Programmable Record Types in Haskell

Arthur Jamet
University of Kent
Canterbury, United Kingdom
aj530@kent.ac.uk

Michael Vollmer
University of Kent
Canterbury, United Kingdom
m.vollmer@kent.ac.uk

Abstract

In Haskell, accessing an object's fields requires *deconstructing* it. Thankfully, it is possible to name the fields of a data type using the *record syntax*, allowing programmers to access objects' fields using their name. This can help improve the readability of Haskell code. However, Haskell's support for record types is limited, as its type system is nominal, and the language does not allow composing record types.

Previous work tackled this issue using type-level computations at compile time and linked lists at runtime. Since these approaches do not use native record data types, operations on such records (e.g. traversals) are consequently slower than on native data types.

In this paper, we leverage meta-programming and code generation to enable easy record composition and simulate structural subtyping, using *type-transforming* functions and typeclasses generated at compile time. The resulting Haskell library, named `type-machine`, generates native record data types. Its API allows users to write their own functions to compose record types. Our approach does not require any compiler or runtime modifications.

Our benchmarks show that records generated by `type-machine` are at least 40% faster to traverse than records defined using state-of-the-art libraries. Additionally, simple programs that define data types using `type-machine` are at least 3x faster to compile than isomorphic programs that use these libraries.

We present use cases of the library along with examples of custom type transformers.

CCS Concepts: • **Information systems** → *Data structures.*

Keywords: Data Types, Records, Code Generation

ACM Reference Format:
Arthur Jamet and Michael Vollmer. 2026. Programmable Record Types in Haskell. In *Proceedings of the 25th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '26), June 29, 2026, Brussels, Belgium.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3814885.3816408

GPCE '26, Brussels, Belgium

ACM ISBN 979-8-4007-2718-4/2026/06
https://doi.org/10.1145/3814885.3816408

```haskell
data Maybe a = Nothing | Just a

getJustOrZero :: Maybe Int -> Int
getJustOrZero (Just n) = n
getJustOrZero Nothing = 0
```

Figure 1. Accessing an ADT field through deconstruction

1 Introduction

In Haskell, one can define data structures using algebraic data types (ADT). ADTs can have multiple *constructors*, and each one of them can have zero or more *fields*. For example, the `Maybe` data type (Figure 1) has two constructors, `Nothing`, which has zero fields, and `Just`, which has one field.

Typically, the fields of a constructor can be accessed by *deconstructing* or *pattern matching* on the constructor. For example, in Figure 1, at line 4, we deconstruct the argument to access the field of the `Just` constructor.

Deconstruction and pattern matching are the most idiomatic ways to access an ADT's fields. This is practical for ADTs whose constructors have a small number of fields, but it can quickly become unwieldy as the number of fields increases. Figure 2a illustrates this problem: accessing a field requires knowing 1) its position in the constructor's definition and 2) the number of fields the constructor has. Additionally, if the definition of the `URL` ADT were to change, the code maintainer would have to update the deconstruction in the `getPath` function.

Moreover, we could argue that the definition of the ADT itself lacks clarity: fields are identifiable by their position and their type, so it is easy to confuse two adjacent fields that have the same type.

To remedy these issues, it is possible to 1) use Haddock-style comments[1] to document the meaning of the fields (as we did in Figure 2a), or 2) use Haskell's *record* syntax[2].

This syntax allows naming fields and accessing them using their names. It can be seen as syntactic sugar, as the definition of `URL` using this syntax in Figure 2b is isomorphic to the one in Figure 2a and both types can be deconstructed the same way. This record syntax enhances the clarity of the ADT's definition and allows avoiding full deconstruction when pattern matching (see the implementation of `getPath` in Figure 2b).

[1] https://hackage.haskell.org/package/haddock
[2] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/traditional_record_syntax.html

```haskell
data URL = URL
    String -- ^ Scheme
    String -- ^ Hostname
    Maybe Int -- ^ Port
    String -- ^ Path
    [(String, String)] -- ^ Query Parameters
    Maybe String -- ^ Fragment (#)

getPath :: URL -> String
getPath (URL _ _ _ path _ _) = path
```

(a) Using the regular syntax

```haskell
data URL = URL {
    scheme :: String,
    hostname :: String,
    port :: Maybe Int,
    path :: String,
    queryParams :: [(String, String)],
    fragment :: Maybe String,
}

getPath :: URL -> String
getPath url = path url
```

(b) Using the record syntax

Figure 2. Syntax for defining ADTs in Haskell

Composing Types. Even though Haskell is a type-oriented language, it does not provide a way to compose data types (e.g. merging two data types together). To simulate data composition, Haskell developers have to *nest* data types.

