`slyce`

A dependently typed toy language
Types Languages and Compilers, Spring 2023

Raven Rothkopf (`rgr2124`)
Gregory Schare (`gs3072`)

May 12, 2023

Contents

1 Project Overview

For our project, we implemented an interpreter for the dependently typed language `slyce`[1]in Haskell. `slyce` is a pure language that features λ abstractions, let-expressions, if-then-else expressions, abstract data types, and pattern matching. In addition to user defined abstract data types, the language comes with several built in data types: namely `Unit`, `Bool`, and `Pair`. `slyce` is also equipped with a parser and exhaustive error checking for program debugging. To implement `slyce`, we referenced tutorials for several dependently typed languages [**?** **?**], most notably Stephanie Weirich's `pi-forall` tutorial [**?**].

Weirich's tutorial consists of 9 chapters, 8 of which we implemented in full. The only exception is `pi-forall`'s irrelevancy checking. Irrelevance is a feature that comes from the notion that with dependent types, components like type annotations and type arguments are only there for proofs and can be ignored during compilation. Polymorphic functions must behave the same regardless of the types they are operating on, so they do not depend on types. We skipped this section for the sake of time and with the comfort that the feature is an optimization, thus not critical to the core functionality of the type checker.

We came into this project knowing absolutely nothing about dependent types, and we have certainly learned a lot throughout this journey. While some mysteries of dependent type checking still remain (covered in Section **??**), we have come away from this project with a deep understanding of much of Weirich's implementation and a core intuition about her decisions.

2 Installation

To run the type checker, first compile `slyce` using `stack build`. Then, to run `slyce` on a source file, such as `hello.sly` located in the `examples/` subdirectory, execute the command

```
stack exec slyce -- -t ./examples/hello.sly
```

`-t` is an optional flag that prints the signatures of the checked definitions. There is also `-s` for printing the scanned tokens, and `-p` for printing the parsed module. If the file failed to scan, parse, or type check, `slyce` will

[1] `slyce` is a reference to the phrase "slice of pie", because pie sounds like Π, as in the Π-calculus, which is the language of dependent types. However, since our project is a simple toy implementation, it is a "slice of Π". Moreover, the "y" in the name looks like an upside-down λ, evoking how our language differs from the λ-calculus.

print an informative error message, complete with the source positions of where the error occurred.

3 `slyce` example programs

To give a taste for `slyce`, we first walk through a set of example programs that can be found in the `examples/` subdirectory of the `slyce` source code. These examples provide a concrete reference for `slyce`'s key features discussed throughout the rest of this report. While we do not explicitly cover the syntax of `slyce`, we hope that a pass over these examples, as well as the rest in the `examples/` subdirectory, serve as an extensive set of use cases. The full syntax is available in the `slyce` parser in `Parser.hs`.

3.1 `vec.sly`: The "Hello World" of dependent types

```
1   data Nat where
2       Zero,
3       Succ of (Nat).
4
5   zero = Zero.
6   one = Succ zero.
7   two = Succ one.
8   three = Succ two.
9
10  data Vec (a:U) (n:Nat) where
11      Nil of (n = Zero),
12      Cons of (m:Nat) (a) (Vec a m) (n = Succ m).
13
14  head : (a:U) -> (n:Nat) -> Vec a (Succ n) -> a.
15  head = \a. \n. \v.
16      match v with
17          | Cons m x xs -> x.
18
19  v : Vec Nat three.
20  v = Cons two one (Cons one two (Cons zero three Nil)).
```

The classic use case for dependent types is a vector parameterized by the type of its elements and its length. In this program, we showcase the use of data types, Π types, function declaration, and pattern matching.

The type-safe vector data type, `Vec` is indexed over a specific length using the dependent type system. The implementation defines two new data types: `Nat` and `Vec`. `Nat` represents natural numbers and is defined

recursively as either Zero or the successor of another Nat. `Vec` represents a vector of values of type `a`, which has type `U` – `slyce`'s type of types – that has a length of `n` elements.

The `Vec` data type has two constructors: `Nil` and `Cons`. `Nil` creates an empty vector with a length of zero, and `Cons` adds an element of type `a` to the beginning of an existing vector of length `m`, resulting in a new vector with the type same type and a length `Succ m`, i.e. the length of the tail incremented by one. The type of the `Cons` constructor includes a proposition that the length of the resulting vector is one more than the length of the input vector. This is not an argument to the constructor, but constraint on the other arguments.

