris language is extensible using *elaborator reflection* [8, 10], which directly exposes the elaboration monad to Idris programs so that Idris can be extended in itself. Concretely, elaborator reflection extends Idris with a primitive monad `Elab`. Just as `IO` values describe effectful programs to be executed by the run-time system, `Elab` values describe effectful programs to be run during elaboration.

We have extended Idris's implementation of elaborator reflection with new primitives that enable it to be used to construct *editor actions*. These editor actions have access to the full power of `Elab`, but instead of running in the course of elaboration, they are manually invoked by programmers to modify already-elaborated programs. With these new primitives, it becomes possible to write domain-specific editor actions for embedded domain-specific languages [23] and to replace parts of the compiler with customizable library code written in Idris. Even more importantly, users who were previously stuck with whatever the developers provided are now empowered to make not only their language, but also their programming environment, their own.

Contributions

We make the following contributions in this paper:

- We explore the features that are necessary to use elaborator reflection to implement editor actions.
- We describe a concrete realization of this design, and the communication protocol that allows it to work in multiple interactive environments.
- We describe a non-trivial editor action that invokes a theorem prover for intuitionistic propositional logic to interactively fill a hole in an incomplete program.
- We demonstrate that editor actions written in Idris are sufficiently powerful to replace parts of implementation by reimplementing a feature that constructs initial implementations of functions, based on their type signatures.

TyDe '18, September 27, 2018, St. Louis, MO, USA

1.1 Editor Interaction

An interactive environment for Idris programming has two key components: the Idris compiler, which is responsible for type checking and code generation, and a text editor, which provides an interface in which users can write programs. While some IDEs tightly couple the text editor component to the compiler component, requiring them to run in the same process and to be written in the same language, Idris provides an interaction protocol that can be used to bring type-directed program construction to *any* sufficiently extensible text editor. While this protocol is based on the Swank protocol used by Common Lisp and Dylan implementations since 2003, it is also similar to the more recent Language Server Protocol used in Visual Studio Code.

Using this protocol, client editors can invoke the type checker, request that the compiler perform a case split on a pattern variable, generate initial implementations for functions based on their type signature, discover the callers of a function, or request documentation. The protocol even allows interactive environments to request the normal form of an expression displayed in an error message, allowing in-place evaluation. However, the file that is being edited has to be loaded to the compiler, which allows the compiler to use the context in the file for the editor actions above. Idris does not yet support a mode of interaction similar to that of Lean, which incrementally type checks the buffer as users type; like Agda, the Idris type checker must be explicitly invoked.

At the time of writing, there are Idris editing modes for GNU Emacs, Atom, Sublime Text, and Visual Studio Code that communicate with the Idris compiler over the IDE protocol and allow compiler-supported editor actions. Each editor requires a certain amount of custom code to connect the user interface to the underlying Idris compiler commands. Because the Idris mode for GNU Emacs is the most featureful, we use it as our running example, but there is nothing inherently Emacs-specific about this technique.

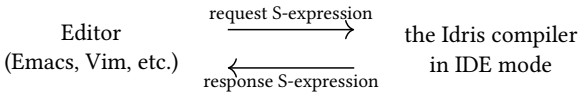

Figure 1. Communication between an editor and Idris

When the user invokes an editor action, the editor has to tell the Idris compiler what to run. Since the editor and the compiler run in separate processes, for each interaction the editor has to send a message to the compiler, and the compiler has to send a one or more replies back to the editor, as seen in figure 1. For ease of parsing, these messages are formatted as McCarthy's S-expressions [33].

The Idris compiler, including its editing commands, is written in Haskell. Traditionally, implementing a new command required extending the compiler, updating the IDE protocol, and then finally extending the user interface of the editor modes.

1.2 Extending An Editor in Idris

Dependently typed languages typically both allow programs to be incomplete and provide support for making them more complete. A limited version of this support could be a facility that substitutes the unit constructor (written `()`, as in Haskell) for a hole of the unit type (also written `()`, as in Haskell), and the reflexivity constructor `Refl` when the goal is a reflexive case of the equality type.

Figure 2 presents an implementation of this editor action. The **`%editor`** keyword registers the declaration as an editor action. Its type states that, when passed a representation of a name from Idris's core language, it will produce a representation of a term in Idris's core language, potentially having elaboration-time side effects. It is passed a name because Idris holes are identified by name.

The first step is to look up the type of the hole to be replaced, using `getType`, which takes a name and returns the type of the definition associated with that name. If the name is ambiguous, `getType` fails. Having discovered the name's type, it then pattern-matches on said type, using Idris's quasiquotation syntax [9].

The first case to be considered is the unit type. In this pattern, a type annotation is needed due to the Haskell-style overloading of the double-parentheses. If the case is the unit type, the quoted form of the unit constructor is returned with `pure`, which is analogous to Haskell's `return`.

```
%editor
easy : TTName -> Elab TT
easy n =
  do ty <- getType n
     case ty of
       `(() : Type) =>
         pure `(() : ())
       `((=) {A=~a} {B=~b} ~x ~y) =>
         do converts a b
            converts x y
            pure `(Refl {A=~a} {x=~x})
       _ =>
         fail [TextPart "Cannot solve"]
```

```
(defun idris-easy ()
  "Invoke the first example action."
  (interactive)
  (idris-elab-hole-arg
   "easy" (list (idris-name-at-point))))
```

Figure 2. A simple editor action in Idris (top) and its Emacs Lisp support code (bottom)

The second case to be considered is the equality type, which is heterogeneous [30] in the Idris standard library. The equality type requires two implicit arguments [37], called `A` and `B`, as well as explicit arguments `x` and `y`. When `A` and `B` are the same type, and `x` and `y` can be judged to be equal according to that type, `Refl` proves the equality. The `converts` action checks whether two quoted terms are judgmentally equal, and fails if they are not. Having checked that the types and their inhabitants coincide, the second case returns `Refl`.