Let us consider TypeScript, an object-oriented and strongly-typed programming language. Its type-level functions allow programmers to assemble and extract fields from type definitions. We will call these functions *type transformers*. For example, in `type Vec3 = Record<'x' | 'y' | 'z', number>`, we define a data type, `Vec3`, using the `Record` type transformer to generate a record with three fields (`x`, `y` and `z`) with type `number`. In `type Vec2 = Omit<Vec3, 'z'>`, we use the `Omit` type transformer to derive a new data type from `Vec3`, without the `z` field.

TypeScript's type transformers make it easy to compose data types. A practical example would be with web servers, where endpoints would return different forms of the same data depending on the request. For example, if the user making a request is an administrator, the server would return a list of `Users`, otherwise it would return one of `UserWithoutEmails` (where `type UserWithoutEmail = Omit<User, 'email'>`).

Subtyping on Records. In Haskell, *subtyping* is enabled by (or can be simulated with) *typeclasses*. ADTs can be *instances* of a typeclass, and functions can use typeclasses as constraints on the type of their parameter. A good example

```haskell
getName :: (HasField "name" a String)
        => a -> String
getName elem = getField @"name" elem
```

Figure 3. Using the `HasField` typeclass for structural subtyping

```typescript
type User = {
    job: string; name: string;
}
type Pet = {
    name: string; age: number;
}

const user: User =
    { job: "Teacher", name: "John" }
const pet: Pet =
    { name: "Clifford", age: 59 }

function getName(
    arg: { name: string }
): string {
    return arg.name;
}

> getName(user) // returns "John Doe"
> getName(pet) // returns "Clifford"
```

Figure 4. An example of a TypeScript function relying on subtyping for its parameter

is Haskell's `print` function, which has type `(Show a) => a -> IO ()`. Here, the type variable `a` has the `Show` constraint, meaning that the function accepts any value as parameter as long as its type is an instance on the `Show` typeclass.

Even though Haskell's typing is primarily nominal[6], GHC provides a `HasField` typeclass that allows simulating structural subtyping for records. As an example, consider Figure 3 where we define a `getName` function isomorphic to the one defined in Figure 4. However, this typeclass only allows accessing fields, not updating them.

On the other hand, TypeScript's typing is *structural*[1]. Generally speaking, structural subtyping allows a record type S to be used where T is expected if S contains *at least* all of the fields of T with compatible types [2, 3]. Figure 4 illustrates this: the `getName` function accepts any values as long as they have a `name` field.

Previous Work. Prior work has explored the general area of record subtyping and extensible records in the context of typed programming language design. For example, Gaster and Jones proposed a system for extensible records and variants [4] which was implemented in the Haskell interpreter Hugs, and which inspired a proposal to add a simpler system to Haskell [6]. Beyond Haskell, there has been research on language design and semantics for extensible records [2, 3].

There have also been prior attempts at embedding extensible records in Haskell via a *library*. A common approach to solving this problem in "ordinary" Haskell code is to use HList [7], a library for typed, heterogeneous lists. It has been used to implement records as essentially an association list, and such a representation allows for the adding/removing of fields in a type-safe way. In another approach, Jeltsch defines a type-safe record composition system [5], which provides a way to fold over composed records, as well as a way to simulate structural subtyping using records fields. This led to the implementation of the `records` library[3]. Similarly, the `extensible` library[4] relies on a custom type-operator-based DSL to generate record types, and *lenses* to access record fields. The simulation of subtyping on records is made possible thanks to an `Associate` typeclass, which enforces the presence of a field with a given name and type. Yet another similar library is `superrecord`, which stands out by allowing type-safe anonymous records, with value-level field definition and very little boilerplate. As with `extensible`'s `Associate`, `superrecord`'s `Has` typeclass simulates structural subtyping on record fields.

These libraries are alike in several ways. First, they all use custom type-level operators to define the structure of a record type. Second, as these records are represented at runtime using structures similar to heterogeneous lists, the use of records defined by these libraries may come with a computing overhead as they are adding memory overhead compared to native Haskell records.

Contributions. In this paper, we present a new way to derive record types and simulate structural subtyping in Haskell using meta-programming and code generation. We introduce `type-machine`[5], a Haskell library and framework that allows composing and deriving record types. It comes with type transformer functions inspired by TypeScript's own type transformers, and generates native Haskell records. Thus, unlike records defined using libraries like `records`, `extensible` and `superrecord`, using records defined by `type-machine` causes negligible computing overhead. Figure 5 shows how the library can be used to generate and compose record types.

In Section 2, we go through the features of `type-machine`. Then, in Section 3, we dive into its architecture and technical details. We present a couple of examples and use-cases in Section 4. We also showcase the computing overhead caused by libraries like `superrecord` and `extensible` in Section 5. Finally, we reflect on the limitations of the library and present possible future work in Section 6.

