Morpheus: Automated Safety Verification of Data-dependent Parser Combinator Programs

Ashish Mishra
Department of Computer Science, Purdue University, USA.

Suresh Jagannathan
Department of Computer Science, Purdue University, USA.

Abstract

Parser combinators are a well-known mechanism used for the compositional construction of parsers, and have shown to be particularly useful in writing parsers for rich grammars with data-dependencies and global state. Verifying applications written using them, however, has proven to be challenging in large part because of the inherently effectful nature of the parsers being composed and the difficulty in reasoning about the arbitrarily rich data-dependent semantic actions that can be associated with parsing actions. In this paper, we address these challenges by defining a parser combinator framework called Morpheus equipped with abstractions for defining composable effects tailored for parsing and semantic actions, and a rich specification language used to define safety properties over the constituent parsers comprising a program. Even though its abstractions yield many of the same expressivity benefits as other parser combinator systems, Morpheus is carefully engineered to yield a substantially more tractable automated verification pathway. We demonstrate its utility in verifying a number of realistic, challenging parsing applications, including several cases that involve non-trivial data-dependent relations.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Parsers, Verification, Domain-specific languages, Functional programming, Refinement types, Type systems.

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.5

Related Version https://doi.org/10.48550/arXiv.2305.07901

Funding Funding for this work was provided in part by DARPA under the SafeDocs program (grant HR0011-19-C-0073).

Acknowledgements The authors thank the reviewers for their insightful and useful comments.

1 Introduction

Parsers are transformers that decode serialized, unstructured data into a structured form. Although many parsing problems can be described using simple context-free grammars (CFGs), numerous real-world data formats (e.g., pdf [**?**], dns [**?**], zip [**?**], etc.), as well as many programming language grammars (e.g., Haskell, C, Idris, etc.) require their parser implementations to maintain additional context information during parsing. A particularly important class of context-sensitive parsers are those built from *data-dependent grammars*, such as the ones used in the data formats listed above. Such *data-dependent* parsers allow parsing actions that explicitly depend on earlier parsed data or semantic actions. Often, such parsers additionally use global effectful state to maintain and manipulate context information. To illustrate, consider the implementation of a popular class of *tag-length-data* parsers; these parsers can be used to parse image formats like PNG or PPM images, networking packets formats like TCP, etc., and use a parsed length value to govern the size of the input payload that should be parsed subsequently. The following BNF grammar captures this relation for a

simplified PNG image.

```
png ::= header . chunk*
chunk ::= length . typespec . content
```

The grammar defines a header field followed by zero or more chunks, where each chunk has a single byte length field parsed as an unsigned integer, followed by a single byte chunk type specifier. This is followed by zero or more bytes of actual content. A useful data-dependent safety property that any parser implementation for this grammar should satisfy is that "*the length of* content *plus* typespec *is equal to the value of* length".

Parser combinator libraries [**?**, **?**, **?**, **?**] provide an elegant framework in which to write parsers that have such data-dependent features. These libraries simplify the task of writing parsers because they define the grammar of the input language and implement the recognizer for it at the same time. Moreover, since combinator libraries are typically defined in terms of a shallowly-embedded DSL in an expressive host language like Haskell [**?**, **?**] or OCaml [**?**], parser implementations can seamlessly use a myriad of features available in the host language to express various kinds of data-dependent relations. This makes them capable of parsing both CFGs as well as richer grammars that have non-trivial semantic actions. Consequently, this style of parser construction has been adopted in many domains [**?**, **?**, **?**], a fact exemplified by their support in many widely-used languages like Haskell, Scala, OCaml, Java, etc.

Although parser combinators provide a way to easily write data-dependent parsers, verifying the correctness (i.e., ensuring that all data dependencies are enforced) of parser implementations written using them remains a challenging problem. This is in large part due to the inherently effectful nature of the parsers being composed, the pervasive use of rich higher-order abstractions available in the combinators used to build them, and the difficulty of reasoning about complex data-dependent semantic actions triggered by these combinators that can be associated with a parsing action.

This paper directly addresses these challenges. We do so by imposing modest constraints on the host language capabilities available to parser combinator programs; these constraints *enable* mostly automated reasoning and verification, *without* comprising the ability to specify parsers with rich effectful, data-dependent safety properties. We manifest these principles in the design of a deeply-embedded DSL for OCaml called Morpheus that we use to express and verify parsers and the combinators that compose them. Our design provides a novel (and, to the best of our knowledge, first) automated verification pathway for this important application class. This paper makes the following contributions:

1. It details the design of an OCaml DSL Morpheus that allows compositional construction of *data-dependent* parsers using a rich set of primitive parsing combinators along with an expressive specification language for describing safety properties relevant to parsing applications.
2. It presents an automated refinement type-based verification framework that validates the correctness of Morpheus programs with respect to their specifications and which supports fine-grained effect reasoning and inference to help reduce specification annotation burden.
3. It justifies its approach through a detailed evaluation study over a range of complex real-world parser applications that demonstrate the feasibility and effectiveness of the proposed methodology.

The remainder of the paper is organized as follows. The next section presents a detailed motivating example to illustrate the challenges with verifying parser combinator applications and presents a detailed overview of Morpheus that builds upon this example. We formalize

```
1  decl ::= typedef . type−expr . id=rawident
2       | extern ...
3       | ...
4  typename ::= rawident
5  type−exp ::= "int" | "bool"
6  expr ::= ... | id=rawident
```

```
1  decl ::= typedef . type−expr . id=rawident [¬
       id ∈ (!identifiers)]
2            {types.add id}
3         | ...
4  typename ::= x = rawident [x ∈ (!types)]{
       return x}
5  type−exp ::= "int" | "bool"
6  expr ::= ... | id=rawident {identifiers.add id ;
       return id}
```

Figure 1 Context-free and context-sensitive grammars for C declarations.

Morpheus's specification language and type system in Secs. **??** and **??**. Details about Morpheus's implementation and benchmarks demonstrate the utility of our framework is given in Sec. **??**. Related work and conclusions are given in Secs. **??** and **??**, respectively.

2 Motivation and Morpheus Overview

To motivate our ideas and give an overview of Morpheus, consider a parser for a simplified C language *declarations, expressions and typedefs* grammar. The grammar must handle context-sensitive disambiguation of *typenames* and *identifiers* [1]. Traditionally, C-parsers achieve this disambiguation via cumbersome *lexer hacks*[2] which use feedback from the symbol table maintained in the parsing into the lexer to distinguish variables from types. Once the disambiguation is outsourced to the lexer-hack, the C-decl grammar can be defined using a context-free-grammar. For instance, the left hand side, Figure **??**, presents a simplified context-free grammar production for a C declaration.

Unfortunately, ad-hoc lexer-hacks are both tedious and error prone. Further, this convoluted integration of the lexing and parsing phases makes it challenging to validate the correctness of the parser implementation. A cleaner way to implement such a parser is to disambiguate *typenames* and *identifiers* when parsing by writing an actual context-sensitive parser. One approach would be to define a shared *context* of two non-overlapping lists of types and identifiers and a stateful-parser using this context. The modified *context-sensitive* grammar is shown in right hand side, Figure **??**.

The square brackets show context-sensitive checks e.g. [¬ id ∈ (!identifier)] checks that the parsed rawident token id is not in the list of identifiers, while the braces show semantic actions associated with parser reductions, e.g. {typed.add id}, adds the token id to types, a list of identifiers seen thus far in the parse.

Given this grammar, we can use parser combinator libraries [**?**, **?**] in our favorite language to implement a parser for C language declarations. Unfortunately, although cleaner than the using unwieldy lexer hacks, it is still not obvious how we might verify that implementations actually satisfy the desired *disambiguation* property, i.e. *typenames* and *identifiers* do not overlap. In the next section we provide an overview of Morpheus that informally presents our solution to this problem.

[1] https://web.archive.org/web/20070622120718/
http://www.cs.utah.edu/research/projects/mso/goofie/grammar5.txt
[2] https://www.lysator.liu.se/c/ANSI-C-grammar-l.html

```
1  let ids = ref []
2  let types = ref []
3  type decl =
4       Typedecl of {typeexp;string}
5       | ...
6  type expression =
7       Address of expression
8       | Cast of string * expression
9       | ...
10      | Identifier of string
11
```

```
expression :
PE^stexc
  {∀ h,
  ldisjoint (sel (h, ids),sel (h, types)) = true)}
    ν :  expression result
  {∀ h, ν, h'.ν = Inl (v1) =>
    ldisjoint (sel (h', ids),sel (h', types)) = true)
    ∧ ν = Inr (Err) => included(inp,h,h') = true }
```

```
12  let expression =
13   do_m char '('
14      tn ← typename
15      char ')'
16      e ← expression
17      return Cast (tn, e))
18   <|> ...
19   <|>
20   do_m
21      id ← identifier
22      let b = List.mem id !types
23      if (!b) then
24         ids := id :: (!ids)
25         return (Identifier id)
26      else
27           fail
```

```
28
```

```
typedecl :
PE^stexc
{∀ h,
ldisjoint (sel (h, ids),sel (h, types)) = true) }
     ν :  tdecl result
{∀ h, ν, h'.ν = Inl (v1) =>
ldisjoint (sel (h', ids),sel (h', types)) = true)
  ∧ ν = Inr (Err) =>  included(inp,h,h') = true }
```

```
29  let typedecl =
30    do_m
31      td ← keyword "typedef"
32      te ← string "bool" <|> string "int"
33      id ← indentifier
34      (* incorrect-check: if (not(List.mem id
              !types)) then*)
35      if (not (List.mem id !ids)) then
36           types := id :: (!types)
37           return Tdecl {typeexp; id}
38      else
39           fail
40
```

```
typename :
PE^stexc
{∀ h.
ldisjoint (sel (h, ids),sel (h, types)) = true}
     ν :  string result
{∀ h, ν, h'.ν = Inl (v) =>
mem (sel (h', types), v) = true
∧ ν = Inr (Err) =>  included(inp,h,h') = true}
```

```
41  let typename =
42      do_m
43        x ← identifier
44        if (List.mem x !types) then
45             return x
46        else
47             fail
```

Figure 2 A simplified C-declaration parser written in Morpheus. Specifications in blue are provided by the programmer; specifications in gray are inferred by Morpheus.

