RFC 001: Extend Quint with Row-Polymorphic Sum Types

Shon Feder

July 21, 2023

Contents

1 Overview

This section gives a concise overview of the principle proposal. In depth discussion follows in the subsequent sections.

1.1 Concrete syntax

Here I illustrate the concrete syntax additions being proposed. See the discussion of concrete syntax for consideration of motivations and trade offs.

1.1.1 Declaration

The proposed syntax follows fits with our existing syntax for records and takes inspiration from Rust.

```
type T = {
  A(int),
  B(str),
  C,
}
```

As indicated by `C`, labels with no associated expression type are allowed. These are sugar for a label typed with the unit (i.e., the empty record). The above will be normalized as

```
type T = {
  A(int),
  B(str),
  C({}),
}
```

1.1.2 Constructors

Following our existing practice of maintaining exotic operators as the normal form of data constructors, we may add an operator `variant` similar to the "injection" operator from Leijen05:

```
<l = _> ::  r .  → <l ::  | r > -- injection
```

This operation injects an expression of type into a sum-type. However, if we don't want to expose the row-polymorphism to users, we'll need a more restrictive typing. This is discussed with the typing rules.

When a sum type declaration is parsed, constructor operations for each variant will be generated internally. E.g., for the type `T` defined above, we will add to the context the following operators:

```
pure def A(x:int): T = variant("A", x)
pure def B(x:str): T = variant("B", x)
pure def C(): T      = variant("C", {})
```

1.1.3 Case analyses

Expressions of a sum type can be eliminated using a case analysis via `match` statements:

```
match e {
  A(a) => ...,
  B(b) => ...,
  C    => ...,
}
```

This construct can either be a primitive or syntax sugar for a more primitive `decompose` operator, discussed below.

We could also consider a case analysis `{ A(a) => e1, B(b) => e2, C => e3 }` – where `e1, e2, e3 : S` – as syntax sugar for an anonymous operator of type `T => S`. `match` would then be syntax sugar for operator application. This follows Scala's case blocks or OCaml's `function` keyword and would allow idioms such as:

```
setOfTs.map({ A(a) => e1, B(b) => e2, C => e3 })
```

instead of requiring

```
setOfTs.map(x => match x { A(a) => e1, B(b) => e2, C => e3 })
```

1.2 Statics

These type rules assume we keep our current approach of using quint strings for labels. But see my argument for simplifying our approach under Drop the exotic operators. See the discussion in Statics below for a detailed explanation and analysis.

1.2.1 Construction

The typing rule for constructing a variant of a sum type:

$$\frac{\Gamma \vdash e\colon (t,c) \quad \Gamma \vdash `l'\colon str \quad definedType(s) \quad free(v)}{\Gamma \vdash \ variant(`l', e) \ : (s, c \wedge s \sim \langle \ l\colon t|v \ \rangle)}$$

This rule is substantially different from Leijen05's

```
<l = i> ::   r .   → <l ::   | r > -- injection
```

because we have decided not to expose the underlying row-polymorphism for sum types at this point.

1.2.2 Elimination

The typing rule for eliminating a variant of a sum type via case analysis:

$$\frac{\Gamma \vdash e : (s,c) \quad \Gamma, x_1 \vdash e_1 : (t, c_1) \quad \ldots \quad \Gamma, x_n \vdash e_n : (t, c_n) \quad \Gamma, \langle v \rangle \vdash e_{n+1} : (t, c_{n+1}) \quad fresh(v)}{\Gamma \vdash \ match \ e \ \{i_1 : x_1 \Rightarrow e_1, \ldots, i_n : x_n \Rightarrow e_n\} : (t, c \wedge c_1 \wedge \ldots \wedge c_n \wedge c_{n+1} \wedge s \sim \langle i_1 : t_1, \ldots, i_n : t_n | v \rangle)}$$

This gives the rule in our system that is sufficient to capture Leijen05's

```
(l   _ ? _ : _) ::   r . <l ::   | r> → (  →  ) → (<r> →  ) →   -- decomposition
```

since we can define decomposition for any label `L` via

```
def decomposeL(e: (L(a) | r), f: a => b, default : r => b) =
  match e {
    | L(x) => f(x)
    | r    => default(r)
  }
```

However we can define `match` as syntax sugar for the decompose primitive if we prefer.