[3] https://hackage.haskell.org/package/records
[4] https://hackage.haskell.org/package/extensible
[5] https://github.com/Arthi-chaud/type-machine

```haskell
$(type_ "Vec2"
    (record ["x", "y"] [t|Int|]))
-- Generates
-- data Vec2 = Vec2 {
--     x :: Int,
--     y :: Int,
-- }

$(type_ "Vec3"
    (union <::> ''Vec2
           <:> record ["z"] [t|Int|]))
-- Generates
-- data Vec3 = Vec3 {
--     x :: Int,
--     y :: Int,
--     z :: Int,
-- }

$(defineIs ''Vec2)
$(deriveIs ''Vec2 ''Vec3)

translateX :: (IsVec2 a) =>
              Int -> a -> a
translateX n v = setX (n + getX v) v

example = translateX (-1) (Vec3 1 2 3)
```

Figure 5. Defining and composing records using `type-machine`

2 Overview

In this section, we present the main features of `type-machine`: the `type_` function, the type transformers that come with the library, and the available utilities to simulate subtyping on records.

2.1 `type_`

The main 'entrypoint' of `type-machine` is the `type_` function. It is a Template Haskell function, meaning that it is evaluated at compile time and can output Haskell code. The function takes two parameters: the name of the ADT to generate, and a type transformer. For example, consider Figure 6a, where we invoke `type_` to define a `UserId` record. It has a single field, `id`, which has type `Int`. For comparison, Figure 6b gives a declaration of an isomorphic record in TypeScript.

2.2 Type Transformers

Type transformers are functions that either define a record type or manipulate the structure of an existing one. `type-machine` comes with a set of type transformers, inspired by TypeScript's own type transformers[6]. It includes:

- `pick` and `omit`, which, given a list of field names, derive a record by *picking* or *omitting* fields.

[6] https://www.typescriptlang.org/docs/handbook/utility-types.html

```
$(type_ "UserId" (record ["id"] [t|Int|]))
```

(a) Using `type-machine`

```
type UserId = Record<'id', number>
```

(b) In TypeScript

Figure 6. Defining a `UserId` record: `type-machine` vs. TypeScript

```
$(type_ "UserWithoutId"
    (omit ["id"] <::> ''User))
```

Figure 7. Using the `omit` type transformer

- `record`, to build records based on a list of keys
- `union` and `intersection`, to get the union and intersection of two records respectively
- `require`, to remove the `Maybe`s wrapped around the given set of fields
- `partial`, which marks all the fields as 'optional'. I.e. each field type will be wrapped in a `Maybe`.
- `apply`, to apply a type to a parameterised record type.

Table 1 provides an exhaustive list of the type transformers included with the library, along with their arguments and example usage.

TypeScript does not provide a type transformer to easily build records with fields that have different types. Instead, this can be done using the language's record syntax. This is not possible in Haskell, at least not in a straightforward manner. Making this possible in `type-machine` is future work.

The library also provides two infix operators, `<:>` and `<::>`, which allow applying/passing values as arguments to type transformers. `<:>` allows passing a type transformer to a type transformer that expects a `Type`. `<::>` is similar, but allows passing an existing data type through its name (prefixed by two single quotes). The reason why these operators exist is that we do not want to have duplicate type transformers that accept only names or nested type transformers. Instead, having these operators allows streamlining the library's API and avoid cognitive overhead. These infix operators are left-associative, meaning that `union <::> 'A <::> 'B` will be parsed as `(union <::> 'A) <::> 'B`, where the expression `union <::> 'A` is a partial application of the function `union`, which would have type `Type -> TM Type`.

For example, Figure 7 gives an example of how the `omit` function can be used. Given a list of keys and an already existing record type, it derives a new record type from `User`, without the `id` field.

`type-machine`'s API exposes all the utilities to needed to implement custom type transformers, allowing users to define their own (see Section 4.3 for an example).

```
$(defineIs ''UserId)

-- Generates the following
class IsUserId a where
    getId :: a -> Int
    setId :: Int -> a -> a
    toUserId :: a -> UserId

instance IsUserId UserId where
    getId = id
    setId newId obj = obj{ id = newId }
    toUserId userId = userId
```

Figure 8. Code generated by `defineIs`

2.3 Structural Subtyping

Now that we can derive record ADTs, we might want to easily use variants of a single ADT across functions. To do so, the library comes with two other Template Haskell function: `defineIs` and `deriveIs`.

2.3.1 Is Typeclasses. The `defineIs` function takes a record's `Name` as parameter and generates an `Is${Name}` typeclass. For each field of the record, the typeclass will define a getter and a setter method. It also defines a function to build said record from a value whose type implements the typeclass. For example, Figure 8 shows the code generated by `defineIs` for the `UserId` type from Section 2.2.

On the other hand, the `deriveIs` function takes two record `Name`s, say `A` and `B`, and generates an instance of the `IsB` typeclass using the definition of `A`. The generation can fail (aborting the compilation process) if any of the required fields are missing in the source type (e.g. if `B` is `UserId` and `A` has no field named `id`). Figure 5 shows an example usage of `deriveIs` (at line 17).