The third and final case matches any other goal, and it fails. Additional cases could be added on an *ad hoc* basis, or a more automatic approach could be taken. See Christiansen [10] or Christiansen and Brady [8] for a description of how to increase the level of automation; this example is chosen to be easier to understand.

Each of Idris's editor actions requires a small amount of editor-specific code to provide a user interface, and editor actions written in Idris are no exception. With a suitable library, most editing actions can be accommodated with fewer than five lines of Emacs Lisp, and we expect the burden to be similar for other extensible editors. Including in-editor documentation, this example requires five lines of Emacs Lisp.

```
ex1 : ()                    ex1 : ()
ex1 = ?ex1_impl             ex1 = ()

ex2 : not False = True      ex2 : not False = True
ex2 = ?ex2_impl             ex2 = Refl

ex3 : False = True          ex3 : False = True
ex3 = ?ex3_impl             ex3 = ?ex3_impl
```

Figure 3. Before and after invoking `easy`

Figure 3 displays the results of executing this editor action on three holes. In the first two examples, the program is completed automatically. In the third example, however, an error is indicated because the underlying `Elab` action fails.

2 Design

The `Elab` monad in Idris primitively keeps track of a state involving a potentially incomplete expression, its type, and any new declarations generated as side effects during elaboration. When an `Elab` script is executed, the incomplete expression is expected to have been completed. Because these updates to the expression occur via side effects, elaborator reflection scripts have the type `Elab ()`. Since the desired metaprogramming effects are captured by the elaboration state, there is nothing interesting to return.

However, `Elab` scripts that are used as editor actions are not able to effect changes to the program by modifying the elaboration state, because the contents of the text editor are not part of the state. Thus, editor actions return their results explicitly, and the serialized results are sent to the editor.

If an editor action needs to send back an expression to the editor, then the action should have the return type `Elab TT`, where `TT` is the type of quoted core Idris terms. Similarly, if a user needs to define an action that creates a function definition, then the action that does that should have the return type `Elab FunDefn`, where `FunDefn` is the type of quoted function definitions. A simple editor action that only needs to send a number back to the editor should return an `Elab Nat`, where `Nat` is the type of natural numbers.

Using the `TT` datatype to send and receive Idris expressions from the editor instead of a precise representation of the concrete syntax of high-level Idris allows programmers to reuse existing `Elab` scripts in custom editor actions. Additionally, `TT` works with the existing elaboration infrastructure, including type checking and evaluation, and we expect it to be far more robust in the face of future changes to Idris, because the high-level language changes much more frequently than the core language. However, since `TT` represents terms and declarations in `TT` rather than high-level Idris, editor actions cannot return an exact concrete syntax. We explain our solution for the gap between the core language terms in `Elab` actions and the concrete syntax used in the editor in sections 2.1.2 and 3.2.

2.1 The `Editorable` Type Class

The problem with allowing editor actions to return inhabitants of any type is that the compiler cannot serialize values of arbitrary types as S-expressions. In order to give users the power to define how each type should be represented as S-expressions, we define a type class[1] called `Editorable`, which outlines what the compiler needs to know about a type to be able to serialize and deserialize values of that type.

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

Figure 4. Definition of the `Editorable` type class.

Whenever users want to inform the compiler about the S-expression representation of values of a type, they have to define an instance of the `Editorable` type class. Later when a user runs an editor action from an editor, the `Editorable` instances are used for communication via S-expressions.

2.1.1 Some `Editorable` Instances

The collection of atoms in Idris's S-expressions already includes many primitive types, such as `String`. Deserializing a

[1] In Idris, type classes are called *interfaces* and instances are called *implementations*.

string succeeds when provided with a string, and fails otherwise. The message thrown on failure can be a non-trivial list structure, which allows Idris's pretty printer to be used to render substrings, but here we elide the concrete messages and focus on the successful cases. Serialization tags the atom appropriately.

```
implementation Editorable String where
  fromEditor (StringAtom s) = pure s
  fromEditor x = fail [{- elided -}]
  toEditor x = pure (StringAtom x)
```

Figure 5. `String` instance of the `Editorable` type class.

Inductive types, such as `Maybe a`, can be represented as lists in which the first element is a tag specifying the chosen constructor. For instance, `Just "abc"` can be represented as `(:Just "abc")`, a list S-expression with a symbol atom as the first element and then the S-expression representation of a string, and `Nothing` can be represented as the symbol `:Nothing`. This can be implemented as follows:

```
implementation Editorable a
            => Editorable (Maybe a) where
  fromEditor (SExpList [SymbolAtom "Nothing"]) =
    pure Nothing
  fromEditor (SExpList [SymbolAtom "Just", x]) =
    do x' <- fromEditor x
       pure (Just x')
  fromEditor x = fail [{- elided -}]
  toEditor (Just x) =
    do x' <- toEditor x
       pure (SExpList [SymbolAtom "Just", x'])
  toEditor Nothing =
    pure (SExpList [SymbolAtom "Nothing"])
```

Figure 6. `Maybe` instance of the `Editorable` type class.

The idea that is introduced here can be used to define an `Editorable` instance for any inductively-defined data type, so long as the arguments to its constructors are also inductively defined. Constructors that do not take any argument are represented as symbol atoms, and the ones that do take arguments are represented as a list S-expression, in which the first element is a symbol atom and the other elements represent the arguments that the constructor takes. We will call this the *constructor-based S-expression representation* of a type.

It is not, however, possible to use the constructor-based representation for every type. In particular, functions and infinite coinductive datatypes do not, in general, admit finite serializations.

In other cases, the constructor-based representation requires too much work to encode and decode in editors. For instance, Idris names have a rich structure, but users know them by their syntax rather than by their internal representation. The `Editorable` instance for the type of quoted Idris names, `TTName`, which appeared in figure 2, represents names using their user-facing syntax. For instance, the Idris name `Prelude.Bool.not`, which has the data type representation `NS "not" ["Bool", "Prelude"]`, is represented by a string atom S-expression, namely `"Prelude.Bool.not"`.