1.3 Dynamics

The dynamics in the simulator should be straightforward and is not discussed here. Translation to Apalache for symbolic execution in the model checker is also expected to be relatively straight forward, since Apalache has a very similar form of row-based sum typing.

The general rules for eager evaluation can be found in PFPL, section 11.2. Additional design work for this will be prepared if needed.

This concludes the tl;dr overview of the proposal. The remaining is an indepth (still v. rough in places, discussion).

2 Discussion

2.1 Motivation

Quint's type system currently supports product types. Product types (i.e., records, with tuples as a special case where fields are indexed by an ordinal) let us specify *conjunctions* of data types in a way that is verifiable statically. This lets us describe more complex data structures in terms of values of specific types that **must** be packaged together. E.g., we might define a rectangle by its length and width and a triangle by the lengths of its three sides. Using Quint's existing syntax for product types, we'd specify this as follows:

```
type Rectangle =
  { l : int
  , w : int }
type Triangle =
  { a : int
  , b : int
  , c : int }
```

Quint's type system does not yet have the the dual construct, sum types (aka "variants", "co-products", or "tagged unions"). Sum types specify *disjunctions* of data types in a way that is verifiable statically. This lets us describe mutually exclusive alternatives between distinct data structures that **may** occur together and be treated uniformly in some context. E.g., we might wish to specify a datatype for shapes, so we can work with collections that include both rectangles and triangles. Using one of the proposed syntax option that will be motivated in the following, this could be specified as

```
type Shape =
  | Rect(rectangle)
  | Tri(triangle)
```

Having both product types and sum types (co-product types) gives us a simple and powerful idiom for specifying families of data structures:

- We describe *what must be given together* to form a product of the specified type, and so *what we may always make use of* by projection when we are given such a product.
- We describe *which alternatives may be supplied* to form a co-product of a specified type, and so *what we must be prepared to handle* during case analysis when we are given such a co-product.

E.g., a `rectangle` is defined by *both* a length *and* a width, packaged together, while a `shape` is defined *either* by a rectangle *or* a triangle. With these definitions established, we can then go on to form and reason about collections of shapes like `Set[shape]`, or define properties common to all shapes like `isEquilateral : shape => bool`[1].

2.2 Context

2.2.1 Existing plans and previous work

We have always planned to support co-products in quint: their utility is well known and widely appreciated by engineers with experience in modern programming languages. We introduced co-products to Apalache in `https://github.com/informalsystems/apalache/milestone/60` for the same reasons. The design and implementation of the latter was worked out by ?? (????) based on the paper "Extensible Records with Scoped Labels". At the core of this design is a simple use of row-polymorphism that enables both extensible variants and extensible records, giving us products and co-products in a one neat package. The quint type system was also developed using row-polymorphism following this design. As a result of this forethought, extension of quint's type system and addition of syntax to support sum-types is expected to be relatively straightforward.

2.2.2 The gist of extensible row-typed records and sum types

The core concept in the row-based approach we've opted for is the following: we can use the same construct, called a "row", to represent the *conjoined*

[1]The expressive power of these simple constructs comes from the nice algebraic properties revealed when values of a type are treated as equal up-to ismorphism. See, e.g., `https://codewords.recurse.com/issues/three/algebra-and-calculus-of-algebraic-data-types`

labeled fields of a product type and the *alternative* labeled choices of a sum type. That the row types are polymorphic lets us extend the products and sums using row variables.

E.g., given the row

$$i_1 : t_1 \ , \ldots \ , i_n : i_n | v$$

with each t_k-typed field indexed by label i_k for $1 \leq k \leq n$ and the free row variable v, then

$$\{i_1 : t_1 \ , \ldots \ , i_n : i_n | v\}$$

is an open record conjoining the fields, and

$$\langle i_1 : t_1 \ , \ldots \ , i_n : i_n | v \rangle$$

is an open sum type presenting the fields as (mutually exclusive) alternatives. Both types are extensible by substituting v with another (possibly open row). To represent a closed row, we omit the trailing $|v$.

2.2.3 Quint's current type system

The current type system supported by quint is based on a simplified version of the constraint-based system presented in "Complete and Decidable Type Inference for GADTs" augmented with extensible (currently, just) records based on "Extensible Records with Scoped Labels". A wrinkle in this genealogy is that quint's type system includes neither GADTs nor scoped labels (and even the extensiblity supported for records is limited). Moreover, due to their respective foci, neither of the referenced papers includes a formalization the complete statics for product types or sum types, and while we have implemented support for product types in quint, we don't have our typing rules recorded.