2.1 Morpheus Surface Language

An important design decision we make is to provide a surface syntax and API very similar to conventional monadic parser combinator libraries like Parsec [**?**] in Haskell or mParser [**?**] in OCaml; the core API that Morpheus provides has the signature shown in Figure **??**. The library defines a number of primitive combinators: eps defines a parser for the empty language, bot always fails, and char c defines a parser for character c. Beyond these, the library also provides a bind (>>=) combinator for monadically composing parsers, a choice (<|>) combinator to non-deterministically choose among two parsers, and a fix combinator to implement recursive parsers. The return x is a parser which always succeeds with a value x. As we demonstrate, these combinators are sufficient to derive a number of other useful parsing actions such as many, count, etc. found in these popular combinator libraries. From the parser writer's perspective, Morpheus programs can be expressed using these combinators along with a basic collection of other non-parser expression forms similar to those found in an ML core language, e.g., first-class functions, let expressions, references, etc.

```
type 'a t
val eps : unit t
val bot : 'a t
val char : char → char t
val (>>=) : 'a t → (a → 'b t) → '
      b t
val <|> : 'a t → 'a t → 'a t
val fix : ('b t → 'b t) → 'b t
val return : 'a → 'a t
```

Figure 3 Signatures of primitive parser combinators supported by Morpheus.

For instance a parser for option p, which either parses an empty string or anything that p parses can be written:

```
let option p = (eps >>= λ_. return None) <|> (p >>= λ x. return Some x)
```

We can also define more intricate parsers like *Kleene-star* and *Kleene-plus*:

```
let star p = fix (λ p_star. eps <|> p >>= λ x. p_star >>= λ xs . return (x :: xs) )
let plus p = fix (λ p_star. p <|> p >>= λ x. p_star >>= λ xs . return (x :: xs) )
```

Figure **??** shows a Morpheus implementation that parses a valid C language decl.[3] The parser uses two mutable lists to keep track of types and identifiers. The structure is similar to the original data-dependent grammar, even though the program uses ML-style operators for assignment and dereferencing. For ease of presentation, we have written the program using *do-notation* as syntactic sugar for Morpheus's monadic bind combinator.

The typedecl parser follows the grammar and parses the keyword *typedef* using the keyword parser (not shown).[4] It uses a choice combinator (<|>) (line **??**), which has a semantics of a non-deterministic choice between two sub-parsers. The interesting case occurs while parsing an identifier (lines **??** - **??**), in order to enforce disambiguation between *typenames* and *identifiers*, the parser needs to maintain an invariant that the two lists, types for parsed *typenames* and ids for parsed *identifiers* are always *disjoint* or *non-overlapping*.

In order to maintain the non-overlapping list invariant, a parsed identifier token (line **??**) can be a valid typename only if it is not parsed earlier as an identifier expression. i.e. it is not in the *ids* list. The parser performs this check at (line **??**). If this check succeeds, the list of typenames (*types*) is updated and a decl is returned, else the parsing fails.

[3] For now, ignore the specifications given in gray and blue.
[4] Morpheus, like other parser combinator libraries provides a library of parsers for parsing keywords, identifiers, natural numbers, strings, etc.

The disambiguation decision is required during the parsing of an expression. The expression parser defines multiple choices. The parser for the *casting* expression parses a typename followed by a recursive call to expression. The typename parser in turn (line **??**) parses an identifier token and checks that the identifier is indeed a typename (line **??**) and returns it, or fails.

The ids list is updated during parsing an identifier expression (line **??**), here again to maintain disambiguation, before adding a string to the ids list, its non-membership in the current types list is checked (line **??**).

Although the above parser program is easy to comprehend given how closely it hews to the grammar definition, it is still nonetheless non-trivial to verify that the parser actually satisfies the required disambiguation safety property. For example, an implementation in which line **??** is replaced with the commented expression above it would incorrectly check membership on the wrong list. We describe how Morpheus facilitates verification of this program in the following section.

2.2 Specifying Data-dependent Parser Properties

Intuitively, verifying the above-given parser for the absence of overlap between the *typenames* and *identifiers* requires establishing the following partial correctness property: if the types and identifiers lists do not overlap when the typedecl parser is invoked, and the parser terminates without an error, then they must not overlap in the output state generated by the parser. Additionally, it is required that the parser consumes some prefix of the input list. Morpheus provides an expressive specification language to specify properties such as these.

Morpheus allows standard ML-style inductive type definitions that can be refined with *qualifiers* similar to other refinement type systems [**?**, **?**, **?**]. For instance, we can refine the type of a list of strings to only denote *non-empty* lists as: type nonempty = { ν : [string] | len (ν) > 0 }. Here, ν is a special bound variable representing a list and (len ν > 0) is a *refinement* where len is a *qualifier*, a predicate available to the type system that captures the length property of a list.

2.2.1 Specifying effectful safety properties

Standard refinement type systems, however, are ill-suited to specify safety properties for effectful computation of the kind expressible by parser combinators. Our specification language, therefore, also provides a type for effectful computations. We use a specification monad (called a *Parsing Expression*) of the form $\mathsf{PE}^{\varepsilon}\ \{\ \phi\ \}\ \nu : \tau\ \{\ \phi'\ \}$ that is parameterized by the *effect* of the computation ε (e.g., state, exc, nondet, and their combinations like stexc for (both state and exc), stnon (for both state and nondet), etc.); and Hoare-style pre- and post-conditions [**?**, **?**, **?**]. Here, ϕ and ϕ' are first-order logical propositions over qualifiers applied to program variables and variables in the type context. The precondition ϕ is defined over an abstract input heap h while the postcondition ϕ' is defined over input heap h, output heap h', and the special result variable ν that denotes the result of the computation. Using this monad, we can specify a safety property for the typedecl subparser as shown at line **??** in Figure **??**. The type should be understood as follows: The *effect* label stexc defines that the parser may have both state effect as it reads and updates the context; and exc effect as the parser may fail. The precondition defines a property over a list of identifiers ids and a list of typenames types in the input heap h via the use of the built-in qualifier sel that defines a select operation on the heap [**?**]; here, ν is bound to the result of the parse. Morpheus also allows user-defined qualifiers, like the qualifier Idisjoint. It establishes the

disjointness/non-overlapping property between two lists. This qualifier is defined using the following definition:

```
qualifier ldisjoint [] l2 → true
                  | l1 [] → true
                  | (x :: xs) l2 → member (x, l2) = false ∧ ldisjoint (xs, l2)
                  | l1 (y :: ys) → member (y, l1) = false ∧ ldisjoint (l1, ys)
```

This definition also uses another qualifier for list membership called member. Morpheus automatically translates these user-defined qualifiers to axioms, logical sentences whose validity is assumed by the underlying theorem prover during verification. For instance, given the above qualifier, Morpheus generates axioms like:

```
Axiom1: ∀ l1, l2 : α list. (empty(l1) ∨ empty (l2)) => ldisjoint (l1, l2) = true
Axiom2: ∀ xs, l2: α list, x : α. ldisjoint (xs, l2) = true ∧ member (x, l2) = false => ldisjoint ((x::
    xs), l2) = true
Axiom3: ∀ l1, l2: α ldisjoint (l1, l2) <=> ldisjoint (l2, l1)
```

The specification (at line **??**) also uses another qualifier, included(inp,h,h'), which captures the monotonic consumption property of the input list inp. The qualifier is true when the remainder inp after parsing in h' is a suffix of the original inp list in h.

The types for other parsers in the figure can be specified as shown at lines **??**, **??**, etc.; these types shown in gray are automatically inferred by Morpheus's type inference algorithm. For example, the type for the typename parser (line **??**) returns an optional string (result is a special option type) and records that when parsing is successful, the returned string is added to the types list, and when unsuccessful, the input is still monotonically consumed.

2.2.2 Verification using Morpheus

Note that the pre-condition in the specification (ldisjoint (Id, Ty) = true)) and the type ascribed to the membership checks in the implementation (line **??**) are sufficient to conclude that the addition of a typename to the types list (line **??**) maintains the ldisjoint invariant as required by the postcondition.

In contrast, an erroneous implementation that omits the membership check or replaces the check at line **??** with the commented line above it will cause type-checking to fail. The program will be flagged ill-typed by Morpheus. For this example, Morpheus generated 21 verification conditions (VCs) for the control-path representing a successful parse and generated 5 VCs for the failing branch. We were able to discharge these VCs to the SMT solver Z3 [**?**], which took 6.78 seconds to verify the former and 1.90 seconds to verify the latter.

3 Morpheus Syntax and Semantics

3.1 Morpheus Syntax

Figure **??** defines the syntax of λ_{sp}, a core calculus for Morpheus programs. The language is a call-by-value polymorphic lambda-calculus with effects, extended with primitive expressions for common parser combinators and a refinement type-based specification language. A λ_{sp} value is either a constant drawn from a set of base types (int, bool, etc.), as well as a special Err value of type exception, an abstraction, or a constructor application. Variables bound to updateable locations (ℓ) are distinguished from variables introduced via function binding (x). A λ_{sp} expression e is either a value, an application of a function or type abstraction,

operations to dereference and assign to top-level locations (see below), polymorphic **let** expressions, reference binding expressions, a **match** expression to pattern-match over type constructors, a **return** expression that lifts a value to an effect, and various parser primitive expressions that define parsers for the empty language (eps), a character (char) parser, and $\perp$, a parser that always fails. Additionally, the language provides combinators to monadically compose parsers (>>=), to implement parsers defined in terms of a non-deterministic choice of its constituents ($< \mid >$), and to express parsers that have recursive ($\mu\,(\mathsf{x} : \tau).p$) structure.

We restrict how effects manifest by requiring reference creation to occur only within **let** expressions and not in any other expression context. Moreover, the variables bound to locations so created (ℓ) can only be dereferenced or assigned to and cannot be supplied as arguments to abstractions or returned as results since they are not treated as ordinary expressions. This stratification, while arguably restrictive in a general application context, is consistent with how parser applications, such as our introductory example are typically written and, as we demonstrate below, do not hinder our ability to write real-world data-dependent parser implementations. Enforcing these restrictions, however, provides a straightforward mechanism to prevent aliasing of effectful components during evaluation, significantly easing the development of an automated verification pathway in the presence of parser combinator-induced computational effects.

3.2 Semantics

Figure **??** presents a big-step operational semantics for λ_{sp} parser expressions; the semantics of other terms in the language is standard. The semantics is defined via an evaluation relation ($\Downarrow$) that is of the form $(\mathcal{H}; e) \Downarrow (\mathcal{H}'; v)$. The relation defines how a Morpheus expression e evaluates with respect to a heap $\mathcal{H}$, a store of locations to base-type values, to yield a value v, which can be a normal value or an exceptional one, the latter represented by the exception constant Err, and a new heap $\mathcal{H}'$.