2.1.2 Primitive `Editorable` Instances

The S-expression representations of quoted Idris code, such as `TT` and `TyDecl`, are the most challenging ones. These types mirror the internal representation of Idris's core language, but they are ordinary inductive data types defined in Idris, which means that the constructor-based representation suffices to represent them.

However, that representation is not particularly convenient for extending editors. The constructor-based representation would be an abstract syntax tree of the TT representation of an Idris expression. Users, however, work with the concrete syntax of Idris itself. When they use editor actions, they expect to see concrete syntax put back into the file, and converting from `TT` to concrete Idris syntax requires a lot of code that should not be duplicated in each editor when it already exists in the Idris compiler. Therefore, for these core datatypes, the editor sends and receive concrete syntax.

If the compiler receives concrete syntax and needs to run `Elab` actions on that, there are many missing steps in between, most important of which is elaboration from a high-level language to the core language. Similarly, if the compiler needs to send back concrete syntax after running `Elab` actions, then it needs to reverse all those steps. In other words, there is a colossal gap between concrete syntax and the core language that needs to be bridged, and this task can be delegated to the `Editorable` type class.

When the S-expression received by the compiler is a string atom that is a piece of Idris code, i.e. concrete syntax, `fromEditor` should parse that string into a high-level language term, and then elaborate that into a core language term. Only after that can the compiler run the `Elab` editor action. Similarly, when the `Elab` action finishes, `toEditor` should convert core language terms into the high-level language terms, a process called *delaboration* in the Idris compiler. Then, the compiler should invoke the pretty printer to get concrete syntax that represents that term. The resulting string can be sent back from the compiler to the editor as a string atom S-expression.

Bridging this gap requires an `Editorable` instance for `TT` that does parsing, elaboration, conversion from the core language to the surface language, and pretty printing. Rather than reimplementing this from scratch in Idris itself, we extended `Elab` to expose these features of the compiler as primitives, following Barzilay's program of *direct reflection* [5]. In particular, these primitives are used to define the instances

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

Figure 7. Definition of the `HasPrim` predicate in Idris.

```
prim__fromEditor : HasPrim a -> SExp -> Elab a
prim__toEditor : HasPrim a -> a -> Elab SExp
```

Figure 8. The new `Elab` primitives.

```
implementation Editorable TT where
  fromEditor x = prim__fromEditor HasTT x
  toEditor x = prim__toEditor HasTT x
```

Figure 9. An `Editorable` instance depending on the new primitives.

of `Editorable` for the core language types like `TT`, `TyDecl` and `FunDefn`. Hard-coding the `Editorable` instances of `TT`, `TyDecl`, `DataDefn`, `FunDefn`, and `FunClause` into the compiler allows by making use of the already existing compiler implementations of the steps listed above.

To achieve this, the existing `Elab` monad needs to be extended with primitives that go through the steps mentioned above. One `Elab` primitive for `fromEditor` and one for `toEditor` suffice; polymorphic primitives constrained by an indexed family provide a principled way to manage the primitive instances of `Editorable`.

Figure 8 uses `HasPrim` to describe the new `Elab` primitives. Using these two primitives, the `Editorable` instances for the core language types all look alike; an example can be seen in figure 9.

2.2 How the Compiler Uses `Editorable` for Communication

We have extended Idris's IDE protocol to support an additional message that represents an invocation of a custom editor action. This message includes the name of the custom action and a list of arguments, and its reply contains Idris's response.

When the editor fires up Idris in IDE mode and loads the file, then it can send a custom action message to Idris. If the compiler receives such a message from the editor, it looks up the name and type of the editor action from the context. From types of the arguments of the `Elab` action, it can find the necessary `Editorable` instances and use the `fromEditor` definitions in them to parse the S-expressions into Idris values. If the number of arguments in the action type and the argument list match, and all arguments can be parsed without any errors, then the compiler can run the `Elab` action, and use `toEditor` to serialize the output, and send it back to the editor.

The compiler can use `Elab` actions whose arguments and return type have `Editorable` instances as custom editor actions. The `easy` action from figure 2 is an example of this, and its usage can be seen in figure 3. When the user puts the cursor on `?ex1_impl` and invokes `idris-easy` in their Emacs session, Emacs sends a message to the compiler that specifies that it wants to run `easy`, and provides a list of arguments, (`list "ex1_impl"`), which is a singleton list containing a string atom. When Idris receives this message, it looks up the name `easy` from the context and finds out that it has the type `TTName -> Elab TT`. Therefore it looks up the `Editorable` instance of `TTName` and runs its `fromEditor` implementation on `"ex1_impl"`, which results in the Idris name for `?ex1_impl`. Then the compiler can execute `easy` and get a core language term `()` as a result. Since core language terms are represented by the `TT` type, Idris has to find the `Editorable` instance of `TT` and run its implementation of `toEditor` on that term, which produces an S-expression to be sent back to the editor.

2.3 Using `Editorable` in Type-Checking

The motivation behind the `Editorable` type class is twofold:

1. to use the `fromEditor` and `toEditor` definitions to serialize and deserialize data before and after an `Elab` action is run; and
2. to check whether a given `Elab` action is suitable to be used as an editor action.

The first motivation is already covered in the previous sections. When Idris encounters a definition that is tagged with the **`%editor`** keyword as an editor action, it first elaborates the type. The next step is to check whether this type is suitable as an editor action. It does this by ensuring that each argument type has an `Editorable` instance, that the return type has `Elab` at its head, and that the argument to `Elab` is also `Editorable`. This rules out dependent types for editor actions—section 6.2 discusses a potential way to lift this restriction in the future.

3 Implementation Concerns

The overall design described in section 2 is not completely sufficient to implement extensible type-directed editing. Some additional machinery proves to be necessary in practice.

3.1 Local Contexts

Expressions can be understood only in a local context that explains the types, and sometimes the values, of their free variables. Editor actions should have access to the local context of bound variables in addition to the global definition

context. The design in section 2, however, has no means of providing them with this local context.