2.3 Statics

This section discusses the typing judgements that will allow us to statically verify correct introduction and elimination of expressions that are variants of a sum type. The following considerations have informed the structure in which the proposed statics are discussed:

- Since sum-types are dual to product types, I consider their complementary typing rules together: first I will present the relevant rule for product types, then propose the complementary rule for sum types. This should help maintain consistency between the two kinds of typing judgements and ensure our implementations of both harmonize.

- Since we don't have our existing product formation or elimination rules described separate from the implementation, transcribing them here can serve to juice our intuition, supplement our design documentation, and perhaps give opportunity for refinement.

- Since our homegrown type system has some idiosyncrasies that can obscure the essence of the constructs under discussion, I precede the exposition of each rule with a text-book example adapted from Practical Foundations for Programming Languages. This is only meant as a clarifying touchstone.

2.3.1 Eliminating products and introducing sums

The elimination of products via projection and the introduction of sums via injection are the simplest of the two pairs of rules.

1. Projection Here is a concrete example of projecting a value out of a record using our current syntax:

```
val r : {a:int} = {a:1}
val ex : int = r.a
// Or, using our exotic field operator, which is currently the normal form
val ex_ : int = r.field("a")
```

   A textbook rule for eliminating an expression with a finite product types can be given as

$$\frac{\Gamma \vdash e \colon \{i_1 \hookrightarrow \tau_1,\ \ldots,\ i_n \hookrightarrow \tau_n\} \quad (1 \leq k \leq n)}{\Gamma \vdash e.i_k \colon \tau_k}$$

   Where i is drawn from a finite set of indexes used to label the components of the product (e.g., fields of a record or positions in a tuple) and $i_j \hookrightarrow \tau_j$ maps the index i_j to the corresponding type τ_j.

   This rule tells us that, when an expression e with a product type is derivable from a context, we can eliminate it by projecting out of e with an index i_k (included in the type), giving an expression of the type t_k corresponding to that index. If we're given a bunch of stuff packaged together we can take out just the one part we want.

In our current system, typechecking the projection of a value out of a record implements the following rule

$$\frac{\Gamma \vdash e\colon (r, c) \quad \Gamma \vdash `l'\colon str \quad fresh(t)}{\Gamma \vdash \ field(e, `l') \ : (t, c \wedge r \sim \{\ l\colon t | tail\_t \ \})}$$

where

- we use the judgement syntax established in ADR5, in which $\Gamma \vdash e : (t, c)$ means that, in the typing context Γ, expression e can be derived with type t under constraints c,
- $fresh(t)$ is a side condition requiring the type variable t to be fresh in Γ,
- $`l'$ is a string literal with the internal representation l,
- c are the constraints derived for the type r,
- $tail\_t$ is a free row-variable constructed by prefixing the fresh variable t with "tail",
- $\{\ l\colon t | tail\_t \ \}$ is the open row-based record type with field, l assigned type t and free row- left as a free variable,
- and $r \sim \{\ l\colon t | tail\_t \ \}$ is a unification constraint.

Comparing the textbook rule with the rule in our system helps make the particular qualities and idiosyncrasies of our system very clear.

The most critical difference w/r/t to the complexity of the typing rules derives form the fact that our system subordinates construction and elimination of records to the language level operator application rather than implementing it via a special constructs that work with product indexes (labels) directly. This is what necessitates the consideration of the string literal $`l'$ in our premise. In our rule for type checking record projections we "lift" quint expressions (string literals for records and ints for products) into product indexes.

The most salient difference is the use of unification constraints. This saves us having to "inspect" the record type to ensure the label is present and obtain its type. These are both accomplished instead via the unification of r with the minimal open record including the fresh type t, which will end up holding the inferred type for the projected value iff the unification goes through. This feature of our type system is of special note for our aim of introducing sum-types: almost all the logic for ensuring the correctness of our typing judgements is delegated to the unification rules for the row-types that carry our fields for product type and sum types alike.