The empty string parser (rule P-EPS) always succeeds, returning a value of type **unit**, without changing the heap. A "bottom" ($\perp$) parser on the other hand always fails, producing an exception value, also without changing the heap. If the argument e to a character parser char yields value (a char 'c'), and 'c' is the head of the input string (denoted by inp) being parsed, the parse succeeds (rule P-CHAR-TRUE), consuming the input and returning 'c', otherwise, the parse fails, with the input not consumed and the distinguished Err value being returned (rule P-CHAR-FALSE). The fixpoint parser $\mu\, x.p$ (P-FIX) allows the construction of recursive parser expressions. The monadic bind parser primitive (rule P-BIND-SUCCESS) binds the result of evaluating its parser expression to the argument of the abstraction denoted by its second argument, returning the result of the evaluating the abstraction's body (P-BIND-SUCCESS); the P-BIND-ERR rule deals with the case when the first expression fails. Evaluation of "choice" expressions, defined by rules P-CHOICE-L and P-CHOICE-R, introduce an unbiased choice semantics over two parsers allowing non-deterministic choices in parsers.

4 Typing λ_{sp} Expressions

4.1 Specification Language

The syntax of Morpheus's type system is shown in the bottom of Figure **??** and permits the expression of *base types* such as integers, booleans, strings, etc., as well as a special heap type to denote the type of abstract heap variables like h, h' found in the specifications described below. There are additionally user-defined datatypes TN (list, tree, etc.), a special sum type (t result) to define two options of a successful and exceptional result respectively, and a

Expression Language

$$
\begin{array}{llll}
c, \mathbf{unit}, \mathsf{Err} & \in & \textit{Constants} & \\
x & \in & \textit{Vars} & \\
\mathsf{inp}, \ell & \in & \textit{RefVars} & \\
v & \in & \textit{Value} & ::= \; c \mid \lambda\,(x:\tau).\,e \mid \Lambda\,(\alpha).\,e \mid \mathsf{D_i}\,\overline{t_k}\,\overline{v_j} \\
e & \in & \textit{Exp} & ::= \; v \mid x \mid p \mid e\;x \mid e\,[\mathsf{t}] \mid \mathsf{deref}\,\ell \mid \ell := e \\
 & & & \mid \mathbf{let}\,x = v\,\mathbf{in}\;e \mid \mathbf{let}\,\ell = \mathsf{ref}\,e\,\mathbf{in}\;e \\
 & & & \mid \mathbf{match}\,v\,\mathbf{with}\;\mathsf{D_i}\,\overline{\alpha}\,\overline{x_j} \to e \mid \mathbf{return}\;e \\
p & \in & \textit{Parsers} & ::= \; \mid \mathsf{eps} \mid \bot \mid \mathsf{char}\,e \mid (\mu\,(x:\tau).\,p) \\
 & & & \mid p >\!\!>\!= e \mid p <\!|\!> p
\end{array}
$$

Specification Language

$$
\begin{array}{llll}
\alpha & \in & \textit{TypeVariables} & \\
\mathsf{TN} & \in & \textit{User Defined Types} & ::= \; \alpha\,\mathsf{list}, \alpha\,\mathsf{tree}, \ldots \\
\mathsf{t} & \in & \textit{Base Types} & ::= \; \alpha \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \mathsf{heap} \mid \mathsf{TN} \mid \mathsf{t} \\
 & & & \mathsf{result} \mid \mathsf{t\,ref} \mid \mathsf{exc} \\
\tau & \in & \textit{Type} & ::= \; \{\nu : \mathsf{t} \mid \phi\} \mid (x:\tau) \to \tau \mid \mathsf{PE}^{\varepsilon}\{\phi_1\}\nu : \mathsf{t}\,\{\phi_2\} \\
\varepsilon & \in & \textit{Effect Labels} & ::= \; \mathsf{pure} \mid \mathsf{state} \mid \mathsf{exc} \mid \mathsf{nondet} \mid \ldots \\
\sigma & \in & \textit{Type Scheme} & := \; \tau \mid \forall\alpha.\,\tau \\
Q & \in & \textit{Qualifiers} & := \; \mathit{QualifierName}(\overline{x_i}) \\
\phi, P & \in & \textit{Propositions} & ::= \; \mathsf{true} \mid \mathsf{false} \mid Q \mid Q_1 = Q_2 \\
 & & & \mid \neg\,\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \forall(x:\mathsf{t}).\phi \\
\Gamma & \in & \textit{Type Context} & ::= \; \varnothing \mid \Gamma,\, x:\sigma \mid \Gamma,\, \ell : \tau\;\mathsf{ref} \mid \Gamma, \phi \\
\Sigma & \in & \textit{Constructors} & ::= \; \varnothing \mid \Sigma,\, \mathsf{D_i}\;\overline{\alpha_k}\;\overline{x_j:\tau_j} \to \tau
\end{array}
$$

Figure 4 λ_{sp} Expressions and Types

special exception type. More interestingly, base types can be refined with *propositions* to yield monomorphic refinement types. Such types [**?**, **?**, **?**] are either *base refinement types*, refining a base typed term with a refinement; *dependent function types*, in which arguments and return values of functions can be associated with types that are refined by propositions; or a *computation type* specifying a type for an effectful computation.

Effectful computations are refined using an effect specification monad:

$$\mathsf{PE}^{\varepsilon}\ \{\forall\,\mathsf{h}.\phi_1\}\,\nu : \mathsf{t}\,\{\forall\,\mathsf{h}, \nu, \mathsf{h}'.\phi_2\}$$

that encapsulates a base type t, parameterized by an effect label ε, with Hoare-style pre- ($\{\forall\,\mathsf{h}.\phi_1\}$) and post- ($\{\forall\,\mathsf{h}, \nu, \mathsf{h}'.\phi_2\}$) conditions. This type captures the behavior of a computation that (a) when executed in a pre-state with input heap h satisfies proposition ϕ_1; (b) upon termination, returns a value denoted by ν of base type t along with output heap h'; (c) satisfies a post-condition ϕ_2 that relates h, ν, and h'; and (d) whose effect is over-approximated by effect label ε [**?**, **?**]. An effect label ε is either (i) a pure effect that records an effect-free computation; (i) a state effect that signifies a stateful computation over the program heap; (ii) an exception effect exc that denotes a computation that might trigger an

$$\boxed{(\mathcal{H}; e) \Downarrow (\mathcal{H}'; v)}$$

$$\text{P-EPS}\ \frac{}{(\mathcal{H}; \mathsf{eps}) \Downarrow (\mathcal{H}; ())} \qquad \text{P-}\bot\ \frac{}{(\mathcal{H}; \bot) \Downarrow (\mathcal{H}; \mathsf{Err})} \qquad \text{P-FIX}\ \frac{(\mathcal{H}; [\mu \mathsf{x} : \sigma.p/\mathsf{x}]p)) \Downarrow (\mathcal{H}'; v)}{(\mathcal{H}; \mu \mathsf{x} : \sigma.p) \Downarrow (\mathcal{H}'; v)}$$

$$\text{P-CHAR-TRUE}\ \frac{(\mathcal{H}; e) \Downarrow (\mathcal{H}; \text{`}c\text{'}) \qquad \mathcal{H}(\mathsf{inp}) = (\text{`}c\text{'} :: \mathsf{s}) \qquad \mathcal{H}' = \mathcal{H}[\mathsf{inp} \mapsto s]}{(\mathcal{H}; \mathsf{char}\ e) \Downarrow (\mathcal{H}'; \text{`}c\text{'})}$$

$$\text{P-CHAR-FALSE}\ \frac{(\mathcal{H}; e) \Downarrow (\mathcal{H}; \text{`}c\text{'}) \qquad \mathcal{H}(\mathsf{inp}) \neq (\text{`}c\text{'} :: s) \qquad \mathcal{H}' = \mathcal{H}[\mathsf{inp} \mapsto \mathsf{inp}]}{(\mathcal{H}; \mathsf{char}\ e) \Downarrow (\mathcal{H}'; \mathsf{Err}))}$$

$$\text{P-BIND-SUCCESS}\ \frac{(\mathcal{H}; p) \Downarrow (\mathcal{H}'; v_1) \quad (\mathcal{H}'; e) \Downarrow (\mathcal{H}'; (\lambda\, x\ :\ \tau.\, e')) \qquad (\mathcal{H}'; [\ v_1/x]e') \Downarrow (\mathcal{H}''; v_2)}{(\mathcal{H}; p \gg\!= e) \Downarrow (\mathcal{H}''; v_2)}$$

$$\text{P-BIND-ERR}\ \frac{(\mathcal{H}; p) \Downarrow (\mathcal{H}'; \mathsf{Err})}{(\mathcal{H}; p \gg\!= e) \Downarrow (\mathcal{H}'; \mathsf{Err}))}$$

$$\text{P-CHOICE-L}\ \frac{(\mathcal{H}; p_1) \Downarrow (\mathcal{H}'; v_1)}{(\mathcal{H}; (p_1 <\!|\!> p_2)) \Downarrow (\mathcal{H}'; v_1))} \qquad \text{P-CHOICE-R}\ \frac{(\mathcal{H}; p_2) \Downarrow (\mathcal{H}''; v_2)}{(\mathcal{H}; (p_1 <\!|\!> p_2)) \Downarrow (\mathcal{H}''; v_2))}$$

Figure 5 Evaluation rules for λ_{sp} parser expressions

exception; (iii) a nondet effect that records a computation that may have non-deterministic behavior; or (iv) a *join* over these effects that reflect composite effectful actions. The need for the last is due to the fact that effectful computations are often defined in terms of a composition of effects, e.g. a parser oftentimes will define a computation that has a state effect along with a possible exception effect. To capture these composite effects, base effects can be joined to build a finite lattice that reflects the behavior of computations which perform multiple effectful actions, as we describe below.

Propositions (ϕ) are first-order predicate logic formulae over base-typed variables. Propositions also include a set of qualifiers which are applications of user-defined uninterpreted function symbols such as mem, size etc. used to encode properties of program objects, sel used to model accesses to the heap, and dom used to model membership of a location in the heap, etc. Proposition validity is checked by embedding them into a decidable logic that supports equality of uninterpreted functions and linear arithmetic (EUFLIA).