The `head` function takes a `Vec` of length `Succ n` and returns its first element, which has type `a`. The function pattern matches on the input vector, using the `Cons` constructor to extract the head element. Since the vector has length `Succ n`, it cannot be empty, so this function is type safe: it cannot result in a runtime error, since passing it an empty vector is a compile-time type error.

As an example of how to use the vector data type constructors, we include a definition of a vector of type `Vec Nat three` (i.e., a vector of three natural numbers). In each invocation of `Cons`, the first argument is the length of the tail vector and the second argument is the element to prepend to the front of the vector.

3.2 `list.sly`: Lists

```
data List (t:U) where
    Nil,
    Cons of (x:t) (xs:List t).

map : (a:U) -> (b:U) -> (a -> b) -> List a -> List b.
map = \a. \b. \f. \l.
    match l with
        | Nil -> Nil
        | Cons x xs -> Cons (f x) (map a b f xs).

foldr : (a:U) -> (b:U) -> (a -> b -> b) -> b -> List a -> b
.
foldr = \a. \b. \f. \acc. \l.
    match l with
      | Nil -> acc
      | Cons x xs -> f x (foldr a b f acc xs).

any : List Bool -> Bool.
any = foldr Bool Bool or False.

all : List Bool -> Bool.
all = foldr Bool Bool and True.
```

This example demonstrates a small library of list functions. Lists are written as in Haskell: they are parameterized by the type of their elements, but do not include information about their length.

Due to parametric polymorphism, the polymorphic `map` and `fold` functions must take type arguments.

3.3 `largeelim.sly`: Propositional equality

```
1   not : Bool -> Bool.
2   not = \x. if x then False else True.
3
4   t : Bool -> U.
5   t = \b. if b then Unit else Bool.
6
7   bar : (y : Bool) -> t y.
8   bar = \b. if b then () else True.
9
10  x : Unit.
11  x = bar True.
12
13  y : Bool.
14  y = bar False.
15
16  z : (Unit = t True).
17  z = Refl.
18
19  w : (Bool = t False).
20  w = Refl.
```

This example demonstrates a simple use of dependent types to write an use a function, `bar`, whose output type depends on the value of its input.

The use of `bar` is shown in `x` and `y`, which have different types depending on what value was passed to `bar`.

Furthermore, we use propositional equality to demonstrate that `t True` really is equal to `Unit`, and respectively for `False` and `Bool`.

`Refl` is a language built-in proof of reflexivity, i.e., witness of an equality type that corresponds to a proposition that two propositions are equal. Since these equality propositions are true by definition of `t`, the program type checks. This works due to our implementation of definitional equality, which reduces terms to weak head normal form when checking if they are equivalent. See implementation for more details.

If we swapped the arguments to the types of `z` and `w`, the program would fail to type check, as it cannot prove that `t False` is equal to `Unit`... because it isn't!

3.4 `sym.sly`: Propositional equality

```
1     sym : (a:U) -> (x:a) -> (y:a) -> (x = y) -> y = x.
2     sym = \a. \x. \y. \pf. subst Refl by pf.
3
4     not : Bool -> Bool.
5     not = \x. if x then False else True.
6
7     false_is_not_true : False = (not True).
8     false_is_not_true = Refl.
9
10    not_true_is_false = sym Bool False (not True)
      false_is_not_true.
```

To elaborate further on propositional equality, we exhibit a function that proves the symmetric property of equality.

Given a proof, `pf`, that proposition `x` equals proposition `y` , `sym` returns a proof that `y = x`. It achieves this by using the built-in function `subst`, which in this case substitutes the reflexivity proof into the proof of equality, thus swapping the variables and returning a new proof.

The rest of the file is an example of using this symmetry prover to show that `False = not True` implies `not True = False`. This would fail to type check if we gave it an untrue equality proposition.

Propositional equality is a deep and subtle art which we have not fully explored or understood. This aspect of `slyce` connects us closely to dependently typed languages like Agda and Coq, which prominently feature the application of dependent types to logic and theorem proving.

4 Our approach

Our implementation process for `slyce` was iterative, and consisted of four main components when tackling any given feature.