`deriveIs` relies on the field names of the *bigger* type to match those of the *smaller* one, forcing clashing names to be in the same scope. Thus, programmers will have to enable the `DuplicateRecordFields` GHC extension[7]. This non-intrusive and mature extension allows one to define ADTs with conflicting field names in the same module.

Thanks to the code generated by `defineIs` and `deriveIs`, we can define structural constraints for derived records. Figure 5 shows an example of this (at line 22): the `translateX` can be invoked with a `Vec3` value, as it has all the necessary fields from `Vec2` (namely `x` and `y`).

2.3.2 GHC's Polymorphic Record Field Selector. Depending on the user's needs, it might not be necessary to generate these `Is` typecasses. GHC provides a `HasField` typeclass, which allow setting a constraint on a record's field[8]. It does not require any extra boilerplate and allows

[7] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/duplicate_record_fields.html
[8] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/hasfield.html

Table 1. List of the type transformers included with the `type-machine` library

Type Transformer Name	Arguments	Example usage	
		Expression	Generated type
pick	[String], Type	`pick ["a", "c"] <::> ''A`	`{ a :: Int }`
omit	[String], Type	`omit ["b", "c"] <::> ''A`	`{ a :: Int }`
record	[String], Q TH.Type	`record ["a", "b"] [t|Int|]`	`{ a :: Int, b :: Int }`
intersection	Type, Type	`intersection <::> ''A <::> ''B`	`{ b :: Int }`
union	Type, Type	`union <::> ''A <::> ''B`	`{ a :: Int, b :: Int, c :: Void }`
require	[String], Type	`require ["a"] <::> ''C`	`{ a :: Int, b :: Int }`
partial	[String], Type	`partial <::> ''C`	`{ a :: Maybe Int, b :: Maybe Int }`
apply	Q TH.Type, Type	`apply [t|Int|] <::> ''D`	`{ a :: Maybe Int }`

with:

```
data A = A { a :: Int, b :: Int }          data C = C { a :: Maybe Int, b :: Int }
data B = B { b :: String, c :: Void }      data D x = D { a :: Maybe x }
```

where 'Type' refers to `type-machine`'s definition of a data type, and TH.Type referes to Template Haskell's,

```
data Type = Type
{ fields :: Map String BangType
  -- ^ Fields of the data type
, typeParams :: [(String, Maybe Kind)]
  -- ^ Type parameters of the ADT
}
```

Figure 9. Definition of `type-machine`'s `Type` type

for more fine-grained restrictions, as it sets constraints on a single field, and not a group of fields like the `Is` typeclass.

However, solely using the `HasField` typeclass to simulate structural subtyping comes with a drawback: it does not allow updating a record's field (It is possible however, to define a custom `HasField` class that provides a method to update a field[9]).

3 Implementation

In this section we present the technicalities behind `type-machine`, from the modeling of record types (the `Type` ADT), to the design of the computation model (the `TM` monad). We will also present the flow of the code generation.

3.1 The Type type

In `type-machine`, type transformers are functions that return a `Type` value, which represents the record type to generate. This data type, defined in Figure 9, contains two fields.

The first one, named `fields`, represents the fields of the record, along with their type and strictness (`Bang-Type`). The second field, `typeParams`, represents the type parameters (and their kind, using `Kind`) of the record type. Both `BangType` and `Kind` are provided by Template Haskell's API.

```
type TM a = WriterT [Log] Q a
```

Figure 10. Definition of the `TM` monad

```
./Example.hs:19:2: warning: [GHC-39584]
TypeMachine: Field 'IAMERROR' not in type.

$(type_ "User2" (omit ["IAMERROR"] <::> ''User))
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Figure 11. Example of a warning issued by a type transformer

3.2 The TM Monad

Type transformers are computations in the `TM` monad (short for Type Machine). Its definition (given in Figure 10) is straightforward: `TM` is a writer monad nested inside a `Q` computation (the main Template Haskell monad). It produces a list of `Logs` (an alias to `String`), representing warning messages that can be issued during the computation.

Wrapping the writer monad in `Q` allows any `TM` computation to use Template Haskell's features, like type reification and quasiquotations (used by the `apply` function). To run a `TM` computation, we use the `runTM` function. It runs the inner `Writer` computation and retrieves the logs and the produced value. It calls the `reportWarning` function, from the `template-haskell` library to prints these logs as warnings[10]. To illustrate the use of these log messages, we can take the example of the `omit` function: passing a field name that does not exist in the given type should not result in an error; instead, the user should be alerted that it will result in a no-op. Figure 11 gives an example of such warning.

[9] See https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/overloaded_record_update.html

[10] Also, depending on the user's tooling, these logs can be reported in their IDE