A type scheme (σ) is either a monotype (τ) or a universally quantified polymorphic type over type variables expressed in prenex-normal form ($\forall\ \alpha.\sigma$). A Morpheus specification is given as a type scheme.

There are two environments maintained by the Morpheus type-checker: (1) an environment Γ records the type of variables, which can include variables introduced by function abstraction as well as bindings to references introduced by let expressions, along with a set of propositions relevant to a specific context, and (2) an environment Σ maps datatype constructors to their signatures. Our typing judgments are defined with respect to a typing environment

$$\Gamma ::= . \mid \Gamma, \mathsf{x} : \sigma \mid \Gamma, \ell\colon \tau \ \mathsf{ref}$$

that is either empty, or contains a list of bindings of variables to either type schemes or references. The rules have two judgment forms: $(\Gamma \vdash e : \sigma)$ gives a type for a Morpheus expression e in Γ; and $(\Gamma \vdash \sigma_1 <: \sigma_2)$ defines a dependent subtyping rule under Γ.

Since our type expressions contain refinements, we generalize the usual notion of type substitution to reflect substitution within refined types:

$$\begin{aligned}
[x_a/x]\{\nu : \mathsf{t}|\phi\} &= \{\nu : \mathsf{t}|[x_a/x]\phi\} \\
[x_a/x](y : \tau) \to \tau' &= (y : [x_a/x]\tau) \to [x_a/x]\tau', y \neq x \\
[x_a/x]\mathsf{PE}^{\mathcal{E}}\{\phi_1\}\{\nu : \mathsf{t}\}\{\phi_2\} &= \mathsf{PE}^{\mathcal{E}}\{[x_a/x]\phi_1\}\{\nu : \mathsf{t}\}\{[x_a/x]\phi_2\}
\end{aligned}$$

4.2 Typing Base Expressions

Figure **??** presents type rules for non-parser expressions. The type rules for non-reference variables, functions, and type abstractions (T-TYP-FUN) are standard. The syntax for function application restricts its argument to be a variable, allowing us to record the argument's (intermediate) effects in the typing environment when typing the application as a whole.

The type rule for the return expression (T-RETURN) lifts its non-effectful expression argument e to have a computation effect with label pure, thereby allowing e's value to be used in contexts where computational effects are required; a particularly important example of such contexts are bind expressions used to compose the effects of constituent parsers.

In the constructor application rule (T-CAPP), the expression's type reflects the instantiation of the type and term variables in the constructor's type with actual types and terms. A match expression is typed (rule T-MATCH) by typing each of the alternatives in a corresponding extended environment and returning a *unified type*. The pre-condition of the *unified* type is a conjunction of the pre-conditions for each alternative, while the post-condition over-approximates the behavior for each alternative by creating a disjunction of each of the possible alternative's post-conditions. Location manipulating expressions (T-DEREF and T-ASSIGN) use qualifiers sel and dom to define constraints that reflect state changes on the underlying heap. The argument ℓ of a dereferencing expression (rule T-DEREF) is associated with a computation type over a tref base type. Its pre-condition requires ℓ to be in the domain of the input heap, and its post-condition establishes that ℓ's contents is the value returned by the expression and that the heap state does not change. The assignment rule (T-ASSIGN) assigns the contents of a top-level reference ℓ to the non-effectful value yielded by evaluating expression e. The pre-condition of its computation effect type requires that ℓ is in the domain of the input heap and that ℓ's contents in the output heap satisfies the refinement (ϕ) associated with its r-value. Finally, rule T-REF types a **let** expression that introduces a reference initialized to a value v. The body is typed in an environment in which ℓ is given a computational effect type. The pre-condition of this type requires that the input heap, i.e., the heap extant at the point when the binding of ℓ to ref v occurs, not include ℓ in its domain; its postcondition constrains ℓ's contents to be some value ν' that satisfies the refinement ϕ associated with v, its initialization expression. The body of the **let** expression is then typed in this augmented type environment.

4.3 Typing Parser Expressions

Figure **??** presents the type rules for Morpheus parser expressions. The (T-SUB) rule defines the standard type subsumption rule. The empty string parser typing rule (T-P-EPS) assigns a type with pure effect and unit return type, while the postcondition establishes the equivalence

Base Expression Typing $\boxed{\Gamma \vdash e : \sigma}$

$$\text{T-VAR}\frac{\Gamma(\mathsf{x}) = \sigma}{\Gamma \vdash \mathsf{x} : \sigma} \quad \text{T-FUN}\frac{\Gamma, \mathsf{x} : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(\mathsf{x} : \tau_1).e \ : \tau_1 \to \tau_2} \quad \text{T-TYPAPP}\frac{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\sigma}{\Gamma \vdash \Lambda\alpha.e[\mathsf{t}] : [\mathsf{t}/\alpha]\sigma}$$

$$\text{T-APP}\frac{\Gamma \vdash e_f : (x \ : \ \{\nu : \mathsf{t} \mid \phi_x\}) \to \mathsf{PE}^{\mathcal{E}}\{\phi\} \ \nu \ : \mathsf{t} \ \{\phi'\} \qquad \Gamma \vdash x_a : \{\nu : \mathsf{t} \mid \phi_x\}}{\Gamma \vdash e_f \ x_a \ : [x_a/x]\mathsf{PE}^{\mathcal{E}}\{\phi\} \ \nu \ : \mathsf{t} \ \{\phi'\}}$$

$$\text{T-TYPFUN}\frac{\Gamma \vdash \mathsf{e} : \sigma \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda\alpha.\mathsf{e} : \forall\alpha.\sigma} \quad \text{T-LET}\frac{\Gamma \vdash v \ : \forall\alpha.\sigma \qquad \Gamma, x \ : \forall\alpha.\sigma \vdash e_2 : \sigma'}{\Gamma \vdash \mathbf{let}\ x \ = \ v\ \mathbf{in}\ e_2 \ : \ \sigma'}$$

$$\text{T-RETURN}\frac{\Gamma \vdash e \ : \{\nu : \mathsf{t} \mid \phi\}}{\Gamma \vdash \mathbf{return}\ e : \mathsf{PE}^{\mathsf{pure}}\{\forall \mathsf{h}.\mathsf{true}\}\ \nu : \ \mathsf{t} \ \{\forall \mathsf{h}, \nu, \mathsf{h}'.\mathsf{h}' = \mathsf{h} \wedge \phi\}}$$

$$\text{T-CAPP}\frac{\Sigma(D_i) = \forall\overline{\alpha_k}.\overline{\mathsf{x_j} : \tau_j} \to \tau \qquad \forall i, j.\Gamma \vdash \mathsf{v}_j : [\overline{\mathsf{t}_k/\alpha_k}][\overline{\mathsf{v_j}/\mathsf{x_j}}]\tau_j}{\Gamma \vdash D_i \ \overline{\mathsf{t}_k}\overline{\mathsf{v}_j} : [\overline{\mathsf{t}/\alpha}][\overline{\mathsf{v_j}/\mathsf{x_j}}]\tau}$$

$$\text{T-MATCH}\frac{\begin{array}{c}\Sigma(D_i) = \forall\overline{\alpha_k}.\overline{\mathsf{x_j} : \tau_j} \to \tau_0 \\ \Gamma \vdash v : \tau_0 \qquad \Gamma_i = \Gamma, \overline{\alpha_k}, \overline{\mathsf{x}_j : \tau_j} \\ \Gamma_i \vdash (\mathsf{D_i}\ \overline{\alpha_k \mathsf{x}_j}) : \tau_0 \qquad \Gamma_i \vdash e_i : \mathsf{PE}^{\mathcal{E}}\{\phi_i\}\ \nu \ : \mathsf{t}\ \{\phi_{i'}\}\end{array}}{\begin{array}{c}\Gamma \vdash \mathbf{match}\ v\ \mathbf{with}\ \mathsf{D_i}\ \overline{\alpha_k \mathsf{x}_j} \to e_i : \\ \mathsf{PE}^{\mathcal{E}}\{\forall\ \mathsf{h}.\bigwedge_i (v \ = \mathsf{D_i}\ \overline{\alpha_k \mathsf{x}_j}) \Rightarrow \phi_i\}\ \nu \ : \mathsf{t}\ \{\forall\ \mathsf{h}, \nu', \mathsf{h}'.\bigvee_i \phi_{i'}\}\end{array}}$$

$$\text{T-DEREF}\frac{\Gamma \vdash \ell \ : \mathsf{PE}^{\mathsf{state}}\{\phi_1\}\ \nu \ : \mathsf{t}\ \mathsf{ref}\ \{\phi_2\}}{\Gamma \vdash \mathsf{deref}\ \ell : \mathsf{PE}^{\mathsf{state}}\{\forall\ \mathsf{h}.\mathsf{dom}(\mathsf{h}, \ell)\}\ \nu' \ : \mathsf{t}\ \{\forall\ \mathsf{h}, \nu', \mathsf{h}'.\mathsf{sel}(\mathsf{h}, \ell) \ = \nu' \wedge \mathsf{h} = \mathsf{h}'\}}$$

$$\text{T-ASSIGN}\frac{\Gamma \vdash \ e \ : \ \{\nu \ : \ \mathsf{t} \mid \ \phi\}}{\Gamma \vdash \ell := e \ : \mathsf{PE}^{\mathsf{state}}\{\forall \mathsf{h}.\mathsf{dom}(\mathsf{h}, \ell)\}\ \nu' \ : \ \mathsf{t}\ \{\forall\ \mathsf{h}, \nu', \mathsf{h}'.\mathsf{sel}(\mathsf{h}', \ell) \ = \ \nu' \wedge \ \phi(\nu')\}}$$

$$\text{T-REF}\frac{\begin{array}{c}\Gamma \ \vdash \ v \ : \ \{\ \nu \ : \ \mathsf{t} \mid \ \phi\ \} \\ \Gamma, \ell \ : \mathsf{PE}^{\mathsf{state}}\{\forall\ \mathsf{h}.\neg\ \mathsf{dom}(\mathsf{h}, \ell)\}\ \nu' \ : \mathsf{t}\ \mathsf{ref}\ \{\forall\ \mathsf{h}, \nu', \mathsf{h}'.\mathsf{sel}(\mathsf{h}', \ell) \ = \\ v \ \wedge \phi(v) \wedge \ \mathsf{dom}(\mathsf{h}', \ell)\} \vdash \ \mathsf{e}_b \ : \mathsf{PE}^{\mathcal{E}}\{\mathsf{dom}(\mathsf{h}, \ell)\}\ \nu'' \ : \mathsf{t}\ \{\phi'_b\}\end{array}}{\begin{array}{c}\Gamma \ , \mathsf{h_i} : \mathsf{heap} \vdash \ \mathbf{let}\ \ell \ = \mathsf{ref}\ \ v\ \mathbf{in}\ \mathsf{e}_b \ : \\ \mathsf{PE}^{\mathcal{E} \sqcup \mathsf{state}}\{\forall\ \mathsf{h}.\neg\ \mathsf{dom}(\mathsf{h}, \ell)\}\ \nu'' \ : \mathsf{t}\ \{\forall\ \mathsf{h}, \nu'', \mathsf{h}.\mathsf{dom}(\mathsf{h_i}, \ell) \wedge \mathsf{sel}(\mathsf{h_i}, \ell) \ = \ v \ \wedge \phi(v) \wedge \ \phi'_b\}\end{array}}$$