For example, the editor might send an expression like `not a` to the compiler, where `a` is bound in the local context. The compiler can parse this expression, but it cannot elaborate it, because without the local context, `a` is meaningless. In a call to a custom editor action, expressions stand alone; they do not come with a context. It can deal with `not`, which is defined in the global context, but when it comes to the local context, elaboration is doomed to fail.

Editor actions have, thus far, been provided with their arguments explicitly. However, local contexts do not have a concrete representation in Idris's syntax, so they cannot be selected directly.

To solve this problem, we take advantage of the fact that lexical contexts correspond to source spans. Each local binding form has a defined scope; this scope corresponds to a production of the abstract syntax tree that originates from a specific range of positions in the editor buffer. We extended the protocol for custom editor actions so that the editor sends a source position along with the action name and its arguments.

Prior to our work, local context information was only available for holes, and they were tied to names of the holes, not their source positions. To be able to keep track of the correspondence between source positions of all expressions and their local contexts, we extended the internal compiler state with an interval map that connects ranges of source positions to the local context that corresponds to the process of elaborating the expression found in that range. We used the standard Haskell finger tree implementation of interval maps [22]. Entries in interval maps are accumulated during elaboration and saved, to be used later in editor interaction.

When initializing the reflected elaboration monad prior to executing an editor action, the local context is initialized with the one that corresponds to the cursor location sent by the editor. The compiler can use that information in the elaboration of terms depending on the local context, such as `not a`. This constrains editor actions to have a single privileged source position; this constraint has not proven difficult in practice, but it could be lifted by making source positions `Editorable` and providing any number of them as ordinary arguments. Editor actions could then have a primitive to enter the lexical scope corresponding to a source position, and to check whether a source span is contained within a particular scope.

We expect that the remembered association between source spans and contexts will enable additional editor features, such as displaying the local context as the user navigates a source file, but we have not yet implemented these features.

3.2 Hard-Coding `Editorable` Instances

Implementing hard-coded instances of the `Editorable` type class in the compiler is challenging to describe since there are many languages involved in different ways. Idris's compiler is written in Haskell, hence there is a Haskell data type that represents Idris syntax trees. Idris's elaborator [7] describes a core language that is smaller than Idris's high-level language, there is also a Haskell data type that represents Idris core syntax trees.

However, elaborator reflection [8, 10] provides new Idris data types that correspond to the Haskell data types to represent Idris core language terms. Outlining how a metaprogramming feature is implemented introduces another layer of metadiscussion, therefore it becomes difficult to use precise terminology. Figure 10 describes the relationship between the different languages and representations and spells out the specific names for moving from one to another.

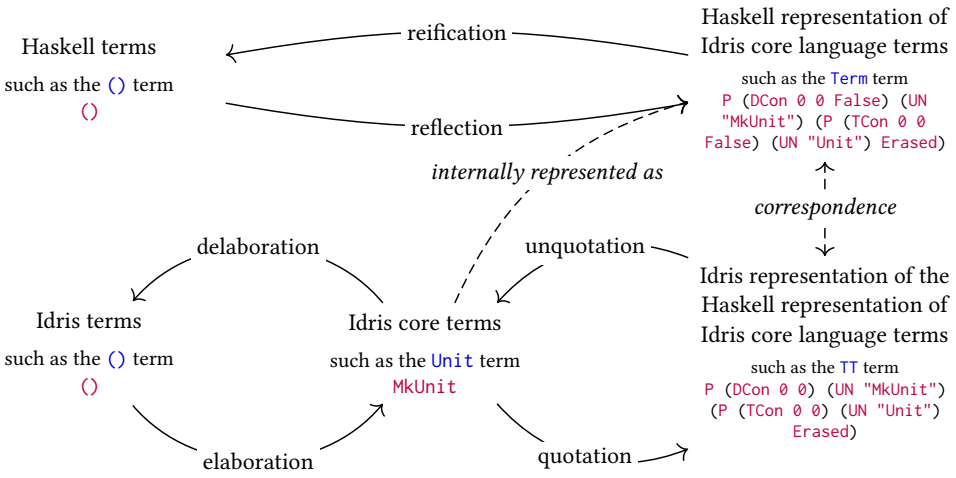

Figure 10. The relationship between reflection, reification, quotation, unquotation, elaboration and delaboration.

During the execution of `prim__fromEditor` in the elaborator, it is given *arg*, a Haskell representation of an Idris core language term representing an S-expression, and *ty*, a Haskell representation of an Idris type. When the elaborator finishes running `prim__fromEditor`, it should create the Haskell representation of an Idris core language term, which should have the type *ty*. The elaborator should have a case for each primitive `Editorable` implementation. For brevity, we will only consider the case in which *ty* corresponds to the Idris type `TT`. Then the elaborator should

1. reify *arg* to get a Haskell S-expression and make sure it is a string atom;
2. parse the string inside the S-expression and get a high-level language term;
3. traverse the high-level language term and resolve namespaces for names, for when it is unambiguous;
4. elaborate the new high-level language term into a core language term, using the local context obtained through section 3.1;
5. reflect the core language term, in order to create a core language term that represents an Idris term of the type `TT`; and finally
6. normalize the reflected term to get a syntax tree in canonical form, and return it.

During the execution of `prim__toEditor` in the elaborator, it is given *ty*, a Haskell representation of an Idris type, and *arg*, a Haskell representation of an Idris core language term representing a term of the type *ty*. When the elaborator finishes running `prim__toEditor`, it should create the Haskell representation of an Idris core language term representing an S-expression. The `TT` case for `prim__toEditor` should

1. reify *arg* to get a Haskell representation of an Idris core language term;
2. delaborate and resugar the core language term into a high-level language term;
3. use pretty printing to get a string that is a piece of code;
4. create a string atom S-expression with that string;
5. reflect the S-expression to get a Haskell term representing an Idris core language term representing an S-expression; and finally
6. normalize the reflected term to get a syntax tree in canonical form, and return it.