2. Injection Here is a concrete example of injecting a value into a sum type using one variant of the proposed syntax:

   ```
   val n : int = 1
   val ex : A(int) | r = A(1)
   ```

   A textbook rule for eliminating an expression belonging to a finite product type can be given as

   $$\frac{\Gamma \vdash e\colon \tau_k \quad (1 \leq k \leq n)}{\Gamma \vdash i_k \cdot e\colon \langle i_1 \hookrightarrow \tau_1,\ \ldots,\ i_n \hookrightarrow \tau_n \rangle}$$

   Where i is drawn from a finite set of indexes used to label the possible alternatives of the co-product and $i_j \hookrightarrow \tau_j$ maps the index i_j to the corresponding type τ_j. We use $\langle \ldots \rangle$ to indicate the labeling is now disjunctive and $i_k \cdot e$ as the injection of e into the sum type using label i_k. Note the symmetry with complementary rule for projection out of a record: the only difference is that the (now disjunctive) row (resp. (now injected) expression) is swapped from premise to conclusion (resp. from conclusion to premise).

   This rule tells us that, when an expression e with a type t_k is derivable from a context, we can include it as an alternative in our sum type by injecting it with the label i_k, giving an element of our sum type. If we're given a thing that has a type allowed by our alternatives, it can included among our alternatives.

   If we were following the row-based approach outlined in Leijen05, then the proposed rule in our system, formed by seeking the same symmetry w/r/t projection out from a product, would be:

   $$\frac{\Gamma \vdash e\colon (t, c) \quad \Gamma \vdash `l'\colon str \quad fresh(s)}{\Gamma \vdash \ variant(`l', e)\ :(s, c \wedge s \sim \{\ l\colon t | tail\_s\ \})}$$

   Comparing this with our current rule for projecting out of records, we see the same symmetry: the (now disjunctive) row type is synthesized instead of being taken from the context.

However, if we don't want to expose the row-polymorphism to users, we need a more constrained rule that will ensure the free row variable is not surfaced. We can address this by replacing the side condition requiring s to be free with a side condition requiring that there it be defined, and in our constraint check that we can unify that defined type with a row that contains the given label with the expected type and is otherwise open.

$$\frac{\Gamma \vdash e\colon (t, c) \quad \Gamma \vdash `l'\colon str \quad definedType(s) \quad free(v)}{\Gamma \vdash \ variant(`l', e) \ \colon (s, c \land s \sim \langle\ l\colon t|v\ \rangle)}$$

2.3.2 Introducing products and eliminating sums

Forming expressions of product types by backing them into records and eliminating expressions of sum types by case analysis exhibit the same duality, tho they are a bit more complex.

1. Packing expressions into records Here is a concrete example of forming a record using our current syntax:

```
val n : int = 1
val s : str = "one"
val ex : {a : int, b : str} = {a : n, b : s}
// Or, using our exotic Rec operator, which is currently the normal form
val ex_ : {a : int, b : str} = Rec("a", n, "b", s)
```

   A textbook introduction rule for finite products is given as

$$\frac{\Gamma \vdash e_1\colon \tau_1 \quad \ldots \quad \Gamma \vdash e_n\colon \tau_n}{\Gamma \vdash \{i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n\}\colon \{i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n\}}$$

   This tells us that for any expressions $e_1 : \tau_1, \ldots, e_n : \tau_n$ derivable from our context we can form a product that indexes those n expressions by $i_1, \ldots, i_n$ distinct labels, and packages all data together in an expression of type $\{i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n\}$. If we're given all the things of the needed types, we can conjoint them all together into one compound package.

   The following rule describes our current implementation:

$$\frac{\Gamma \vdash (`i_1`, e_1\colon (t_1, c_1)) \quad \ldots \quad \Gamma \vdash (`i_1`, e_n\colon (t_n, c_n)) \quad fresh(s)}{\Gamma \vdash Rec(`i_1`, e_1, \ldots, `i_n`, e_n) \ \colon \ (s, c_1 \land \ldots \land c_n \land s \sim \{i_1\colon t_1, \ldots, i_n\colon t_n\})}$$

The requirement that our labels show up in the premise as quint strings paired with each element of the appropriate type is, again, an artifact of the exotic operator discussed later, as is the `Rec` operator in the conclusion that consumes these strings. Ignoring those details, this rule is quite similar to the textbook rule, except we use unification of the fresh variable `s` to propagate the type of the constructed record, and we have to do some bookkeeping with the constraints from each of the elements that will be packaged into the record.