Figure 6 Typing Semantics for Morpheus Base Expressions

Parser Expression Typing $\boxed{\Gamma \vdash e : \sigma}$

$$\text{T-SUB}\ \frac{\Gamma \vdash e : \sigma_1 \qquad \Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash e : \sigma_2}$$

$$\text{T-P-EPS}\ \frac{}{\Gamma \vdash \mathsf{eps} : \mathsf{PE}^{\mathsf{pure}}\ \{\forall \mathsf{h}.\ \mathsf{true}\}\ \nu : \mathsf{unit}\ \{\forall \mathsf{h}, \nu, \mathsf{h}'.\mathsf{h}' = \mathsf{h}\}}$$

$$\text{T-P-BOT}\ \frac{}{\Gamma \vdash \bot : \mathsf{PE}^{\mathsf{exc}}\ \{\forall \mathsf{h}.\ \mathsf{true}\}\ \nu : \mathsf{exc}\ \{\forall \mathsf{h}, \nu, \mathsf{h}'.\mathsf{h}' = \mathsf{h} \wedge \nu = \mathsf{Err}\}}$$

$$\text{T-P-CHAR}\ \frac{\begin{array}{c}\Gamma \vdash e : \{\nu' : \mathsf{char} \mid \nu' = \text{`c'}\} \\ \phi_2 = \forall \mathsf{h}, \nu, \mathsf{h}'.\forall \mathsf{x}, \mathsf{y}. \\ (\mathsf{Inl}(\mathsf{x}) = \nu \implies \mathsf{x} = \text{`c'} \wedge \mathsf{upd}(\mathsf{h}', \mathsf{h}, \mathsf{inp}, \mathsf{tail}(\mathsf{inp}))) \wedge \\ (\mathsf{Inr}(\mathsf{y}) = \nu \implies \mathsf{y} = \mathsf{Err} \wedge \mathsf{sel}(\mathsf{h}, \mathsf{inp}) = \mathsf{sel}(\mathsf{h}', \mathsf{inp}))\end{array}}{\Gamma \vdash \mathsf{char}\ e : \mathsf{PE}^{\mathsf{state} \sqcup \mathsf{exc}}\{\forall \mathsf{h}.\mathsf{true}\}\ \nu\ :\ \mathsf{char}\ \mathsf{result}\ \{\phi_2\}}$$

$$\text{T-P-CHOICE}\ \frac{\Gamma \vdash p_1\ :\ \mathsf{PE}^{\varepsilon}\ \{\phi_1\}\ \nu_1\ :\ \tau\ \{\phi_1'\} \qquad \Gamma \vdash p_2\ :\ \mathsf{PE}^{\varepsilon}\ \{\phi_2\}\ \nu_2\ :\ \tau\ \{\phi_2'\}}{\Gamma \vdash (p_1 <|> p_2) : \mathsf{PE}^{\varepsilon \sqcup \mathsf{nondet}}\ \{(\phi 1 \wedge \phi_2)\}\ \nu : \tau\ \{(\phi_1' \vee \phi_2')\}}$$

$$\text{T-P-FIX}\ \frac{\Gamma, \mathsf{x} : (\mathsf{PE}^{\varepsilon}\ \{\phi\}\ \nu\ :\ \mathsf{t}\ \{\phi'\}) \vdash\ p : \mathsf{PE}^{\varepsilon}\ \{\phi\}\ \nu\ :\ \mathsf{t}\ \{\phi'\} \qquad \mathsf{x} \notin FV(\phi, \phi')}{\Gamma \vdash \mu \mathsf{x}\ :\ (\mathsf{PE}^{\varepsilon}\ \{\phi\}\ \nu\ :\ \mathsf{t}\ \{\phi'\}).\ p\ :\ \mathsf{PE}^{\varepsilon}\ \{\phi\}\ \nu\ :\ \mathsf{t}\ \{\phi'\}}$$

$$\text{T-P-BIND}\ \frac{\begin{array}{c}\Gamma\ \vdash\ p\ :\ \mathsf{PE}^{\varepsilon}\ \{\phi_1\}\ \nu\ :\ \mathsf{t}\{\phi_{1'}\} \qquad \Gamma\ \vdash\ e\ :\ (x : \tau) \to \mathsf{PE}^{\varepsilon}\ \{\phi_2\}\ \nu' : \mathsf{t}'\ \{\phi_{2'}\} \\ \Gamma' = \Gamma, x : \tau, \mathsf{h_i} : \mathsf{heap} \quad \mathsf{h_i}\ \text{fresh}\end{array}}{\begin{array}{c}\Gamma' \vdash p \gg= e : \\ \mathsf{PE}^{\varepsilon}\ \{\forall \mathsf{h}.\ \phi_1\ \mathsf{h}\ \wedge \phi_{1'}(\mathsf{h}, x, \mathsf{h_i}) \Rightarrow \phi_2\ \mathsf{h_i}\} \\ \nu'\ :\ \mathsf{t}'\ \mathsf{result} \\ \{\forall \mathsf{h}, \nu', \mathsf{h}', \mathsf{y}.\ (x \neq \mathsf{Err} \Rightarrow \nu' = \mathsf{Inl}\ \mathsf{y} \wedge \phi_{1'}(\mathsf{h}, x, \mathsf{h_i}) \wedge \phi_{2'}(\mathsf{h_i}, \mathsf{y}, \mathsf{h}')) \wedge \\ (x = \mathsf{Err} \Rightarrow \nu' = \mathsf{Inr}\ \mathsf{Err} \wedge \phi_{1'}(\mathsf{h}, x, \mathsf{h_i}))\}\end{array}}$$

Subtyping $\boxed{\Gamma \vdash \sigma_1 <: \sigma_2}$

$$\text{T-Sub-Base}\ \frac{\Gamma \vdash \{\nu : \mathsf{t} \mid \phi_1\} \qquad \Gamma \vdash \{\nu : \mathsf{t} \mid \phi_2\} \qquad \Gamma \vDash \phi_1 \Rightarrow \phi_2}{\Gamma \vdash \{\nu : \mathsf{t} \mid \phi_1\} <: \{\nu : \mathsf{t} \mid \phi_2\}}$$

$$\text{T-Sub-Schema}\ \frac{\Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash \forall \alpha.\sigma_1 <: \forall \alpha.\sigma_2}$$

$$\text{T-Sub-Arrow}\ \frac{\Gamma \vdash \tau_{21} <: \tau_{11} \qquad \Gamma \vdash \tau_{12} <: \tau_{22}}{\Gamma \vdash (\mathsf{x} : \tau_{11}) \to \tau_{12} <: (\mathsf{x} : \tau_{21}) \to \tau_{22}}$$

$$\text{T-Sub-TVar}\ \frac{}{\Gamma \vdash \alpha <: \alpha}$$

$$\text{T-Sub-Comp}\ \frac{\Gamma \vDash \phi_2 \Rightarrow \phi_1 \qquad \Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash \varepsilon_1 \leq \varepsilon_2 \qquad \Gamma, \phi_2 \vDash (\phi_{1'} \Rightarrow \phi_{2'})}{\Gamma \vdash \mathsf{PE}^{\varepsilon_1}\ \{\phi_1\}\ \tau_1\ \{\phi_{1'}\} <: \mathsf{PE}^{\varepsilon_2}\ \{\phi_2\}\ \tau_2\ \{\phi_{2'}\}}$$

Figure 7 Typing semantics for primitive parser expressions and subtyping rules.

of the input and the output heaps. The T-P-BOT rule captures the always failing semantics of $\perp$ with an exception effect exc and corresponding return types and return values while maintaining the stability of the input heap. The type rules governing a character parser (T-P-CHAR) is more interesting because it captures the semantics of the success and the failure conditions of the parser. We use a sum type (α result) to define two options representing a successful and exceptional result, resp. (with the Err exception value in the latter case), using standard injection functions to differentiate among these alternatives. In the successful case, the returned value is equal to the consumed character, captured by an equality constraint over characters. In the successful case, the structure of the output heap with respect to the parse string inp must be the same as the input heap except for the absence of the 'c', the now consumed head-of-string character. In the failing case, the input remains unconsumed. Note that we also join the effect labels (state $\sqcup$ exc), highlighting the state and exception effect. These effect labels form a standard join semi-lattice with an ordering relation ($\leq$) [5].

Rule T-P-CHOICE defines the static semantics for a non-deterministic choice parser. It introduces a non-determinism effect to the parser's composite type. The effect's precondition requires that either of the choices can occur; we achieve this by restricting it to the conjunction of the two preconditions for the sub-parsers. The disjunctive post-condition requires that both the choices must imply the desired goal postcondition for a composite parser to be well-typed. The effect for the choice expression takes a join over the effects of the choices and the non-deterministic effect.

Rule (T-P-FIX) defines the semantics for the terminating recursive fix-point combinator. Given an annotated type τ for the parameter x, if the type of the body p in an extended environment which has x mapping to τ, is τ, then τ is also a valid type for a recursive fixpoint parser expression. The T-P-BIND rule defines a typing judgement for the exceptional monadic composition of a parser expression p with an abstraction e. The composite parser is typed in an extended environment (Γ') containing a binding for the abstraction's parameter x and an intermediate heap $\mathsf{h_i}$ that acts as the output/post-state heap for the first parser and the input/pre-state for the second. The relation between these heaps is captured by the inferred pre-and post-conditions for the composite parser. There are two possible scenarios depending upon whether the first parser p results in a success (i.e. x $\neq$ Err) or a failure (x = Err).