1. Read `pi-forall`'s explanation of the specific feature we sought to implement
2. Attempt to implement the feature while referencing the explanation
3. Compare our approach with Weirich's source code
4. Test our additions with some examples and debug (a lot)

Following the style of the `pi-forall` tutorial, we began with a simple, core language without any bells and whistles. Our first iterations of `slyce` were without error messages or a parser. Thus, step **5** of the implementation process was originally primitive. We began with simply loading our modules into Haskell's `ghci` environment and testing from there. As we progressed farther in the tutorial and type checking features built in complexity, we added these tools in response to the need for convenience of testing and better error reporting for debugging. This process ensured that we deeply understood the foundation of our type checker before we began to add new, more involved features. It also allowed us to motivate our own development of the peripheral features of our language; since Weirich's tutorial focuses on only the type checker and a few other components, we had the freedom to adopt our own approach to the parser, pretty printer, error tracer, and datatypes.

The core theories behind dependent type checking in `slyce` largely mirror that of `pi-forall`. Weirich's tutorial covers the theory and leaves all the other features that must come with a usable dependent type checker up to interpretation. For that reason, we diverge the most from `pi-forall` in our parser and error tracing approaches, though upon reflection in Section **??**, we wish we had implemented our parser in the `pi-forall` style from the beginning due to fundamental differences between parser combinators and parser generators.

5 Background: Dependent types

The key feature showcased in `slyce` is dependent type checking.

Dependent types are one direction in Barendregt's lambda cube describing types that depend on terms. In a dependently typed language like `slyce`, little or no distinction is made between terms and types. Types may be parameterized not just by other types, but by terms as well. For instance, a type like `Vec (a:U) (n:Nat)` is parameterized by a type and a term, and specific instances of this type in a type signature may even call type constructors, data constructors, or functions that return types, such as `Succ (n:Nat)`.

The examples above provide a good overview of some of the key features of dependently typed languages. Some of the especially cool features of dependent types, which motivated us to explore this area, include type safety and a robust system for propositions as types.

Implementing dependent types requires the programmer to pay attention

to many subtle details that are not as relevant in other type systems such as System F. The fundamental difference is that because types may depend on terms, deciding whether two types (or terms) are equivalent for the purposes of type checking is highly non-trivial, and necessarily involves some amount of compile-time evaluation. Following Weirich's tutorial, we have accomplished this using weak head normal form reduction of terms/types.

Some other difficulties that may arise when implementing dependent types may include: unification, distinguishing constructors and functions, and many more possibilities to introduce parser ambiguities because the line between types and terms breaks down.

The subtleties of dependent types imply various approaches to their implementation. The implementation that follows has its own advantages and limitations. Regrettably, since we focused mainly on implementing Weirich's tutorial, we are unable to provide a detailed comparison between this and other approaches, nor are we confident to describe the theoretical underpinnings of our particular implementation and a proof of its properties. This would require a knowledge of dependent type theory that we simply do not possess. However, we hope that our future studies will expose us to the depth of dependent type theory. Implementing this language has allowed us to dip our toes in the water, and we are excited to dive in.

6 Bidirectional type system

The `slyce` type system makes use of *bidirectional typing*. A bidirectional type system splits up type rules into two categories of judgements: *inference* judgements and *checking* judgements. The following rules serve as a sketch for `slyce`'s core bidirectional type system, though they notably do not include rules for checking data constructors, equality types, and case expressions. To get a sense for these typing rules, please refer to [**?**].

6.1 Type Inference: $\Gamma \vdash a \Rightarrow A$

Type inference, $\Gamma \vdash a \Rightarrow A^2$, dictates that in the context Γ, we should infer that a term a has type A. The inference rules for `slyce` are outlined below. In many rules, type inference depends on type checking.

Judgements I-app and I-let make use of *definitional equality* to perform type inference. The explanation and implementation of this property can be found in section **??**.