The other primitive instances behave similarly.

4 Applications

In this section, we present a custom domain-specific editor action, an editor action that is meant to replace a built-in Idris IDE mode feature, and an editor action that improves on Idris's proof search mechanism in a specific logic.

4.1 Regular Expression Simplification

One of the most promising benefits of our work is that it allows authors of embedded domain-specific languages [23] (or eDSLs) to write domain-specific custom editor actions that assist eDSL users. One such language that most programmers are already familiar with is regular expressions.

```
data Regex = Empty
           | Epsilon
           | Lit Char
           | Concat Regex Regex
           | Or Regex Regex
           | Star Regex
```

Figure 11. Definition of regular expressions.

Mirroring the formal definition of regular expressions, the Idris definition of regexes has constructors for $\varnothing$, ε, literal characters, concatenation ($\cdot$), alternation ($|$) and the Kleene star ($*$), as seen in figure 11.

The most common usage of regular expressions is to determine whether a string is in its language. For that, users would have to write regular expression literals using the `Regex` data type. If the user wants to check whether the regular expression $a*$ accepts the string *"aaa"*, they would call `accepts (Star (Lit 'a')) "aaa"`.

However, there is no guarantee that the user would write the regular expression in its simplest form. Especially for more complex regular expressions, it is easy to overlook simpler versions. For instance, a user might write the regex term `Or Epsilon (Star (Lit 'a'))`, representing $\varepsilon|a*$, instead of `Star (Lit 'a')`, representing $a*$. A custom editor action can perform this simplification automatically.

For reasons of space, we will not explain the actual simplification algorithm, since there are many external sources such as Ortiz and Anaya [36] and Harper [21] that do, and it is not essential to understanding how the custom editor action works. We take the function `simplify : Regex -> Regex` as a given, and proceed to describe how it can be used to construct an editor action.[2]

The custom editor action to simplify regexes should consume a regex, returning a potentially simpler regex. However, when the editor sends expressions to the compiler, it sends them as strings containing snippets of code, which are then parsed and elaborated in the compiler. The editor action should therefore have the type `TT -> Elab TT`. The input and output are not regular expressions; they are ASTs of Idris code that represents regular expressions.

Simplification, however, is a function from `Regex` to `Regex`, not from `TT` to `TT`. At the same time, regular expression simplification is vastly easier to implement on actual `Regex`es, rather than on their quotations. To implement the editor action, figure 12 defines two functions `unquote` and `quote` that converts between `TT` and the constructors of `Regex`. The

[2]Full source code is available at http://github.com/joom/edit-time-tactics/tree/master/code/regex.

conversion from `Regex` to `TT` in the `quote` function cannot fail, since all regexes must have a core language representation. However, the conversion from `TT` to `Regex` can fail in the `unquote` function, since if `unquote` can be given any core language term, including ones that represent ill-typed Idris terms, ones that have types other than `Regex`, or `Regex`es that contain free variables.

The `unquote` and `quote` functions are boilerplate code that should eventually either be derived [10] or added as polymorphic primitives in the `Elab` monad that directly reflect the internal term, similar to Agda's `TC` monad [1]. We leave either of these for future work.

```
unquote : TT -> Elab Regex
unquote `(Empty) = pure Empty
unquote `(Epsilon) = pure Epsilon
unquote `(Lit ~c) = do c' <- unquote c
                       pure (Lit c')
unquote `(Concat ~x ~y) = do x' <- unquote x
                             y' <- unquote y
                             pure (Concat x' y')
unquote `(Or ~x ~y) = do x' <- unquote x
                         y' <- unquote y
                         pure (Or x' y')
unquote `(Star ~x) = do x' <- unquote x
                        pure (Star x')
unquote t = fail [{- elided -}]

quote : Regex -> TT
quote Empty = `(Empty)
quote Epsilon = `(Epsilon)
quote (Lit c) = `(Lit ~(quote c))
quote (Concat x y) =
  `(Concat ~(quote x) ~(quote y))
quote (Or x y) = `(Or ~(quote x) ~(quote y))
quote (Star x) = `(Star ~(quote x))
```

Figure 12. Functions to convert between values of the `Regex` type and their representation in the core language.

The functions `unquote`, `quote`, and `simplify` provide all the building blocks needed to define the custom editor action, which are combined in figure 13 to make up the necessary `Elab` action.

The Emacs Lisp support code required to run the custom editor action gets the region selected in the editor by the user, sends it as the first argument for the `simplifyInEditor` editor action, receives a response from the compiler, and replaces the code in the response with the selected region.

Figure 14 shows an example editor session using the regex simplification editor action. The user selects a region containing an expression and executes the Emacs Lisp function, which replaces the selected expression with the simplified version of the same regex.

```
%editor
simplifyInEditor : TT -> Elab TT
simplifyInEditor t =
  do r <- unquote t
     pure (quote (simplify r))
```

```
(defun idris-simplify-regex ()
  "Replace selection with simplified regex."
  (interactive)
  (let* ((regexp (buffer-substring-no-properties
                  (region-beginning)
                  (region-end)))
         (result (idris-elab-edit
                  "simplifyInEditor" regexp)))
    (replace-region
     (region-beginning) (region-end) result)))
```

Figure 13. Definition of `Elab` action for regex simplification, and the necessary Emacs Lisp support code to run.

```
if accepts (Or Epsilon (Star (Lit 'a'))) "aaa"
  then {- elided -} else {- elided -}

if accepts (Star (Lit 'a')) "aaa"
  then {- elided -} else {- elided -}
```

Figure 14. Before and after invoking regular expression simplification

4.2 Reimplementing the Built-In "Add Clause" Action

Idris's editor modes support a built-in editor action called "Add initial match clause to type declaration." When the cursor is on the type signature of a function that does not have any clauses, the user can run this editor action and get an initial pattern clause for the function. For instance, invoking the command on the declaration

```
copy : (n : Nat) -> a -> Vect n a
```

results in the clause

```
copy n x = ?copy_rhs
```

which has a bound variable for each explicit argument in `copy`'s type.