2. Performing case analysis Here is a concrete example of case analysis to eliminate an expression belonging to a sum type using one of the proposed syntax variants:

```
val e : <a:int, b:str> = <a:1>
def describeInt(n: int): str = if (n < 0) then "negative" else "positive"
val ex : str = match e {
  a : x => describeInt(x),
  b : x => x,
}
```

A textbook rule for eliminating an expression that is a variant of a finite sum type can be given as

$$\frac{\Gamma \vdash e\colon \langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n \rangle \quad \Gamma, x_1 : \tau_1 \vdash e_1 \colon \tau \quad \ldots \quad \Gamma, x_n : \tau_n \vdash e_n \colon \tau}{\Gamma \vdash \ \mathit{match}\ e\ \{i_1 \cdot x_1 \hookrightarrow e_1 | \ldots | i_n \cdot x_n \hookrightarrow e_n\} : \tau}$$

Note the complementary symmetry compared with the textbook rule for product construction: product construction requires `n` expressions to conclude with a single record-type expression combining them all; while sum type elimination requires a single sum-typed expression and `n` ways to convert each of the `n` alternatives of the sum type to conclude with a single expression of a type that does not need to appear in the sum type at all.

The proposed rule for quint's type system is given without an attempt to reproduce our practice of using quint strings. This can be added in later if needed:

$$\frac{\Gamma \vdash e : (s, c) \quad \Gamma, x_1 \vdash e_1 : (t, c_1) \quad \ldots \quad \Gamma, x_n \vdash e_n : (t, c_n) \quad \Gamma, \langle v \rangle \vdash e_{n+1} : (t, c_{n+1}) \quad \mathit{fresh}(v)}{\Gamma \vdash \ \mathit{match}\ e\ \{i_1 : x_1 \Rightarrow e_1, \ldots, i_n : x_n \Rightarrow e_n\} : (t, c \wedge c_1 \wedge \ldots \wedge c_n \wedge c_{n+1} \wedge s \sim \langle i_1 : t_1, \ldots, i_n : t_n}$$

Compared with quint's rule for product construction we see the same complementary symmetry. However, we also see a striking difference: there is no row variable occurring in the product construction, but the row variable plays an essential function in sum type elimination of row-based variants. Row types in records are useful for extension operations (i.e., which we don't support in quint currently) and for operators that work over some known fields but leave the rest of the record contents variable. But the core idea formalized in a product type is that the constructor *must* package all the specified things together so that the recipient *can* chose any thing; thus, when a record is constructed we must supply all the things and there is no room for variability in the row. For sum types, in contrast the constructor *can* supply any one thing (of a valid alternate type), and requires the recipient *must* be prepared to handle every possible alternative.

In the presence of row-polymorphis, however, the responsibility of the recipient is relaxed: the recipient can just handle a subset of the possible alternatives, and if the expression falls under a label they are not prepared to handle, they can pass the remaining responsibility on to another handler.

Here is a concrete example using the proposed syntax, note how we narrow the type of `T`:

```
type T = A | B;;
def f(e) = match e {
  | A => 1
  | B => 2
  | _ => 0
}

// f can be applied to a value of type T
let a : T = A
let ex : int = f(A)

// but since it has a fallback for an open row, it can also handle any other var
let foo = Foo
let ex_ : int = f(foo)
```

Here's the equivalent evaluated in OCaml as proof of concept:

```
utop #
type t = [`A | `B]
let f = function
  | `A -> 1
  | `B -> 2
  | _  -> 0
let ex = f `A, f `Foo
;;
type t = [ `A | `B ]
val f : [> `A | `B ] -> int = <fun>
val ex : int * int = (1, 0)
```

2.4 Dynamics

TODO

2.5 Concrete Syntax

Other languages with polymorphic variants:

- ReScript: https://rescript-lang.org/docs/manual/latest/polymorphic-variant
- OCaml: https://v2.ocaml.org/manual/polyvariant.html

Considerations

- Assuming we support anonymous variant types, we need a way of constructing variants without pre-defined constructors. Potential approaches include:
  - A special syntax that (ideally) mirrors the syntax of the type
  - A special lexical marker on the labels (what ReScrips and OCaml do), e.g.,
  - Reserve uppercase letters for variant injectors

```
`A(1)
```

2.5.1 Sketch of an alternative syntax

Option 2

This group of alternatives follows Leijen05:

```
type T =
  < A : int
  , B : str
  >
```

The case of labels initial letters could vary in either option.

