University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Jacek Olczyk

Student no. 385896

Default values in Haskell record fields

Master's thesis
in COMPUTER SCIENCE

Supervisor:
PhD. Josef Svenningsson
Chalmers University of Technology

PhD. Marcin Benke
Instytut Informatyki

Warsaw, X 2022

Abstract

This thesis presents a prototype implementation of the syntax and semantics for specifying the default values for fields of record types in Haskell. The proposed syntax aims to have no impact on existing Haskell code and minimal syntactic intrusion when the proposed feature is used. The use of default values can and will happen only when using the braced record construction syntax already present in Haskell. The syntax for defining default values for fields follows recognizable conventions from other programming languages in which this feature is present. This feature facilitates more seamless integration of Haskell codebases into multi-language codebases unified by interface definition languages such as Apache Thrift.

Keywords

functional programming, Haskell, TODO

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatyka

Subject classification

D. Software
D.127 TODO pick candidate
D.127.6. Numerical blabalysis

Tytuł pracy w języku polskim

Domyślne wartości pól rekordów w Haskellu

Contents

Introduction

Algebraic data types have been a staple of functional programming languages since their earliest days. A typical extension of their utility for everyday programming is the record syntax, present in most popular functional programming languages. However, a notable functionality often missing from that feature is the ability to provide default values for individual record fields. While widely used workarounds are known, they suffer from various problems that a compiler-assisted solution can avoid.

This work aims to solve these problems in the case of the Haskell language by proposing a custom syntax inside record definitions that enables the programmer to provide a default value to any field of a given record type. In the beginning, we further detail the background and motivation behind these changes. Then, we outline the syntactic changes applied to the grammar of Haskell alongside alternative solutions and reasoning behind the final choice. Following that, we explain in detail the semantic behavior of introduced features and the purely semantic changes to existing constructs. We also explain the development process and decisions made when implementing the feature in GHC, the leading compiler of the Haskell language. Finally, we investigate the performance impact of the change on the runtime of the compilation using GHC as well as the compiled programs themselves. The latter is accomplished by running benchmarks on [jo: TODO].

Chapter 1

Background

1.1. Overview of record syntax

A major drawback algebraic data types usually face when contrasted with data types commonly found in imperative languages (like `struct` in C) is their unwieldiness when it comes to data structures containing many fields. While most imperative and virtually all object-oriented languages feature a prominent and easy-to-use syntax for both naming and accessing individual fields of their data structures, the default for most functional languages has been to prefer data structures with unnamed fields. However, as projects grow larger and larger, having only unnamed data structure fields becomes increasingly cumbersome and error-prone. Thus, most functional languages provide an alternate record syntax for defining their data types.

In most cases, record syntax provides two important features: the ability to name individual fields of a data structure; and the automatic generation of field selector functions which absolves programmers from having to write the otherwise necessary, but cumbersome boilerplate code. [jo: example at https://en.wikibooks.org/wiki/Haskell/More_on_datatypes]

1.2. Field defaulting

1.2.1. Introduction

Despite their relative ubiquity in programming languages, most instances of record syntax have an important usability feature missing when compared to their imperative counterparts — the ability to leave out certain fields empty when creating a concrete object of a given type. Especially when dealing with data structures containing a great number of fields, e.g. types that represent a configuration file, the necessity of providing every field of the structure with a value can become unwieldy very quickly.

1.2.2. Prior attempts at solving the issue

This issue has been heretofore partially mitigated thanks to another common feature of record syntax in programming languages: the record update syntax. It allows for the construction of a record in such a way that any unspecified fields are copied over from an already existing record. If the author of a type then provides their users with a globally visible "default" object with all its fields already filled out, a user can utilize the record update syntax to mimic the desired functionality.

However, this approach is not without its drawbacks. Firstly, it can prove a challenge if some fields don't lend themselves easily to default values. For example, a data type might have an ID component along with invariants guaranteeing its uniqueness. In such a situation, great care must be taken when creating a new record to ensure that the ID component is always modified from the value provided by default. It is also possible to circumvent that using optional types (like Haskell's `Maybe`), but this makes the data structure cumbersome to use for anything other than its creation. Unlike it is for its imperative counterpart, this approach does not offer the programmer any automated systems that would detect the absence of such a value. Given that a major advantage functional languages tout over others is the strength of their automated compile-time error detection, this is an area in which clear improvement can be made.

In some languages, like Haskell, it is also possible to create a record without specifying all of its fields. However, in that case, the fields are still automatically filled with bottom values. To make matters worse, such a creation creates compile-time warnings when defining the default record, and no compile-time warnings when some of the bottoms are not replaced. Because of that, this solution is even worse than the previous one.