There is no longer any need to implement this feature in Haskell as part of the compiler. This section describes the implementation of an editor action in Idris itself that generates initial clauses for top-level type declarations without implicit arguments or interface constraints. A version that handles these additional features is longer, but involves no additional concepts. The complete Idris code that implements this editor action can be seen in figure 15.

```
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 <- getType n
     env <- getEnv
     ty' <- normalise env ty
     let (argTys, retTy) = collectTypes ty'
     argNames <- for argTys (const fresh)
     let lhsUntyped =
       foldl RApp (Var n) (map Var argNames)
     (lhsTyped, _) <- check env lhsUntyped
     holeName <- fresh
     let rhs = Bind holeName (GHole retTy) (V 0)
     pure (MkFunClause lhsTyped rhs)
```

Figure 15. Implementation of the editor action for "add clause".

The `collectTypes` function takes a type and dissects it into components, and returns a pair of the list of inputs and the output type. For instance, calling `collectTypes` with the input `` `(Nat -> Bool -> String) `` returns `([`(Nat), `(Bool)],`(String))`.

The `addClause` action only takes one input, which is the name of the function declaration for which an initial clause has been requested. Using this name, it looks up the type of that function, normalizes the type, and gets its components using `collectTypes`. The list of input types is named *argTys*, and the output type is named *retTy*. For each member of *argTys*, it generates a new user-accessible name using `fresh`. A more featureful implementation would attempt to preserve names from the type signature, only generating fresh names when the user had not provided a name or in the presence of shadowing.

A pattern match clause consists of a left-hand side, which is an application of the function being defined to either constructors or pattern variables, and a right-hand side, which is the expression that results when the pattern on the left-hand side matches. Having found names for each pattern variable, the left hand side of the initial clause is constructed by applying the function being defined, using `RApp`. The `Var` constructor injects names into terms. The right hand side of the initial clause should consist only of a hole for the user to fill in, indicated by the `GHole` term constructor. Because Idris holes are binding forms, the de Bruijn index `0` refers back to this new hole.

The Emacs Lisp code necessary to run `addClause` as a custom editor action is almost identical to the existing add-clause editor action. The only difference is that the call to the primitive add-clause editor action in the IDE protocol is replaced by a call to `addClause`.

Using this editor action on the declaration

```
copy : (n : Nat) -> a -> Vect n a
```

results in an initial match clause

```
copy a b = ?c
```

which was just as expected. However, this new version is much more readily extensible by users.

4.3 A Theorem Prover for Intuitionistic Propositional Logic

In this section, we describe the procedure Hezarfen,[3] which can decide intuitionistic propositional logic theorems, similar to Coq's `tauto` tactic. This procedure will be based on Dyckhoff's LJT [13] and its Haskell implementation Djinn [4], which generates Haskell expressions for a given type. Djinn is a standalone program that takes commands interactively, and when it generates an expression it prints it on the screen. Hezarfen, on the other hand, is a library that provides an `Elab` action that can be used as a tactic in proofs, and a custom editor action to be run when the built-in proof search mechanism does not suffice.

The prover consists of mutually recursive functions that try to break the goal type down into components, recursively finds terms that satisfy the components, and then glues them together based on the initial matched goal type.

Later in the prover there is also a term simplifier, similar to Haskell's pointfree style converter.[4] It performs η-reduction, removes unused **`let`** bindings, and similar simplification steps repeatedly until it reaches a fixed point. However, this simplifier is tailored for Hezarfen's proof terms; it is not general-purpose. The necessity of further work on a general purpose one is discussed in section 6.1.

```
comm : (a, b : Type) -> Either a b -> Either b a
comm = ?comm_impl

comm : (a, b : Type) -> Either a b -> Either b a
comm = \x, y => either Right Left
```

Figure 16. Before and after invoking Hezarfen

Figure 16 displays the results of executing this editor action on a hole: Hezarfen finds a term with the desired type in which `either` is a non-dependent eliminator for `Either`. Observe that the type of `comm` corresponds to the proposition

[3] The name is pronounced "has are fan", and it means polymath in Turkish. Source code is available at http://github.com/joom/hezarfen.
[4] http://hackage.haskell.org/package/pointfree

$(a \vee b) \rightarrow (b \vee a)$, and by finding the term, Hezarfen proves the proposition.

5 Related Work

Type-directed editing, metaprogramming, and extensible programming environments are not unique to Idris. Our work was inspired by a long tradition of empowering programmers to customize their tools.

5.1 In Lean

Lean [12] has a tactic metaprogramming system [14] that is similar to Idris's elaborator reflection. During a visit to Microsoft Research, Leonardo de Moura and the second author added support to Lean for *hole commands*, which are tactics that implement editor actions, but only in the context of a hole. They allow the contents of the hole to be transformed into an arbitrary string, which replaces the hole. Using the pretty-printing features of Lean's tactic system, terms can be placed in holes.

Because Lean's editor actions only work in the context of a hole, and can only take quoted terms as arguments, no custom Emacs Lisp is necessary to invoke them. The user simply right-clicks a hole, and a list of commands appears. In comparison to our custom editor action mechanism presented in our work, Lean's system is less expressive, but more convenient. 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. A system like Lean's hole commands could be implemented as a small extension to Idris's editor actions that allows them to be specially registered and imposes the same restrictions on the action types.

5.2 In Haskell

Template Haskell [39] is the primary 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. Unlike elaborator reflection, Template Haskell does not expose the general-purpose elaboration mechanisms (such as GHC's constraint solver) through `Q`. Template Haskell metaprograms generate only expressions and definitions.

Brian McKenna, however, has implemented a simplifier[5] for the output of Template Haskell and arranged for the simplified code to be inserted into Emacs automatically. With further development, this feature could eventually gain the expressive power of Idris editor actions.