Injection using a syntax that is symmetrical with records and matches thy type syntax

```
val a : T = <A:1>
```

Since option 2 suggests a syntactically unambiguous representation of variant formation, we could avoid generating the injectors and/or this could be the normal form for injection.

```
def f(n: int): <C:int, D:str | s> =
  if (n >= 0) <A:n> else <D:"negative">
```

Compare with the corresponding annotation for a record type:

```
def f(n: int): {C:int, D:str | s} =
  if (n >= 0) {C:n, D:"positive"} else {C:n, B:"negative"}
```

Option 2

```
match e {
  | A : a => ...,
  | B : b => ...
}
```

2.5.2 Declaration

1. Copy rust exactly

    ```
    enum T {
      A(int),
      B(int)
    }
    ```

    - Breaks with our current convention around type declarations and keywords.

- The part enclosed in prackets is syntactically indistinguishable from a block of operators that we'd combine with `any` or `and`.
- May mislead users to try injecting values into the type via Rust's `T::A(x)` syntax, which clashes with our current module syntax.
- Anonymous variant types would look really confusing

  ```
  def foo(x: int): {A(int), B(int) | t} = { if (and { C(x), D(x) }) A(x) els
  ```

- Closer to Rust but further from TypeScript and ReScript

2.5.3 Annotation

- OCaml
- Rust

2.5.4 Case analysis

sugar for case analysis and pattern matching

Ergonomic support for sum types requires eliminators, ideally in the form if case analysis by pattern matching.

Rust's pattern syntax is not terribly far off from our syntax:

```
match x {
    A    => println!("a"),
    B    => println!("b"),
    C(v) => println!("cv"),
    _    => println!("anything"),
}
```

The `match` is a close analogue to our existing `if` expressions, and the reuse of the `=>` hints at the connection between case elimination and anonymous operators. The comma separated alternatives enclosed in `{...}` follow the variadic boolean action operators, which seems fitting, since sum types are disjunction over data.

One question if we adopt some form of pattern-based case analysis is how far we generalize the construct. Do we support pattern matching on scalar like ints and symbols? Do we support pattern matching to deconstruct compound data such as records and lists? What about sets? Do we allow pattern expressions to serve as anonymous operator (like Scala)?

My guess is that in most cases the gains in expressivity of specs would justify the investment, but it is probably best to start with limiting support to defined sum types and seeing where we are after that.

- Examples of use in existing specs
- Translation into sum types over rows (following our reference paper)
- Mapping into TLA

Considerations

Possible confusion around eliminator syntax in absence of full pattern matching. Alternative:

```
match x {
    A : _ => println!("a"),
    B : _ => println!("b"),
    C : v => println!("cv"),
    _ : _ => println!("anything"),
}
```

Alternatively we just flag a parsing error of the deconstructor arg is not a free variable, and inform the user that full pattern matching isn't yet supported.

NOTE: Loss of consistency in declaration vs. construction/elimination.

2.6 Additional consideration

- Pattern matching
- User defined parametric type constructors

2.6.1 Drop the exotic operators

- Remove the special product type operators `fieldNames`, `Rec`, `with`, `label`, and `index`, or add support for first-class labels As is, I think these are not worth the complexity and overhead.

Compare our rule with the projection operation from "Extensible Records with Scoped Labels", which does not receive the label 'l' as a string, instead treating it as a special piece of syntax:

```
(_.l) :: r . (l|r)  {l ::  | r } → `
```

Another point of comparison is Haskell's "Datatypes with Field Labels", which generates a projection function for each label, so that defining the datatype

```
data S = S1 { a :: Int, b :: String }
```

will produce functions

```
a :: S -> Int
b :: S -> String
```

Abandoning this subordination to normal operator application would leave us with a rule like the following for record projection:

$$\frac{\Gamma \vdash e\colon (r, c) \quad fresh(t)}{\Gamma \vdash \ e.l\ : (t, c \wedge r \sim \{\ l\colon t | tail\_t\ \})}$$

This would allow removing the checks for string literals, instead leaving that to the outer-level, syntactic level, of our static analysis. A similar simplification would be follow for record construction: the rule for `Rec` would not need to validate that it had received an even number of argument of alternating string literals and values, since this would be statically guaranteed by the parsing rules for the $\{l_1 : v_1, \ldots, l_n : v_n\}$ syntax. This would be a case of opting for the "Parse, don't validate" strategy.