Secondly, the necessity of creating a custom object to use as a default can lend itself to improper coding style and more difficulty in codebase navigation. In order to create a default object, one must first decide on a name that it will be given, which in large codebases necessitates either coming up with a naming convention or dealing with the repercussions of inconsistent naming. Furthermore, there is no requirement for the programmers to place the default object in the same place, or even in the same file as the definition of the type itself. This can quickly make the issue of finding the default object a true detective's task.

1.2.3. Use cases

Aside from the obvious functionality improvements, allowing default fields in records has some impact on the overall Haskell ecosystem. Many large multi-language codebases, such as the giant monorepos used by Big Tech companies, use interface definition languages (IDLs) to interface between code written in different languages. IDLs usually provide this through a language-agnostic syntax for defining data structures. This syntax usually (as is the case in Apache Thrift) contains a way to provide default values for fields of the data structures being defined. Because of that, any language that doesn't natively support default field values in data structures will need to use workarounds before being able to interface with IDLs. [jo: TODO: Josef's more detailed background on FB's use case]

Chapter 2

GHC architecture

This section contains an abridged overview of the architecture of the GHC compiler to serve as background for implementation details provided later.

Chapter 3

Syntax

Among a wide variety of languages supporting providing default values to fields of data structures, there has been a virtually unanimous agreement on the syntax used for providing the default values. Any language that supports this feature on the compiler level seems to agree on the base syntax being `field_name = value`, with minor variations depending on the given language's syntax for field type annotations. Thus, declaring an integer field with default value of 1 in C++, Java, and likely many others, looks like this: `int x = 1;`. As another example, the same is accomplished in TypeScript with the syntax `x: number = 1,`.

3.1. Syntax for declaring a default field value

With that in mind, we needed to accommodate the `field_name = value` syntax to existing syntax for Haskell records. A single field in a Haskell record currently looks like the following:

```
fieldName :: FieldType
```

Considering the above examples, a few alternative approaches for the default value syntax come to mind.

- type after value

  ```
  fieldName = field_value :: FieldType
  ```

- type after name

  ```
  fieldName :: FieldType = field_value
  ```

- variants of both of the above, but allowing for omitting the type of the expression

In total four variants, and the one chosen here is the "type after value" approach without omitting the type signature.

3.1.1. Order of syntactic components

To decide between "type after value" and "value after type", we looked at language constructs already prevalent in existing Haskell code.

A regular top-level Haskell binding with a type annotation can have multiple forms:

1. standalone type annotation

   ```
   name :: Type
   name = value
   ```

2. type after value

   ```
   name = value :: Type
   ```

3. value after type

   ```
   name :: Type = value
   ```

By far, the most popular of these variants is 1, but it's not readily adaptable to record fields. To choose between 2 and 3, we note that 3 is very rarely used in real code. Furthermore, at first glance, it might seem as if the default value is being assigned to the type, not the field name. In contrast, 2 looks like a familiar construct — a binding name without a type annotation, and its right-hand-side expression with one (and it is already parsed as such by GHC).

3.1.2. Omitting the type signature

It would seem like a good idea to allow users to omit the type signature for a field with a user-supplied default value. After all, it seems possible to always infer the type from the defaulting expression. As it turns out, the reality is not that simple. Many tools that are part of the GHC project currently need to be able to present the type of all fields of a given datatype without the information provided by a full typecheck of the program. The list of these tools includes the typechecker itself. While the typechecker itself could theoretically be modified to avoid this problem, forcing a typecheck on other tools (such as Haddock) would unnecessarily slow down their performance without a noticeable gain. As providing type annotations is a good practice in almost all use cases and virtually all existing code already has to do so for all their record fields, this loss of functionality seems to have negligible cost.

3.2. Syntax for use of defaulted fields

The chosen method for initializing fields with their default value is through braced record syntax. Because the only other way to construct a record is through a regular constructor function, which cannot have optional arguments (just like all other Haskell functions), braced record syntax is the sole method of assigning default values to fields. Omitting a given defaulted field is the only requirement for activating the mechanism. Since omitting fields in record construction was already allowed (but produced a compile-time warning), no syntax change is necessary.

[jo: TODO: maybe talk about the possibility of defaulting multiple fields in one line, e.g. "a = 1, b = 2, c = 3 :: Int")]

3.3. Summary

Below is a complete example of a Haskell record type with one of its fields defaulted. In our view, the final syntax combines the best compromise between readability, new user learning

curve, and implementation viability.

```haskell
data NewRecordType
  = NewRecordTypeCtor
  { fieldName :: SomeType
  , fieldName2 = expression2 :: SomeType2
  }
```

No syntax change is necessary for the case of the usage of default values.