5.3 In Agda

Lindblad and Benke [26] introduced a term search algorithm called Agsy that saves users' time by automating parts of a proof or program that are straightforward but tedious to write. Agsy is used regularly by Agda users. Kokke and Swierstra [24] used Agda's prior reflection system to define a new proof search mechanism in Agda itself. The Hezarfen editor action we discussed in section 4.3 is not as advanced as their `auto` function, yet in their paper, they suggested an IDE feature that replaces a call to their `auto` with the proof terms it generates. We generalized their suggestion to all `Elab` procedures, 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.

5.4 In Coq

Coq has a metaprogramming mechanism called `template-coq`[6] that is based on Malecha's term reification [29]. Recently, a typed version of this system was also introduced [3], making it easier to write reliable code that uses quotations. However, we are not aware of any work on using template metaprograms in Coq to write new features for the editor.

5.5 Other Languages

Not every new language is conceived of as being primarily a mapping from the set of strings to the disjoint union of machine code and error messages, with its users and tooling as an afterthought. Some are designed from the start with a customizable interactive environment in mind. This tradition dates back to early work on Lisp, particularly the Lisp machines and Interlisp-D [40], as well as Smalltalk [19]. These environments are highly customizable, but they do not allow users to continue to use their preferred editors. Idris now occupies a space between the total freedom of Smalltalk and a language such as Haskell for which editor support is an afterthought.

Racket is a language that focuses on the paradigm of *language-oriented programming* [15], in which problems are solved by first constructing the most appropriate language to solve them. One part of this process is extensible, metaprogrammable tooling, especially the DrRacket [17] IDE. For instance, Feltey et al. [16] demonstrate a quite concise implementation of a Java-like language, including refactoring tools. It is certainly possible to implement dependently typed languages in the Racket ecosystem: both Cur [6] and Pie [18] already exist, and the latter includes a simple editor action system that is presently extensible only in Racket but could support other languages as well.

Structured editors are an alternative means of interacting with a programming language. Alfa is a structural proof editor [20], descended from an earlier system called Alf [27, 28]. These structure editors are not, however, customizable using programs written in their type theories. Likewise, while Epigram [31] supported type-driven structured editing, it was

[5] http://hackage.haskell.org/package/th-pprint

[6] https://github.com/Template-Coq/template-coq

not extensible in itself. The ongoing Hazel project [34, 35] employs the tools of programming language theory to describe interactions with a type-aware structured editor; however, they have not yet reflected this language of interactions back into their object language.

6 Future Work

The story of dependently typed languages that can be reprogrammed in themselves is only just beginning. Further developments can increase the convenience and reliability of Idris's editor actions.

6.1 Proof Simplification

Christiansen and Brady [8] showed that elaborator reflection can be used as a tactic language for interactive theorem proving. It is possible to use `Elab` tactics to define custom editor actions and reuse existing proof automation efforts directly from the editor.

`Elab` tactics generate a proof term during elaboration, but the artifact is only a call to the tactic, which allows users to ignore the proof terms generated by the tactics. However gigantic or hideous the proof terms are, readers of the code will only see that the tactics satisfy the goal, while the proof term itself remains hidden. Many well-known proof automation procedures, such as Coq's `omega` [38], make use of this fact to hide large, complicated proof terms. However, when using `Elab` tactics to define custom editor actions, the result of the action is an expression that is visible to the user. Thus, brevity and readability are desirable qualities in the proof terms generated by those tactics. Requiring all tactic authors to simplify their own expressions qualities is burdensome, and it hampers the reuse of existing tactics. If there were a generic mechanism to simplify and minimize generated proof terms, and even write them in a way that makes use of dependent pattern matching, then existing tactics would become much more useful for implementing editor actions. Ideally, a finished program that was written with custom editor actions based on proof automation should be indistinguishable from one written without.

6.2 A Universe of Actions

Section 2.3 described how the Idris compiler checks whether all components of an editor action type have an instance of the `Editorable` type class. However, it is not necessary to implement this as an additional step during elaboration: it would suffice to encode the allowed types of editor actions as a universe à la Tarski [2]. The universe would include only those functions whose domains have `Editorable` instances and whose ranges are in the universe, as well as other types that have `Editorable` instances. Figure 17 demonstrates an implementation of this universe.

```
data Act : Type where
  Done : (a : Type) -> Editorable a => Act
  Arg : (a : Type) -> Editorable a =>
     (a -> Act) -> Act

actTy : Act -> Type
actTy (Done ty) = Elab ty
actTy (Arg ty f) = (v : ty) -> actTy (f v)
```

Figure 17. Universe encoding of types feasible to be treated as editor actions.

```
easy : actTy (Arg TTName (\n => Done TT))
easy n = {- elided, same as before -}
```

Figure 18. `easy` rewritten as a universe encoded editor action.

```
getTypes : actTy (Arg Nat (\n =>
                   Arg (Vect n TTName) (\_ =>
                     Done (Vect n (Maybe TT)))))
getTypes n v =
  for v (\n =>
    do l <- lookupTy n
       case l of
         [(_, _, ty)] => pure (Just ty)
         _ => pure Nothing)
```

Figure 19. A dependently typed editor action that would be possible with the universe encoding.

Figure 18 shows how the type of the `easy` editor action from figure 2 would change with this encoding. Observe that `actTy (Arg TTName (Done TT))` evaluates to `TTName -> Elab TT`, therefore the definition of `easy` does not have to change.

The most important outcome of this change would be the increase in the expressiveness of editor action types, enabling interesting dependent types to be used for editor actions. The current implementation rules out dependently typed editor actions, while this universe encoding would allow them. Figure 19 shows a hypothetical editor action that takes a vector of some length that contains function names and returns a vector of the *same* length that contains the types found for the function names.