Figure 1: Type inference rules for `slyce`

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \text{I-var} \qquad \frac{}{\Gamma \vdash \mathbf{U} \Rightarrow \mathbf{U}}\ \text{I-type} \qquad \frac{\Gamma \vdash A \Leftarrow \mathbf{U} \quad \Gamma, x : A \vdash B \Leftarrow \mathbf{U}}{\Gamma \vdash (x : A) \to B \Rightarrow \mathbf{U}}\ \text{I-Pi}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathbf{U} \quad \Gamma \vdash a \Leftarrow A}{\Gamma \vdash (a : A) \Rightarrow A}\ \text{I-ann} \qquad \frac{\Gamma \vdash a \Rightarrow A \qquad \Gamma, x : A, x = a \vdash b \Rightarrow B}{\Gamma \vdash\ \mathbf{let}\ x = a\ \mathbf{in}\ b \Rightarrow B[a/x]}\ \text{I-let}$$

$$\frac{\Gamma \vdash a \Rightarrow A \qquad \mathbf{whnf}\ A \rightsquigarrow (x : A_1) \to B \qquad \Gamma \vdash b \Leftarrow A_1}{\Gamma \vdash a\ b \Rightarrow B[b/x]}\ \text{I-app}$$

6.2 Type Checking: $\Gamma \vdash a \Leftarrow A$

Type checking, $\Gamma \vdash a \Leftarrow A$, makes use of information from the context Γ, like the types of top-level definitions, to look up the type of a and check that it matches the known type A. The checking rules for `slyce` are outlined below.

Figure 2: Type checking rules for `slyce`

$$\frac{\Gamma, x : A \vdash a \Leftarrow B}{\Gamma \vdash \lambda x.a \Leftarrow (x : A) \to B}\ \text{C-lambda} \qquad \frac{\Gamma \vdash a \Rightarrow A \qquad \Gamma, x : A, x = a \vdash b \Leftarrow B}{\Gamma \vdash \mathbf{let}\ x = a\ \mathbf{in}\ b \Leftarrow B}\ \text{C-let}$$

$$\frac{\Gamma \vdash x \Leftarrow \mathbf{Bool} \quad \Gamma, x = \mathbf{True} \vdash b_1 \Leftarrow A \qquad \Gamma, x = \mathbf{False} \vdash b_2 \Leftarrow A}{\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 \Leftarrow A}\ \text{C-if}$$

$$\frac{\Gamma \vdash z \Rightarrow (x : A_1\ *\ A_2) \qquad \Gamma, x : B_1, y : B_2, z = (x, y) \vdash b \Leftarrow B[(x, y)/z]}{\Gamma \vdash \mathbf{let}\ (x, y)\ =\ z\ \mathbf{in}\ b \Leftarrow B}\ \text{C-letpair}$$

Judgements C-let, C-letpair and C-if make use of *definitional equality* and *propositional equality* to type check. The explanation and implementation of these properties can be found in Sections **??** and **??**.

[2] We are following the style of Weirich's bidirectional type system: $\Rightarrow$ for type inference

7 Haskell implementation: code overview

7.1 Main.hs

The entrypoint to our type checker is Main.hs. Here, we parse the command line options and arguments, call the scanner, parser, and type checker, and print the results or report an error.

7.2 Ast.hs

The abstract syntax tree. This file contains data type definitions for all of the constructs in the language, along with some important typeclass instances and derivations so that we can use `Unbound` and other libraries on our language.

7.3 Context.hs

This file defines the monad, `TcMonad`, that we use to pass state through our type checker. The monad consists of a stack of monad transformers: `Unbound`'s freshness monad for name capture, a `ReaderT` for getting declarations and position information from the environment, an `ExceptT` for error handling and debugging messages, and `IO` at the bottom for printing the result.

This file also defines the helper functions which interact with `TcMonad`. These largely fall into two categories: functions which lookup a name in the environment, and functions which extend the environment with a new definition or type signature. These are used frequently for a variety of purposes throughout our type checker.

7.4 TypeCheck.hs

This is the core of our implementation and the largest file in our software. TypeCheck exports the functions for taking a module or an individual declaration and type checking it in a given context. These are called `typeCheckModule` and `typeCheckDecl` respectively. To do this, it makes use of one central function called `typeCheckTerm` and a number of helpers.

`typeCheckTerm` is a syntax-directed function that takes a term and, optionally, a type to check it against. If no type is given, it tries to infer the type. It has a case for every possible syntactic construct in the AST.

---

and $\Leftarrow$ for type checking. These can be swapped for $\uparrow$ and $\downarrow$ respectively when referring to the bidirectional style outline in the lecture notes.