```
1  type_ :: String -> TM Type -> Q [Dec]
2  type_ newTyName tm = do
3      resTy <- runTM tm
4      return [typeToDec newTyName resTy]
```

Figure 12. Implementation of the `type_` function

3.3 Code Generation Flow

Thanks to the `Type` type and the `TM` monad, the implementation of the `type_` function (given in Figure 12) is straightforward.

As we saw in Section 2.1, the function takes two parameters: the name of the record to generate and a `TM` computation that produces a `Type`.

First, at line 3, we evaluate that computation using `runTM` and retrieve the resulting `Type`. Then, we convert our `Type` value to a Template Haskell *declaration* (i.e. a `DataD`), using the `typeToDec` function (line 4). When *return*-ing this declaration (line 4), the code generation will start, and the derived type will be injected at the call site of the `type_` function.

4 Examples and Use Cases

In this section, we present three use cases for `type-machine`. The first two are in the context of web development, where one might need to quickly compose data types, while the third shows how to use `type-machine`'s API to create a custom type transformer. For all three examples, the expressiveness of `type-machine` allowed writing the code more concisely, compared to if they were written using other HList-based libraries. Using the latter would have noticeably increased the size of the code and impacted its readability.

4.1 Composable Component Props

In front-end development, displayable elements usually take the form of *components* which can be customised through *properties* (or *props*) passed as parameter. It is not unusual to compose components: a component A could actually use a component B, passing it only some of its properties, and expect the user to provide the rest of these properties.

To illustrate this, we are going to use the Miso front-end development framework[11], and define a `button` component as in Figure 13a. It has three properties: a colour, a size, and a label.

We might want to define a component that lays out buttons in a row with preset colours and sizes. Thus, we only require the user to provide labels for these buttons. We could define a `buttonRow` component, whose property is a list of the properties needed to build each button. We define this component using `type-machine` to derive its props from `button`'s (see Figure 13b).

Composing properties that way improves the maintainability of such code bases, as modifying the definition of `ButtonProps` would automatically change the definition of `ButtonRowItemProps`, leaving the programmer only to update the parameters to pass to each `buttonRow` component.

[11] https://github.com/dmjio/miso

```
data Color = White | Black | Grey
    deriving (Show)
data Size = XS | SM | MD | LG

toPixel :: Size -> Int

data ButtonProps = ButtonProps
  { label :: String
  , color :: Color
  , size :: Size
  }

button :: ButtonProps -> View a
button p = button_
  [ style_ $ M.fromList
    [ ("color", pack (
        toLower . show <$> color p))
    , ("padding", pack $ printf "%spx"
        (toPixel (size p)))
    ]
  ]
  [text $ pack (label p)]
```

(a) Defining a base button component

```
$(type_ "ButtonRowItemProps"
    (omit ["size", "color"]
          <::> ''ButtonProps))

type ButtonRowProps = [ButtonRowItemProps]

buttonRow :: ButtonRowProps -> View a
buttonRow props = div_ [] (props <&>
  (\p -> button (ButtonProps{
    label = label p,
    size = LG, color = White
  })))
```

(b) Defining a button group component with derived props

Figure 13. Defining and composing component props in front-end web development

4.2 Ensuring Consistency across Data Models

In a web application, it is common to have multiple derivations of a same model. For example, there could be one model to represent an item in the database, one to define the structure of an endpoint's response, and one to define a form to create an item. Although they are not identical, these models still share a common, base structure. Without any automated derivation system, updating one of these models means having to manually update the others. We could use `type-machine` to reduce this maintenance cost.

```haskell
data UserRecord = { id :: Int,
  name :: String, password :: String }

$(type_ "UserForm"
  (omit ["id"] <::> ''UserRecord))
$(type_ "UserResponse"
  (omit ["password"]
    <::> 'UserRecord))

$(defineIs ''UserResponse)
$(deriveIs ''UserResponse ''UserRecord)

deriving instance FromJSON UserForm
deriving instance ToJSON UserResponse
```

Figure 14. Defining and deriving a `User` model

Consider Figure 14 where we define models for a user management application. The `UserRecord` (line 1) is the database model. We derive it to get the `UserForm` model (line 4). The latter does not contain the `id` field, as it would not make sense to ask for a unique numeric identifier when creating a user. Then, we define a `UserResponse` model (line 6) which defines the structure of the response that the application will send when one is requesting a `UserRecord`. For obvious security reasons, we do not want to include the user's password in the response, so we use the `omit` type transformer to remove it from the response's structure.

At lines 10 and 11, we use `defineIs` and `deriveIs` to define that `UserRecord` is 'a subclass of' or 'has more fields than' `UserResponse`. This will generate a typeclass named `IsUserResponse`, which provides the `toUserResponse` function, which can turn a `UserRecord` into a `UserResponse`. We also derive instances of the `FromJSON` and `ToJSON` typeclasses from the `aeson` library[12], which allow converting models from/to JSON.

A possible implementation of a web server using these models is given in Figure 15. Using the `servant` library[13], we define two endpoints (line 1-9): one that accepts POST requests with a JSON-encoded `UserForm`, and one that looks up a record using its id. In the `createUser` endpoint handler (line 12), we call `saveUserRecord` to persist the user's information received from the `UserForm`. Note that we had to build a `UserRecord` by hand (line 29) because `UserForm` 'has fewer fields than' `UserRecord`[14]. However, at line 16, we avoid converting records manually by using the `toUserResponse` function generated earlier. The `getUser` endpoint handler is more straightforward (line 20): we simply get a `UserRecord` from the database and use `toUserResponse` to build the appropriate response.

[12] https://hackage.haskell.org/package/aeson
[13] https://hackage.haskell.org/package/servant
[14] Alternatively, we could have used the `RecordWildCards` Haskell extension to simplify this step.