However, writing editor actions with dependent data types would require writing more complex `Editorable` instances. Figure 20 shows the `Editorable` instance for length-indexed vectors, which uses lists to denote vectors and hence has to check if the lengths match in every deserialization. The problem of interoperability between indexed families and simple datatypes is known as *dependent interoperability*; Dagand et al. [11] provide a solution that could be adopted in Idris.

```
implementation Editorable a
           => Editorable (Vect n a) where
  fromEditor {a} {n} (SExpList l) =
    do l' <- traverse (fromEditor {a = a}) l
           <|> fail [{- elided -}]
       case decEq (length l') n of
         Yes pf =>
           pure (replace {P = \k => Vect k a}
                         pf (fromList l'))
         No _ => fail [{- elided -}]
  fromEditor _ = fail [{- elided -}]
  toEditor v = toEditor (toList v)
```

Figure 20. `Editorable` instance for length-indexed vectors.

6.3 Surface-Language Syntax

Editor actions presently accept and produce representations of `TT`, rather than high-level Idris, which greatly simplifies the implementation and maintenance of editor actions. For many applications, this does not matter, because the *meaning* of an expression is more important than how it is written. In some cases, however, this lack of expressive power might be a problem. For instance, it is presently impossible to define an editor action that converts a use of idiom brackets [32] into the equivalent **do**-notation, as both expressions have the same representation in the core language. In the future, it would be interesting to explore representations of the syntax of high-level Idris that are robust in the face of change and extension.

7 Conclusion

In this paper, 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 [8].

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. This demonstrates the value of directly reusing the compiler's implementations.

We have demonstrated editor actions such as simple proof searches and a DSL-specific action, as well as a demonstration of rewriting part of Idris in itself. We hope that Hezarfen will eventually be a better proof search than the built in action. We believe there is potential to replace even more of the built-in editor actions with custom editor actions written in Idris, such as case-splitting and lifting a hole into a lemma. We can also add new general editor actions such as renaming a binder, renaming a function within a file, pruning unused arguments in a function, and so forth.

As the library of elaborator actions grows, more building blocks will be available for custom editor actions. Even today, however, authors of libraries and DSLs can include custom editor actions with their packages, giving library and DSL authors access to power that was previously reserved for compiler implementors.

If we are serious about type-driven interactive programming, we need to give users the power to control not only their programming language, but also their programming environment. Idris's editor actions are one small step towards that goal.

Acknowledgments

We would like to thank Daniel R. Licata for his extensive support while advising the first author's M.A. thesis [25], on which this paper is largely based. The first author was generously supported by Wesleyan University. The second author began working on this project during a postdoctoral fellowship at Indiana University, where he was supported by the United States National Science Foundation grant 1540276. Galois, Inc. has generously supported further work. We would also like to thank Daniel P. Friedman, Cyrus Omar and Austin Tasato for their comments on our draft, and the anonymous reviewers for their constructive and helpful feedback.

References

[1] 2016. Changelog for Agda-2.5.1. http://hackage.haskell.org/package/Agda-2.5.1/changelog. (16 April 2016). Accessed: 2018-05-19.

[2] Thorsten Altenkirch and Conor McBride. 2003. Generic programming within dependently typed programming. In *Generic Programming*. Springer, 1–20.

[3] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Typed Template Coq–Certified Meta-Programming in Coq. In *The Fourth International Workshop on Coq for Programming Languages*.

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

[5] Eli Barzilay. 2006. *Implementing reflection in Nuprl*. Ph.D. Dissertation. Cornell University.

[6] William J Bowman. 2016. Growing a Proof Assistant. *Higher-Order Programming with Effects*.

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

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

[9] David Raymond Christiansen. 2014. Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection. In *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages*. ACM, 1.

[10] David Raymond Christiansen. 2016. *Practical Reflection and Metaprogramming for Dependent Types.* Ph.D. Dissertation. IT University of Copenhagen, Software and Systems Section.

[11] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. 2018. Foundations of dependent interoperability. *Journal of Functional Programming* 28 (2018), e9.

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

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

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

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

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

[17] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. 1999. Programming Languages As Operating Systems (or Revenge of the Son of the Lisp Machine). In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*. ACM, 138–147.

[18] Daniel P. Friedman and David Thrane Christiansen. 2018. *The Little Typer*. MIT Press, Cambridge, MA, USA.

[19] Adele Goldberg. 1984. *Smalltalk-80: The Interactive Programming Environment*. Addison Wesley.

[20] Thomas Hallgren. 1998. The proof editor Alfa. http://www.cse.chalmers.se/hallgren/Alfa. (1998).

[21] Robert Harper. 1997. Notes on Regular Expression Simplification. https://www.cs.cmu.edu/~fp/courses/97-212/handouts/regsimp.pdf. (1997). Accessed: 2017-06-06.

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

[23] Paul Hudak. 1996. Building Domain-specific Embedded Languages. *Comput. Surveys* 28, 4es, Article 196. https://doi.org/10.1145/242224.242477

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

[25] Joomy Korkut. 2018. *Edit-Time Tactics in Idris*. Master's thesis. Wesleyan University, Middletown, CT, USA.

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

[27] Lena Magnusson. 1994. *The implementation of ALF - a proof editor based on Martin-Löf's monomorhic type theory with explicit substitution*. Ph.D. Dissertation. Chalmers Tekniska Högskolan.

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

[29] Gregory Michael Malecha. 2014. *Extensible Proof Engineering in Intensional Type Theory*. Ph.D. Dissertation. Harvard University.

[30] Conor McBride. 2000. *Dependently typed functional programs and their proofs*. Ph.D. Dissertation. University of Edinburgh. College of Science and Engineering. School of Informatics.

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

[32] Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (2008), 1–13.

[33] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195.

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

[35] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *Summit on Advances in Programming Languages (SNAPL 2017)*.

[36] Alejandro AR Trejo Ortiz and Guillermo Fernández Anaya. 1998. Regular expression simplification. *Mathematics and computers in simulation* 45, 1-2 (1998), 59–71.

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

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

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

[40] Guy L. Steele, Jr. and Richard P. Gabriel. 1993. The Evolution of Lisp. In *The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*. ACM, New York, NY, USA, 231–270.