When dealing with pattern matching, we make use of several helpers that convert a pattern to a declaration so that it can be added to the context. This allows us to check the bodies of case branches with the variables from the pattern in context. It is important to note that we have left exhaustivity checking unimplemented due to time constraints. This does not reduce the expressiveness of our language, but it would be a useful warning to report to the user, and a good exercise to implement.

Type checking type and data constructor applications requires checking the arguments against a "telescope" of formals that have type signatures or constraints that must be obeyed. Since this checking is complex, we define some helper functions and use those. One key aspect of these functions is that, because this is a dependent type system, each subsequent formal in a telescope may reference the names of previous formals. Hence the telescoping structure, wherein every time we successfully check an actual argument with a formal, we extend the context with this new definition of the name so that later formals in the telescope can use it.

Throughout the type checker, we use functions from the `Equality` module in order to compare two terms for equivalence, reduce terms to weak head normal form, unify two terms, and ensure that certain terms have certain types. See below for details.

7.5 Equality.hs

This file defines definitional equality for our type system. The key functions it exports are `equal`, for checking if two terms are definitionally equal; `whnf`, for reducing a term to weak head normal form; and `unify`, for unifying terms to make them equivalent. There are other functions defined here for various small conveniences.

`equal`, `whnf`, and `unify` are, like `typeCheckTerm`, all structured as casing over the various syntactic forms in the AST. While these were simple to implement, they harbor a depth and subtlety seen nowhere else in this type checker. We do not purport to fully understand how and why unification works. This is one place where we are unsure what effect a small change in the implementation might have. However, we understand that unification is used for creating declarations that unify (i.e. produce a single declaration that extends the context with a succint definition) patterns in pattern matching branches with the scrutinee, and constraints in data types with the arguments passed in to constructors.

7.6 Scanner.x and Parser.y

We implemented the scanner and parser in Alex and Happy, respectively. The parser uses a state monad that carries the names of type and data constructors so that we can use them in the type checker. Our initial idea when we implemented data types was to differentiate them from other variable names during parsing using this monad, but unfortunately Happy does not appear to support this feature.

Currently, there are many shift/reduce and reduce/reduce errors, mainly due to ambiguities surrounding the parsing of function application and constructor application. If we had more time, we would rewrite this as a parser combinator using `Parsec`.

7.7 PrettyPrint.hs

This file implements a very basic pretty printer for our language. There is nothing especially to note here.

8 Key features

We would like to highlight and discuss a few key features with an eye towards implementation.

8.1 Π types

The central feature of any dependent type system is the Π type. This is the type of functions where the output type may depend on the input type. In order to implement this, our `Pi` constructor in the AST takes a name and type for the input type and a term for the output type, where the name is bound in the output type.

To type check lambda abstractions, we must check the type of the body with the bound variable of the abstraction added to the context with the input type of the `Pi` type.

To type check function application, we must instantiate the `Pi` type of the function with the value being applied.

8.2 Equality types

With propositions as types, equality types are a very interesting feature of our language. Equality types merely represent that two . They are associated with two forms: `Refl` (a value of an equality type) and `Subst a`

`b`, which allows us to use equality proofs to transform arbitrary terms into desired equivalent terms.

When type checking `Refl`, we must verify that the types it claims are equal are in fact definitionally equal.

When type checking `Subst`, we verify that that proof passed in has equality type, and then we add new definitions to the context that allow the type checker to make use of equality to produce the desired new output type.

8.3 Flow sensitivity

Flow sensitivity is an optimization for eliminators like `If` that, instead of just simply type checking each branch of the `If`, makes use of the contextual information implied by the flow. Specifically, in `If`, when type checking the consequent branch, if the condition can be reduced to a variable, we add a new declaration to the context just for this branch that equates the condition with `True` (and respectively for the `False` case). This can help simplify the type checking of each branch.

We perform a similar optimization for the `LetPair` eliminator.

8.4 Data types

Data types are the feature that allows us to express more interesting constructs, proofs, and programs in our language.

The implementation of data types is briefly described above, but it is too complex to go into detail here. Please see our implementation, which contains many comments documenting the process of type checking data types.