```haskell
type UserApi =
       ReqBody '[JSON] UserForm
       :> Post '[JSON] UserResponse
  :<|>
       ":id" :> Capture "id" Int
       :> Get '[JSON] UserResponse

server :: Server UserApi
server = createUser :<|> getUser

-- Endpoints handlers
createUser userForm = do
    userRec <- saveUserRecord userForm
    let response =
        toUserResponse userRec
    return response

getUser userId = do
    userRecord <- getUserRecord userId
    let response =
        toUserResponse userRecord
    return response

-- Database
saveUserRecord :: UserForm
    -> Db UserRecord
saveUserRecord userForm = do
    newId <- getNextId
    let record = UserRecord newId
        (name userForm)
        (password userForm)
    save record
    return record

getUserRecord :: Int -> Db UserRecord
getUserRecord = getByPrimaryKey
```

Figure 15. A web server implementation using `type-machine` to define the data types

4.3 Renaming Fields

In some contexts, it is preferable to define two separate data types even though they have the same fields. For example, in the context of K-means clustering, we deal with points and centroids. Although centroids are not points, they both are defined by coordinates. Thus, with `type-machine`, we could use a `Point` data type to derive a `Centroid` ADT.

In this example, we will use a two-dimensional space. Thus, we could give `Point` the following definition (which could also have been generated with the `record` type transformer):

```haskell
data Point = Pt { x :: Int, y :: Int }
```

Since centroids and points share a similar structure, we could either define `Centroid` as a type synonym, or derive it from `Point` using `type-machine`. We could imagine that, for clarity reasons, one might want to prefix the centroid's

```haskell
import Data.Map.Strict (mapKeys)

rename ::
    (String -> String) -> Type -> TM Type
rename renameField ty =
    return (ty{fields = renamed})
  where
    renamed =
        mapKeys renameField (fields ty)
```

Figure 16. Implementation of a custom type-transformer that renames the fields of the given record type

fields names with 'centroid_'. Thus, we aim to generate the following type:

```haskell
data Centroid = Ctrd { centroid_x :: Int
                     , centroid_y :: Int }
```

To achieve this, we could define a custom type transformer, named `rename`, which takes as parameter a function that returns a field's new name (i.e. with type `(String -> String)`).

A possible implementation of this type transformer is given in Figure 16. The implementation is quite simple. We simply use the `mapKeys` function from `Data.Map` to apply the renaming function to each field name and `return` the transformed record.

We could extend this function to add some error handling. For example, we should check for duplicated or empty field names. To abort the execution of the type-transformer, we could use the `fail` function, from the `MonadFail` typeclass, which `TM` is an instance of. (We leave this as an exercise to the reader).

To invoke the type transformer and generate our `Centroid` record, we would use the following statement:

```haskell
$(type_ "Centroid"
    (rename ("centroid_" ++) <::> ''Point))
```

5 Benchmarks

Since `type-machine` generates native ADTs, building and traversing records defined with the library should theoretically be faster than for records defined by the libraries mentioned in Section 1. However, since Template Haskell entails an additional step in the compilation process, we suspect that programs that depends on `type-machine` might take longer to compile.

In this section, we test these intuitions by comparing the time it takes to build and traverse records defined by `type-machine`, `extensible` and `superrecord`, as well as the time it takes to compile small programs that use these libraries. We should note that we could not run our benchmarks on the `records` library, as we did not manage to compile it with a modern Haskell setup. While we only evaluate the performance of `type-machine` through microbenchmarks, evaluating the speed-ups enabled by the library in real-world applications is future work.

Table 2. Execution times for building records with 5 fields

Library	type-machine	extensible	superrecord
Build Time	11.28 ns	12.65 ns	14.54 ns

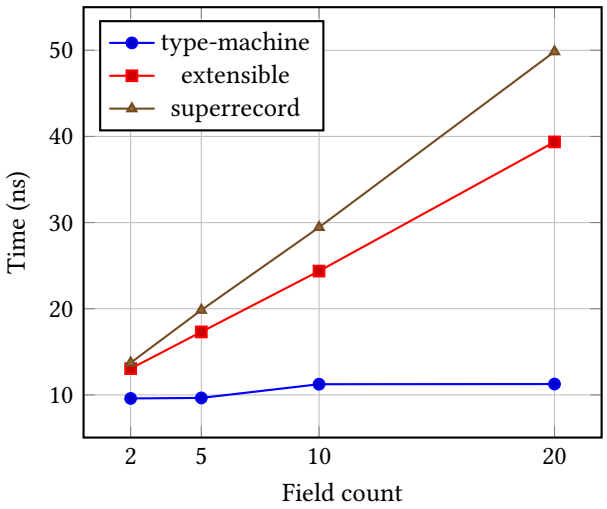

Figure 17. Execution times for traversing records of various sizes

The benchmarks were run on an Intel machine with two Xeon Gold 6244 CPUs at 3.60 GHz, with 32Gb of RAM, running Ubuntu 22.04 LTS. We used the version 0.9.2 of `extensible` and version 0.5.1.0 of `superrecord`. Code was compiled with GHC 9.10, using the O2 optimisation flag. We ran our benchmarks using the `criterion` library[15] which runs each case as many times as possible within 5 seconds, and computes the mean time and standard deviation.

5.1 Runtime Performance

To evaluate the runtime performance of these libraries, we defined, using each library, isomorphic record types with 2 to 20 fields, all with type `Int`. Table 2 and Figure 17 present the time it takes to build and traverse these records respectively. Traversals will consist of summing the values of each field of the records.

First, we notice that the time it takes to build the records is overall similar across the three libraries. `extensible` and `superrecord` seem to be slightly slower than `type-machine` (1.12x and 1.28x, respectively). This gap seems to be linear with the number of fields.

On the other hand, the traversal benchmarks really highlight the computing overhead caused by `extensible` and `superrecord`'s use of HLists. Data types defined using these

[15] https://hackage.haskell.org/package/criterion

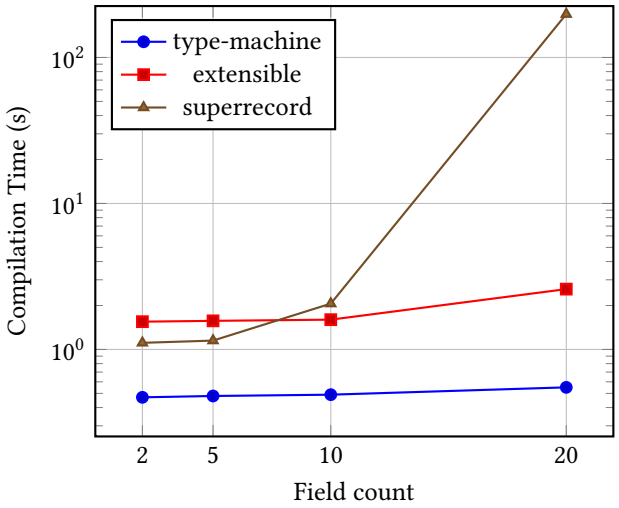

Figure 18. Compilation times for programs defining records

libraries are at least 1.4x slower to traverse than the ones defined using `type-machine`. The speed-ups enabled by `type-machine` are also linear with the number of fields.

This confirms our intuition that since `extensible` and `superrecord` do not create native Haskell ADTs, their records are consequently slower to use. This can be explained by the fact that since HLists are not first-class citizens to the compiler, the latter cannot deal with them as easily as with native ADTs.

5.2 Compilation Time

To evaluate the impact of `type-machine` on compilation times, we write programs that define records with 2, 5, 10 and 20 fields, along with a traversal function, using `type-machine`, `superrecord` and `extensible`. Figure 18 presents the time it takes to compile these programs.

The results are surprising; it turns out that the `type-machine` programs are the fastest to compile. The `extensible` programs are at least three times slower to compile, and the `superrecord` programs can take up to 3m18s to build, 360x more time than the `type-machine` ones. We suspect that the type system is to blame for these longer compilation times, as `superrecord` and `extensible` use labels and type-level operators, whereas `type-machine` only relies on Template Haskell and compile-time reflection.

Conclusion. In conclusion, programs that use `type-machine` to generate record types can leverage faster compilations times as well as faster record manipulations, compared to `superrecord` and `extensible`. This is mainly because `type-machine` generates native records, without any advanced features of the type system like type-level operators or labels.

6 Limitations and Future Work

In this section, we go through some limitations of `type-machine`, whether it is from the applicability point-of-view, from the implementation of the existing type transformers, or from the tight coupling with Template Haskell. We also present possible future work.

Building Records of Fields with various Types. There is no type transformer to easily build records with fields that have different types. Instead, users would have to invoke the `union` and `record` type transformers for each field. This is unwieldy. Providing an easy alternative is future work. For example, we could use quasi-quotation to allow defining such records using Haskell's native syntax.

Applicability. Since `type-machine` is focused on record types, it is only natural to limit the kinds of ADT that can be derived using the `type_` function: only record ADTs with exactly one constructor will be accepted. We consider this a reasonable restriction, as most record types only have one constructor, and since the library works with named fields, accepting non-record ADTs would not make sense.

Field Disambiguation. With our will to simulate structural subtyping, it is easy to end up with multiple records with clashing field names within the same module. Thankfully, it is not a problem if the user enables the `DuplicateRecordField` GHC extension. However, there is another shortcoming to consider: as of the writing of this paper, the compiler is not able to disambiguate record fields when they are used as functions. Thus, the following code snippet does not type-check:

```
data A = A {a :: Int}
data B = B {a :: Int}

getIntFromA :: A -> Int
getIntFromA obj = a obj
```

The compiler complains that `a` on the last line is ambiguous, even though the type of `obj` is not. Instead, we must implement `getIntFromA` using destruction and selection with the field's name:

```
1  getIntFromA :: A -> Int
2  getIntFromA A{a} = a
```

Surprisingly, at line 2, `a` is correctly typed as `Int`.

Simulating Subtyping and Lenses. In addition to simulating subtyping through the code generated by the `defineIs` and `deriveIs` functions, we could rely on the popular Haskell library `lens`[16]. It uses Template Haskell to generate lenses, which are basically pairs or getters and setters for

[16] https://hackage.haskell.org/package/lens

record fields. It provides the `makeClassy` Template Haskell function which behaves like `defineIs` but uses lenses instead of separate getters and setters. The library also provides the `makeFields` which generates typeclasses similar to `HasField`, for each field of the given data type, again using lenses. However, the library does not come with a way to derive instances, like `deriveIs`.

Finally, a companion library to `lens`, `generic-lens`[17], provides its own `HasField` typeclass. Instances for this typeclass are automatically derived by the compiler, but using it requires using a syntax similar to GHC's `HasField`:

```
getIntFromA :: A -> Int
getIntFromA a = a ^. field @"a"
```

Using `lens`, we could simplify the `Is` typeclasses, but on the other hand, it requires the user to rely on another dependency and use lenses to interacts with fields.

Support for Non-Record ADTs. The library needs a way to identify fields. Currently, it only relies on their names to identify them. Thus, only record types can be used with the library. However, we could support non-record ADTs, by using the fields' type and position in the declaration to identify them. In case of ambiguity (e.g. two fields with the same type), we could ask the programmer to provide a list of string identifiers to name fields, or require the use of custom pragmas to label fields.

More generally, subtyping for ADTs with unnamed fields seems to be an unexplored topic. Even the `lens` library provides limited support for them. Being able to access/modify unnamed fields from data types from unstable/complex APIs like GHC's or Template Haskell's could help maintain code that use them.

7 Conclusion

In this paper, we present a way to compose record types and simulate structural subtyping in Haskell using meta-programming and code generation. Our Haskell library, `type-machine`, takes inspiration from TypeScript's type-level functions. Since our library generates native Haskell records, their use does not entail any runtime overhead. Additionally, the library comes with a small array of type transformers and exposes a simple API that allows programmers to define their own.

As far as we are aware, `type-machine` is the first library to leverage meta-programming in order to generate and compose native Haskell record types.

Our microbenchmarks show that records defined using `type-machine` can be up to 28% faster to build and at least 40% faster to traverse than records defined using `super-record` or `extensible`. Moreover, simple programs that define data types using `type-machine` are at least 3x faster to compile than isomorphic programs that use these libraries. We suspect it is because our approach does not rely on the type system as much as these libraries do. Further evaluation of the library in real-world applications is future work.

Future work on the library includes making the structural subtyping utilities (`Is` and `HasField`) more usable and flexible, potentially using lenses. Additionally, we could add support for non-record ADTs by comparing their fields' types and position in the constructor's definition.

[17] https://hackage.haskell.org/package/generic-lens

References

[1] 2025. *TypeScript Documentation: Type Compatibility.* https://www.typescriptlang.org/docs/handbook/type-compatibility.html Accessed: 22/05/2025.

[2] Luca Cardelli. 1984. A semantics of multiple inheritance.. In *Proc. of the International Symposium on Semantics of Data Types* (Sophia-Antipolis, France). Springer-Verlag, Berlin, Heidelberg, 51–67.

[3] Luca Cardelli. 1994. *Extensible records in a pure calculus of subtyping.* MIT Press, Cambridge, MA, USA, 373–425.

[4] Benedict R Gaster and Mark P Jones. 1996. *A polymorphic type system for extensible records and variants.* Technical Report. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University ....

[5] Wolfgang Jeltsch. 2010. Generic record combinators with static type checking. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming.* 143–154.

[6] Mark P Jones and Simon Peyton Jones. 1999. Lightweight Extensible Records for Haskell. In *Haskell Workshop.* ACM, ACM. https://www.microsoft.com/en-us/research/publication/lightweight-extensible-records-for-haskell/ Paris.

[7] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell.* 96–107.

Received 2026-02-27; accepted 2026-04-23