In the successful case, the inferred conditions capture the following properties: a) the output of the combined parser is a success; b) the post-condition for the first expression over the intermediate heap $\mathsf{h_i}$ and the output variable x should imply the precondition of the second expression (required for the evaluation of the second expression); and, c) the overall post-condition relates the post-condition of the first with the precondition of the second using the intermediate heap $\mathsf{h_i}$. The case when p fails causes the combined parser to fail as well, with the post-condition after the failure of the first as the overall post-condition. Note that the core calculus is sub-optimal in size since λ_{sp} supports both return and eps, even though the latter could be modeled using return. However, this design choice enables decidable typechecking by limiting the combination of higher-order functions, combinators and states. This is achieved using a limited bind p >>= e, rather than the general e >>= e, allowing for the definition of semantic actions e that only perform limited state manipulation, i.e., reading and updating locations. Thus >>= and < | > only take parser arguments; thus, eps <|> p is not equivalent to (return () <|> p), in fact the latter is disallowed. Another such design restriction shows up in the typing rules, e.g., the typing rule for function application

[5] Details of the effect-labels and their join semi-lattice is provided in the accompanied technical report [**?**]

(T-APP) restricts the arguments to be of *basetype*, thus disallowing expressions returning abstractions or computations, like return (λx. e) or return (x := e) A more general definition for >>= will allow valid HO arguments, like λx. e »= e1, but translating such general HO stateful programs to decidable logic fragments is not always feasible, as is discussed in other fully dependent type systems [**?**].

The subtyping rules enable the propagation of refinement type information and relate the subtyping judgments to logical entailment. The subtyping rule for a base refinement (T-Sub-Base) relates subtyping to the logical implication between the refinement of the subtype and the supertype. The (T-Sub-Arrow) rule defines subtyping between two function refinement types. The (T-Sub-Comp) rule for subtyping between computation types follows the standard Floyd-Hoare rule for *consequence*, coupled with the subtyping relation between result types and an ordering relation between effects($\leq$). The subtyping rule for type variables (T-Sub-TVar) relates each type variable to itself in a reflexive way, while the subtyping for a type-schema lifts the subtyping relation from a schema to another schema.

4.4 Properties of the Type System

▶ **Definition 1** (Environment Entailment $\Gamma \models \phi$)**.** *Given* $\Gamma = \ldots, \overline{\phi_i}$*, the entailment of a formula* ϕ *under* Γ *is defined as* $(\bigwedge_i \phi_i) \implies \phi$

In the following, $\Gamma \models \phi(\mathcal{H})$ extends the notion of semantic entailment of a formula over an abstract heap $\Gamma \models \phi$ (h) to a concrete heap using an interpretation of concrete heap $\mathcal{H}$ to an abstract heap h and the standard notion of well-typed *stores* ($\Gamma \vdash \mathcal{H}$).[6]

To prove soundness of Morpheus typing, we first state a soundness lemma for pure expressions (i.e. expressions with non-computation type).

▶ **Lemma 2** (Soundness Pure-terms)**.** *If* $\Gamma \vdash e : \{\nu \ : \ \mathsf{t} \mid \phi \ \}$ *then:*

- Either e is a value with $\Gamma \models \phi \ (\ e)$
- OR Given there exists a v, such that $(\mathcal{H};\ e) \Downarrow (\mathcal{H};\ \mathsf{v})$ then $\Gamma \vdash \mathsf{v} : \mathsf{t}$ and $\Gamma \models \phi\ (\mathsf{v})$

▶ **Theorem 3** (Soundness Morpheus)**.** *Given a specification* $\sigma = \forall\overline{\alpha}.\ \mathsf{PE}^{\varepsilon}\ \{\phi_1\}\ \nu : \mathsf{t}\ \{\phi_2\}$ *and a Morpheus expression* e*, such that under some* Γ*,* $\Gamma \vdash e$*:* σ*, then if there exists* $\mathcal{H}$ *such that* $\Gamma \models \phi_1(\mathcal{H})$ *then:*

1. Either e is a value, and: $\Gamma, \phi_1 \models \phi_2\ (\mathcal{H},\ e,\ \mathcal{H})$
2. Or, if there exists an $\mathcal{H}'$ and v such that $(\mathcal{H};\ e) \Downarrow (\mathcal{H}';\ \mathsf{v})$, then
   $\exists\ \Gamma'$, $\Gamma \subseteq \Gamma'$ and (consistent $\Gamma\ \Gamma'$), such that:
   a. $\Gamma' \vdash \mathsf{v} : \mathsf{t}$.
   b. $\Gamma', \phi_1\ (\mathcal{H}) \models \phi_2\ (\mathcal{H},\ \mathsf{v},\ \mathcal{H}')$

where (consistent $\Gamma\ \Gamma'$) is a Boolean-valued function that ensures that $\forall$ x $\in$ (dom (Γ) $\cap$ dom (Γ')). $\Gamma \vdash \mathrm{x} : \sigma \implies \Gamma' \vdash \mathrm{x} : \sigma$. Additionally, $\forall\phi.\ \Gamma \models \phi \implies \Gamma' \models \phi$.

Proof. The soundness proof is by induction on typing rules in Figures **??** and **??**, proving the soundness statement against the evaluation rules in Figures **??**.[7] ◀

[6] Details are provided in the accompanied technical report [**?**].
[7] The **decidability** theorem and proofs for all the theorems are provided in the technical report [**?**].

5 Evaluation

5.1 Implementation

Morpheus is implemented as a deeply-embedded DSL in OCaml[8] equipped with a refinement-type based verification system. It encodes the typing rules given in Section **??** and a parser translating an OCaml-based surface language of the kind presented in our motivating example to the Morpheus core, described in Section **??**. To allow Morpheus programs to be easily used in an OCaml development, its specifications can be safely erased once the program has been type-checked. Note that a Morpheus program, verified against a safety specification is guaranteed to be safe when erased since verification takes place against a stricter memory abstraction; in particular, since Morpheus programs are free of aliasing by construction and thus remain so when evaluated as an ML program. This obviates the need for a separate interpreter/compilation phase and gives Morpheus-verified parsers efficiency comparable to the parsers written using OCaml parser-combinator libraries [**?**, **?**].

Morpheus specifications typically require meaningful qualifiers over inductive data-types, beyond those discussed in our core language; in addition to the qualifiers discussed previously, typical examples include qualifiers to capture properties such as the length of a list, membership in a list, etc. Morpheus provides a way for users to write simple inductive propositions over inductive data types, translating them to axioms useful for the solver, in a manner similar to the use of *measures* and *predicates* in other refinement type works [**?**, **?**]. For example, a qualifier for capturing the length property of a list can be written as:

```
qualifier len [] → 0 | len (x :: xs) → len (xs) + 1.
```

Morpheus generates the following axiom from this qualifier:

$$\forall\ \mathsf{xs} : \alpha\ \mathsf{list},\ \mathsf{x} : \alpha.\ \mathsf{len}\ (\mathsf{x} :: \mathsf{xs}) = \mathsf{len}\ (\mathsf{xs}) + 1 \wedge \mathsf{len}\ [] = 0$$

Morpheus is implemented in approximately 9K lines of OCaml code. The input to the verifier is a Morpheus program definition, correctness specifications, and any required qualifier definitions. Given this, Morpheus infers types for other expressions and component parsers, generates first-order verification conditions using the typing semantics discussed earlier, and checks the validity of these conditions.

5.2 Results and Discussions

We have implemented and verified the examples given in the paper, along with a set of benchmarks capturing interesting, real-world safety properties relevant to data-dependent parsing tasks. The goal of our evaluation is to consider the effectiveness of Morpheus with respect to generality, expressiveness and practicality. Table **??** shows a summary of the benchmark programs considered. Each benchmark is a Morpheus parser program affixed with a meaningful safety property (last column). The first column gives the name of the benchmark. The second column of the table describes benchmark size in terms of the number of lines of Morpheus code, without the specifications. The third column gives a pair D/P, showing the number of unique derived (D) combinators (like count, many, etc.) used in the benchmark from the Morpheus library, and the number of primitive (P) parsers (like string, number, etc.) from the Morpheus library used in the benchmark; the former provides some insight on the usability of our design choices in realizing extensibility. The fourth column

[8] https://github.com/aegis-iisc/morpheus.git

lists the size of the grammar along with the number of production rules in the grammar. The fifth column gives the number of verification conditions generated, followed by the time taken to verify them (sixth column). The overall verification time is the time taken for generating verification conditions plus the time Z3 takes to solve these VCs. All examples were executed on a 2.7GHz, 64 bit Ubuntu The next column quantifies the annotation effort for verification. It gives a ratio (#A/#Q) of required user-provided specifications (in terms of the number of conjuncts in the specification) to the total specification size (annotated + inferred). User-provided specifications are required to specify a top-level safety property and to specify invariants for fix expressions akin to loop invariants that would be provided in a typical verification task. Finally, the last column gives a high-level description of the data-dependent safety property being verified.

Our benchmarks explore data-dependent parsers from several interesting categories.[9] The first category, represented by Idris do-block, Haskell case-exp and Python while-statement, capture parsing activities concerned with layout and indentation introduced earlier. Languages in which layout is used in the definition of their syntax require context-sensitive parser implementations [**?**, **?**]. We encode a Morpheus parser for a sub-grammar for these languages whose specifications capture the layout-sensitivity property.

The second category, represented by png and ppm consider data-dependent image formats like PNG or PPM. Verifying data-dependence is non-trivial as it requires verifying an invariant over a monadic composition of the output of one parser component with that of a downstream parser component, interleaved with internal parsing logic.

The next category, captured by xauction, xprotein, and health, represent data-dependent parsing in data-processing pipelines over XML and CSV databases. For xauction and xprotein, we extend XPath expressions over XML to *dependent* XPath expressions. Given that XPath expressions are analogous to regular-expressions over structured XML data, *dependent* XPath expressions are analogous to dependent regular-expressions over XML. We use these expressions to encode a property of the XPath query over XML data for an online auction and protein database, resp. Note that verifying such properties over XPath queries is traditionally performed manually or through testing. In the case of health, we extend regular custom pattern-matching over CSV files to stateful custom pattern-matching, writing a data-dependent custom pattern matcher. We verify that the parser correctly checks relational properties between different columns in the database.

The next two categories have one example each: we introduced the c typedef parser in Section **??** that uses data dependence and effectful data structures to disambiguate syntactic categories (e.g., *typenames* and *identifiers*) in a language definition. Benchmark streams defines a parser over streams (i.e. input list indexed with natural numbers).