In order to effectively use data types, we implement general dependent pattern matching via a `Match` expression, equivalent to Haskell's `case x of ...` expression. The implementation of this involves unifying each pattern with the reduced scrutinee in order to effectively type check each body of the match expression.

9 Discussion and reflections

Upon much reflection and experience with this tutorial, our main takeaway from this project is to *always trust Stephanie Weirich*! Many times throughout our development process, after checking our implementation of a feature with `pi-forall`, we would conclude that the subtle differences in our approaches were irrelevant. Later down the line–sometimes much later–we

would come to realize that those differences actually make or break the type checker.

A concrete example of this is the API call to the `Unbound.Generics.LocallyNameless` library's freshness monad to generate fresh names for our terms. In `pi-forall` and `slyce`, the `Unbound.unbind` is used to unbind term names from their bodies in λ expressions and Π types. When unbinding two different term names from their bodies and checking for propositional equivalence, `pi-forall` uses `Unbound.unbind2Plus` to unbind the two terms at the same time. After reading the documentation that was referred to in the tutorial on this call, we concluded that our original approach of unbinding the names separately was satisfactory. It was only after implementing extensive error messages, painstakingly tracing a program through the entire type checker, and hours of debugging when we realized that this decision was the source of very subtle naming error. `Unbound.unbind2Plus` unbinds two names and gives them the same fresh name, while `Unbound.unbind`ing separately results in different names, eventually causing an error when performing equivalence checking.

We suffered a similar comeuppance when, after initially deciding that it would be simpler to use the Happy LALR parser generator instead of using an LL parser combinator like Weirich, we found that our parser was unable to satisfactorily parse data types. This was the last chapter and the last thing we implemented, so it became a frantic search to find a way to distinguish constructors from function application and type constructors from data constructors. We settled on a solution that works for most cases, but due to time limitations, we are left with numerous shift/reduce and reduce/reduce errors that make our parser fragile; it frequently incorrectly parses function application. Had we more time, we would either reimplement our parser using parser combinators like Weirich does, or adopt the approach that SSLANG takes of using SYB's `everywhere` function to modify the abstract syntax tree after an initial parse.

One drawback of this reflection is that `pi-forall` is a delicate piece of software. It left very little room for experimentation or divergence from the source code, especially as we got further and further into the tutorial. This is not necessarily a criticism of the tutorial, as it certainly accomplishes its pedagogical goal and we learned a ton. This is more a reflection on the complexity and subtlety of dependent type theory. That being said, the difficulties we encountered forced us to really understand what was going on. If Weirich had included more documentation or explanation of some more arcane choices, we might not have come away with such a deep understanding of our own implementation.

The specificity of the `pi-forall` language and its quirks also rendered the many other dependent type checker tutorials unhelpful because little seemed to carry over to Weirich's approach. Even though many of the problems other tutorials tackle are the same, the approach tends to be completely different, and in many cases, we found ourselves unsure about whether small changes to the implementation would change important properties of the type system, such as making it unsound. We lack confidence in our ability to reason about how minor alterations to the type checker would have butterfly effects on the properties of our system.

For example, we struggled with the lack of documentation and relevance of the `Unbound.instantiate` call for substitution when performing reduction via weak head normal form. In our implementation, when performing substitution into the bodies of let-expressions and applications, we were first unbinding the binder and then performing the substitution. Weirich, on the other hand, uses the special `Unbound.instantiate` call which essentially combines these processes. Upon investigation, we discovered that Weirich herself implemented this feature into the library specifically for `pi-forall`, and `Unbound.instantiate` was just made available in the latest release of the library, though this tutorial was published last year.

Another example: early on, we figured there was no good reason that Weirich processes functions and signatures one at a time, as opposed to adding them all to the context at once. For simplicity, and because Weirich did not explain this choice in the tutorial or the code documentation, we opted for the latter. This ended up making our type system unsound, and a long session with ChatGPT helped us understand how we could detect unsoundness through examples. We ended up implementing an approach nearly identical to that of Weirich.

We believe that the process of "misconception to realization to correction" is a very valuable outcome of the project. The only way we could truly understand the value and inner workings of the implementation was to try it ourselves and then come to the conclusion that we were wrong, after re-examination. We would not have been able to conceptualize much of `pi-forall`'s source code without this time-consuming yet ultimately rewarding process.