5.2.1 Annotation overhead vs inference

There are some interesting things to note in the second to last column(#A/#Q); First, as the benchmarks (grammars) become more complex, i.e., have a greater number of functions (sub-parsers), the ratio decreases (small is better). In other words, the gains of type-inference become more visible (e.g., haskell, idris, c typedef). The worst (highest) ratio is for the PPM parser. This parser is interesting because, even though the grammar is small, it makes multiple calls to fixpoint combinators. Thus, the user must provide specifications for the top-level parser and each fix-point combinator, thus increasing (#A). Additionally, given a

[9] The grammar for each of our implementations is given in the technical report [**?**].

Name	# Loc	D/P	G(#prod)	# VCs	T (s)	(#A/#Q)	data-dependence
haskell	110	5/4	20 (7)	17	8.11	9/39	layout-sensitivity
idris	115	5/5	22(8)	33	10.46	7/26	layout-sensitivity
python	47	3/3	25 (7)	23	7.44	6/20	layout-sensitivity
ppm	46	5/2	21 (7)	20	5.33	4/9	tag-length-data
png chunk	30	3/4	10 (2)	12	3.38	2/7	tag-length-data
xauction	54	4/4	31 (10)	19	6.70	2/8	data-dependent XPath expression
xprotein	45	3/3	24(6)	22	6.23	4/10	data-dependent XPath expression
health	40	4/3	15(5)	13	4.56	2/8	data-dependent CSV pattern-matching
c typedef	60	4/4	14 (5)	21	6.78	4/16	context-sensitive disambiguation
streams	51	4/2	12 (4)	16	5.21	2/9	safe stream manipulation

Table 1 Summary of Benchmarks : #Loc Loc defines the size of the parser implementation in Morpheus; D/P gives the number of derived/primitive combinator uses in the parser implementation; grammar size G(# prod) defines size of the grammar along with the number of production rules in the grammar; #VCs defines number of VCs generated; T(s) is the time for discharging these VCs in seconds; (#A/#Q) defines the ratio of number of conjuncts used in the specification provided by the user (#A) to the total number of conjuncts (#Q) across all files in the implementation; Property gives a high-level description of the data-dependent safety property.

small number of functions (sub-parsers) due to small grammar size, the gains due to inference are also low. In summary, these trends show that the efforts needed for verification are at par with other Refinement typed languages like, Liquid Types [**?**], FStar [**?**], etc, and as the parsers become bigger, the benefits of inference become more prominent.

5.3 Case Study: Indentation Sensitive Parsers

As a case study to illustrate Morpheus's capabilities, we consider a particular class of stateful parsers that are *indentation-sensitive*. These parsers are characterized by having indentation or layout as an essential part of their grammar. Because indentation sensitivity cannot be specified using a context-free grammar, their specification is often specified via an orthogonal set of rules, for example, the offside rule in Haskell.[10] Haskell language specifications define these rules in a complex routine found in the lexing phase of the compiler [**?**]. Other indentation-sensitive languages like Idris [**?**] use parsers written using a parser combinator libraries like Parsec or its variants [**?**, **?**] to enforce indentation constraints.

Consider the Idris grammar fragment shown in Figure **??**. The grammar defines the rule to parse a do-block. Such a block begins with the do keyword, and is followed by zero or more do statements that can be let expressions, a binding operation (←) over names and expressions, an external expression, etc. The Idris documentation specifies the indentation rule in English governing where these statements must appear, saying that the "*indentation of each* do *statement in a* do*-block* Do* *must be greater than the current indentation from which the rule is invoked* [**?**]." Thus, in the Idris code fragment shown in Figure **??**, indentation sensitivity constraints require that the last statement is not a part of the do-block, while the inner four statements are. A correct Idris parser must ensure that such indentation rules are

[10] https://www.haskell.org/onlinereport/haskell2010/haskellch10.html

```
DoBlock ::= 'do' OpenBlock Do*
      CloseBlock;
Do ::=
    'let' Name TypeSig' '=' Expr
  | 'let' Expr' '=' Expr
  | Name '← ' Expr
  | Expr' '← ' Expr
  | Ext Expr
  | Expr
```

(a) An Idris grammar rule for a do block

```
expr = do
          t ← term
          symbol "+"
          e ← expr
          pure t + e
       symbol '*'
```

(b) An input to the parser.

Figure 8 An Idris grammar rule for a do block and an example input.

preserved.

Figure **??** presents a fragment of the parser implementation in Haskell for the above grammar, taken from the Idris language implementation source, and simplified for ease of explanation. The implementation uses Haskell's Parsec library, it implements indentation rules using a state abstraction (called IState) that stores the current indentation level as parsing proceeds. The parser then manually performs reads and updates to this state and performs indentation checks at appropriate points in the code (e.g. line **??**, **??**). The IdrisParser (line **??**) is defined in terms of Parsec's parser monad over an Idris state (here, IState), which along with other fields has an integer field (ist) storing the current indentation value. A typical indentation check (e.g. see lines **??** - **??**) fetches the current value of ist using getIst, fetches the indentation of the next lexeme using the Parsec library function indent, and compares these values.

The structure of the implementation follows the grammar (Figure **??**): the doBlock parser parses a reserved keyword "do" followed by a block of do_ statement lists. The indentation is enforced using the parser indentedDoBlock (defined at line **??**) that gets the current indentation value (allowed) and the indentation for the next lexeme using indent, checks that the indentation is greater than the current indentation (line **??**) and updates the current indentation so that each do statement is indented with respect to this new value. It then calls a parser combinator many (line **??**), which is the Parsec combinator for the Kleene-star operation, over the result of indentedDo, i.e., indentedDo^*. The indentedDo parser again performs a manual indentation check, comparing the indentation value for the next lexeme against the block-start indentation (set earlier by indentedDoBlock at line **??**) and, if successful, runs the actual do_ parser (line **??**). Finally, indentedDoBlock resets the indentation value to the value before the block (line **??**).

Unfortunately, it is non-trivial to reason that these manual checks suffice to enforce the indentation sensitivity property we desire. Since they are sprinkled throughout the implementation, it is easy to imagine missing or misplacing a check, causing the parser to misbehave. More significantly, the implementation make incorrect assumptions about the effectful actions performed by the library that are reflected in API signatures. In fact, the logic in the above code has a subtle bug [**?**] that manifests in the input example shown in Figure **??**.

Note that the indentation of the token 'mplus' is such that it is not a part of either do block; the implementation, however, parses the last statement as a part of the inner do-block, thereby violating the indentation rule, leading to the program being incorrectly parsed.

```
data IState = IState {
    ist :: Int
    ...
} deriving (Show)
data PTerm = PDoBlock [PDo]
data PDo t = DoExp t | DoExt t
           | DoLet t t | ...
type IdrisParser a = Parser IState a

getIst :: IdrisParser IState
getIst = get
putIst :: (i : Int) → IdrisParser ()
pustIst i = put {ist = i}

doBlock :: IdrisParser PTerm
doBlock = do
            reserved "do"
            ds ← indentedDoBlock
            return (PDoBlock ds)
indentedDo :: IdrisParser (PDo PTerm)
indentedDo = do
               allowed ← ist getIst
               i ← indent
               if (i <= allowed)
                  then fail ("end of block")
               else do_
indent :: IdrisParser Int
indent =
    do
     if (lookAheadMatches (operator)) then
        do
          operator
          return (sourceColumn.getSourcePos)
      else
          return (sourceColumn.getSourcePos)
do_ :: IdrisParser (PDo PTerm)
do_ = do
          reserved "let"
          i ← name
          reservedOp "="
          e ← expr
          return (DoLet i e)
    <|> do
          e ← expr
          return (DoExt i e)
    <|> do e ← expr
           return (DoExp e)
indentedDoBlock :: IdrisParser [PDo PTerm]
indentedDoBlock =
    do
       allowed ← ist getIst
       lvl' ← indent
       if (lvl' > allowed) then
          do
             putIst lvl'
             res ← many (indentedDo)
             putIst allowed
             return res
       else fail "Indentation error"

lookAheadMatches :: IdrisParser a →
     IdrisParser Bool
lookAheadMatches p =
    do
       match ← lookAhead (optional p)
       return (isJust match)
```

Figure 9 A fragment of a Parsec implementation for Idris do-blocks with indentation checks.

```
1  expr = do
2           t ← term
3           do
4             symbol "+"
5             e ← expr
6             pure t + e
7         `mplus` pure t
```

Figure 10 An input expression that is incorrectly parsed by the implementation shown in Figure **??**.

The problem lies in a mismatch between the contract provided by the library's indent function and the assumptions made about its behavior at the check at line **??** in the indentedDo parser (or similarly at line **??**). Since checking indentation levels for each character is costly, indent is implemented (line **??**) in a way that causes certain lexemes (user defined operators like 'mplus') to be ignored during the process of computing the next indentation level. It uses a lookAheadMatches parser to skip all lexemes that are defined as operators. In this example, indent does not check the indentation of lexeme 'mplus', returning the indentation of the token pure instead. Thus, the indentation of the last statement is considered to start at pure, which incorrectly satisfies the checks at line **??** or line **??**, and thus causes this statement to be accepted as part of indentedDoBlock. Unearthing and preventing such bugs is challenging. We show how implementing the same parser in Morpheus allows us to catch the bug and verify a correct version of the parser. Figure **??** shows a Morpheus implementation for a portion of the Idris doBlock parser from Figure **??** showing the implementation of three parsers for brevity, doBlock, indentedDo, and indent, along with other helper functions. The structure is similar to the original Haskell implementation.

To specify an *indentation-sensitivity* safety property, we first define an inductive type for a parse-tree (tree) and refine this type using a dependent function type, (offsideTree i), that specifies an indentation value for each parsed result.

```
type tree = Tree {term : pterm; indentT : int; children : tree list}
type offsideTree i = Tree {term : pterm;  indentT : { v : int | v > i }; children : (offsideTree i) list}
```

This type defines a tree with three fields:

- A term of type pterm.
- The indentation (indentT) of a returned parse tree, the refinement constraints on indentT requires its value to be greater than i.
- A list of sub-parse trees (children) for each of the terminals and non-terminals in the current grammar rule's right-hand side, each of which must also satisfy this refinement.

Morpheus additionally automatically generates *qualifiers* like, indentT, children, etc, for each of the datatype's constructors and fields with the same name that can be used in type refinements. However, this type is not sufficiently expressive to specify the required safety property for doBlock that requires "*the indentation of the parse tree returned by* doBlock *must be greater than the current value of* ist" because ist is an effectful heap variable.

We can specify a safety property for a doBlock parser as shown on line **??** in Figure **??**. Again, the type specification in blue are provided by the programmer. The type should be understood as follows: The effect label (stexc) defines that the possible effects produced by the parser include state and exc. The precondition binds the value of the mutable state variable ist, a reference to the current indentation level, to I via the use of the built-in qualifier sel that defines a select operation on the heap [**?**]. The return type (offsideTree I result) obligates the computation to return a parse tree (or a failure) whose indentation must be greater than I. The postcondition constraints that the final value of the indentation is to be reset to its value prior to the parse (a *reset* property) when the parser succeeds (case $\nu =$ Inl (_)) or that the input stream inp is monotonically consumed when the parser fails (case $\nu =$ Inr (Err)). The types for other parsers in the figure can be specified as shown at lines

Figure 11 Morpheus implementation and specifications for a portion of an Idris Do-block with indentation checks, do_m is a syntactic sugar for Morpheus's monadic bind. Specifications given in Blue are provided by the parser writer; Gray specifications are inferred by Morpheus.

??, ??, ??, etc.; these types shown in gray are automatically inferred by Morpheus's type inference algorithm.

5.3.1 Revisiting the Bug in the Example

The **bug** described in the previous paragraph is unearthed while typechecking the indentedDo implementation or the indentedDoBlock implementation. We discuss the case for indentedDo case here. To verify that doBlock satisfies its specification, Morpheus needs to prove that the type inferred for the body of indentedDo (lines ??- ??):

1. has a return type that is of the form, offsideTree I. Concretely, the indentation of the returned tree must be greater than the initial value of ist (i.e. indentT (ν) > I).
2. asserts that the final value of ist is equal to the initial value.

Goal (1) is required because indentedDo is used by indentedDoBlock (see Figure ??), which is then invoked by doBlock, where its result constructs the value for children, whose type is offsideTree I list. Goal (2) is required because doBlock's specified post-condition demands it. Type-checking the body for indentedDo yields the type shown at line ??. The two conjuncts in the post-condition correspond to the *then* (failure case) and *else* (success case) branch in the parser's body.

The failure conjunct asserts that the input stream is consumed monotonically if the indentation level is greater than ist. The success conjunct is the post-condition of the do_ parser. This inferred type is, however, too weak to prove goal (1) given above, which requires the combinator to return a parse tree that respects the offside rule. The problem is that indent's type (line ??), inferred as:

```
indent : PE^state{true} ν : int {∀ h, ν, h'. sel (h', inp) ⊆ sel(h, inp)}
```

does not allow us to conclude that indentedDo satisfies the indentation condition demanded by doBlock, i.e., that it returns a well-typed (offsideTree I). This is because the type imposes no constraint between the integer indent returns and the function's input heap, and thus offers no guarantees that its result gives the position of the first lexeme of the input list.

We can revise indent's implementation such that it does not skip any reserved operators and always returns the position of the first element of the input list, allowing us to track the indentation of every lexeme:

```
indent : PE^state {true} ν : int{∀ h, ν, h'.ν = pos (sel (h, inp)) ∧ sel (h', inp) ⊆ sel (h, inp)}
let indent = do_m s ← !inp
             return (sourceColumn s)
```

This type defines a stronger constraint, sufficient to type-check the revised implementation and raise a type error for the original. For this example, Morpheus generated 33 Verification Conditions (VCs) for the revised successful case and 6 VCs for the failing case. We were able to discharge these VCs to the SMT Solver Z3 [?], yielding a total overall verification time of 10.46 seconds in the successful case, and 2.06 seconds in the case when type-checking failed.

This example highlights several key properties of Morpheus verification: The specification language and the type system allows verifying interesting properties over inductive data types (e.g., the offsideTree property over the parse trees). It also allows verifying properties dependent on state and other effects such as the *input consumption* property over input streams (inp). Secondly, the annotation burden on the programmer is proportional to the complexity of the top-level safety property that needs to be checked. Finally, the similarities between the Haskell implementation and the Morpheus implementation minimize the idiomatic burden placed on Morpheus users.

6 Related Work

Parser Verification. Traditional approaches to parser verification involve mechanization in theorem provers like Coq or Agda [?, ?, ?, ?, ?, ?, ?]. These approaches trade-off both automation and expressiveness of the grammar they verify to prove full correctness. Consequently, these approaches cannot verify safety properties of data-dependent parsers, the subject of study in this paper. For instance, RockSalt [?] focuses on regular grammars, while [?, ?] present interpreters for parsing expression grammars (without nondeterminism) and limited semantic actions without data dependence. Jourdan et al. [?] gives a certifying compiler for LR(1) grammars, which translates the grammar into a pushdown automaton and a certificate of language equivalence between the grammar and the automaton. More recently CoStar [?] presents a fully verified parser for the ALL(*) fragment mitigating some of the limitations of the above approaches. However, unlike Morpheus, CoStar does not handle data-dependent grammars or user-defined semantic actions.

Deductive synthesis techniques for parsers like Narcissus [?] and [?] focus mainly on tag-length-payload, binary data formats. Narcissus [?] provides a Coq framework (an encode_decode tactic) that can automatically generate correct-by-construction encoders and decoders from a given user format input, albeit for a restricted class of parsers. Notably, the system is not easily extensible to complex user-defined data-dependent formats such as the examples we discuss in Morpheus. This can be attributed to the fact that the underlying encode_decode Coq tactic is complex and brittle and may require manual proofs to verify a new format. In contrast, Morpheus enables useful verification capabilities for a larger class of parsers, albeit at the expense of automatic code generation and full correctness. Writing a safe parser implementation for a user-defined format in Morpheus is no more difficult than manually building the parser in any combinator framework with the user only having to provide an additional safety specification. EverParse [?] likewise focuses mainly on binary data formats, guaranteeing full-parser correctness, albeit with some expressivity limitations. For example, it does not support user-defined semantic actions or global data-dependences for general data formats. Compared to these efforts, the properties Morpheus can validate are more high-level and general. E.g., "non-overlapping of two lists of strings" in a C-decl parser; "layout-sensitivity properties", etc,. Verifying these properties requires reasoning over a challenging combination of rich algebraic data types, mutable states, and higher-order functions.

[?] also explore types for parsing, defining a core type-system for context-free expressions. However, their goals are orthogonal to Morpheus and are targeted towards identifying expressions that can be parsed unambiguously.

Data-dependent and Stateful Parsers Morpheus allows writing parsers for data-dependent and stateful parsers. There is a long line of work aimed at writing such parsers [?, ?, ?, ?]. None of these efforts, however, provide a mechanism to reason about the parsers they can express. Further, many of these systems are specialized for a particular class/domain of problems, such as [?] for data-dependent grammars with trivial semantic actions, or [?] for indentation sensitive grammars, etc. Morpheus is sufficiently expressive to both write parsers and grammars discussed in many of these approaches, as well as verifying interesting safety properties. Indeed, several of our benchmarks are selected from these works. In contrast, systems such as [?] argue about the correctness of the input parsed against the underlying CFG, a property challenging to define and verify as a Morpheus safety property, beyond simple string-patterns and regular expressions. We leave the expression of such grammar-related properties in Morpheus as a subject for future work.

Refinement Types. Our specification language and type system builds over a refinement type system developed for functional languages like Liquid Types [?] or Liquid Haskell [?]. Extending Liquid Types with *bounds* [?] provides some of the capabilities required to realize data-dependent parsing actions, but it is non-trivial to generalize such an abstraction to complex parser combinators found in Morpheus with multiple effects and local reasoning over states and effects.

Effectful Verification Our work is also closely related to dependent-type-based verification approaches for effectful programs based on monads indexed with either pre- and post-conditions [?, ?] or more recently, predicate monads capturing the weakest pre-condition semantics for effectful computations [?]. As we have illustrated earlier, the use of expressive and general dependent types, while enabling the ability to write rich specifications (certainly richer than what can be expressed in Morpheus), complicates the ability to realize a fully automated verification pathway.

Verification using natural proofs [?] is based on a mechanism in which a fixed set of proof tactics are used to reason about a set of safety properties; automation is achieved via a search procedure over in this set. This idea is orthogonal to our approach where we rather utilize the restricted domain of parsers to remain in a decidable realm. Both our effort and these are obviously incomplete. Another line of work verifying effectful specifications use characteristic formulae [?]; although more expressive than Morpheus types, these techniques do not lend themselves to automation.

Local Reasoning over Heaps Our approach to controlling aliasing is distinguished from substructural typing techniques such as the ownership type system found in Rust [?]. Such type systems provide a much richer and more expressive framework to reason about memory and effects, and can provide useful guarantees like memory safety and data-race freedom etc. Since our DSL is targeted at parser combinator programs which generally operate over a much simplified memory abstraction, we found it unnecessary to incorporate the additional complexity such systems introduce. The integration of these richer systems within a refinement type framework system of the kind provided in Morpheus is a subject we leave for future work.

Parser Combinators There is a long line of work implementing Parser Combinator Libraries and DSLs in different languages [?]. These also include those which provide a principled way for writing stateful parsers using these libraries [?, ?]. As we have discussed, none of these libraries provide an automated verification machinery to reason about safety properties of the parsers. However, since they allow the full expressive power of the host language, they may, in some instances, be more expressive than Morpheus. For example, Morpheus does not allow arbitrary user-defined higher-order functions and builds only on the core API discussed earlier. This may require a more intricate definition for some parsers compared to traditional libraries. For example, traditional parser combinator libraries typically define a higher-order combinator like many_fold_apply with the following signature and use this combinator to concisely define a *Kleene-star* parser:

```
many_fold_apply : f : ('b → 'a → 'b) → (a : 'a) → (g : 'a → 'a) → p : ('a, 's) t → ('b, 's) t
let many p = many_fold_apply (fun xs x → x :: xs) [] List.rev p
```

Contrary to this, in Morpheus, we need to define Kleene-star using a more complex, lower-level fixpoint combinator.

7 Conclusions

This paper presents Morpheus, a deeply-embedded DSL in OCaml that offers a restricted language of composable effectful computations tailored for parsing and semantic actions and a rich specification language used to define safety properties over the constituent parsers comprising a program. Morpheus is equipped with a rich refinement type-based automated verification pathway. We demonstrate Morpheus's utility by using it to implement a number of challenging parsing applications, validating its ability to verify non-trivial correctness properties in these benchmarks.