Chapter 4

Semantics

4.1. Examples

The semantics for the proposed extension of the language are very simple. Recall the (slightly renamed) example type from the previous chapter:

```haskell
data Record1
   = RecordCtor2
   { field1 :: Int
   , field2 = expression1 :: String
   }
```

In order to utilize the mechanism inserting default values, usage of the record construction syntax is necessary:

```haskell
exampleRecord :: Record1
exampleRecord
  = RecordCtor1
  { field1 = 12
  -- field2 is initialized with the default value
  }
```

Notably, this is the only possible way of constructing a record with a non-$\perp$ default value. Below we give non-examples of constructions that do not change their semantics with this change (but could reasonably be expected to). In the first example, the field is initialized with the $\perp$ value, both in the current version of Haskell and with the changes described in this thesis applied to GHC.

```haskell
data Record2
   = RecordCtor2
   { field1 :: Int
   , field2 :: String
   }

exampleRecord :: Record2
exampleRecord
  = RecordCtor2
  { field1 = 12
  -- field2 is initialized with the ⊥ value, raising an error when evaluated
  }
```

In the second example, we note that record update is not affected by these changes. Every unspecified field in the record update will be copied directly from the old record, even if its value is ⊥. The example record created by the following code is equal to `RecordCtor3 { field1 = 5, field2 = 4 }`.

```haskell
data Record3
  = RecordCtor3
  { field1 = 1 :: Int
  , field2 = 2 :: String
  }

baseRecord :: Record3
baseRecord = RecordCtor3 { field1 = 3, field2 = 4 }

exampleRecord :: Record3
exampleRecord
  = baseRecord
  { field1 = 5
  -- field2 is initialized using the value in baseRecord, not the default value
  }
```

4.2. A more formal statement

For precision, we offer a reasonably rigorous semantic description of field defaulting:

Description 1. Let T be a type with a record constructor T' containing a field f for which a default expression e has been provided. Then, if while calling the constructor T' via the record construction syntax, field f is not given any value, during evaluation the construction happens *as-if* field f had been specified and given the value e.

4.3. Notable observations

Given Haskell's usual semantics as a pure and lazy language, the defaulting expression will only be evaluated at most once, if it's needed. This allows for arbitrarily complex expressions without risking unnecessarily long runtime of the program. Even if the defaulted field's type is a computation inside the `IO` monad (and thus can be considered to have side effects), the evaluation of the expression only produces instructions for the runtime system without actually executing them. Thanks to that we can be certain that the default values will behave as expected.

4.4. Cross-module behavior

[jo: TODO pending further investigation]

Chapter 5

Implementation

We implemented the proposed change to the language inside the leading Haskell compiler, GHC (Glorious Haskell Compiler). The algorithm is a journey that takes the default expression from being declared to being substituted when a field is missing. It consists of the following main steps:

1. Parse the defaulting expression inside the constructor.

2. Give it a unique name and group it together with top-level bindings so that dependency analysis, renaming and typechecking are performed without disturbances.

3. Mark defaulted fields as not actually missing.

4. During desugaring to Core, replace any missing fields with their default values.

The necessary changes made to the GHC code can be broken down into X parts:

1. Changes to datatypes

2. Parser

3. Renamer

4. Desugarer

5.1. Changes to datatypes

[jo: Maybe the sections below could use references to specific files?]

[jo: Now I realize that the fields should have been called default expressions in the code, not initializing expressions...] For the initial prototype we tried to keep the changes to existing internal GHC data structures to a minimum. First necessary change was to the `ConDeclField` type, which represents a single field of a record constructor. We modify it by adding a field called `cd_fld_ini` of type `Maybe (LHsExpr pass)`. The `L` prefix in `LHsExpr` means that the expression contains information about its location in the source file. The `pass` parameter signifies the current phase of the compilation [jo: ref. Trees that grow]. This modification allows us to pass the defaulting expression from the parser to the renamer.

The renamer then moves information about types and their constructors from the data structures that represent abstract syntax to internal structures that are easier to work with. There, information about record fields is stored inside the `FieldLabel` type, to which we added a new field, `flIniExpr :: Maybe Name`. The `Name` refers to the generated name of the binding generated for the expression.

5.2. Parser

Parser uses Happy, a grammar-based parser generator for Haskell. The existing grammar node for record fields is called `fielddecl`. Its only production is `fielddecl : sig_vars '::' ctype`. The above parses a comma-separated list of variable names, followed by the `::` operator, followed by the type of the field(s). The names are a list to allow for declaring multiple fields with the same type in one line, e.g. `a, b, c :: Foo`. We want to modify it to allow for `= expr` between the name of the field and the type. One obvious solution would be to add another production producing `var '=' exp '::' ctype` (as we only want to allow a single name to be defaulted on one line). This unfortunately produces a reduce/reduce conflict, as `exp` can already contain a type signature. We solve this conflict by extracting the production for an expression with a type signature to a separate node, `typedexp`. This allows us to parse `var '=' typedexp`, which forces a type signature and guarantees no conflicts. The aforementioned production is introduced as a new node, `varini`, for clarity.

5.3. Renamer

5.3.1. Preprocessing

Before we start the renaming process, we want to make sure we are making as much use of existing GHC piping as possible. Thus, at the beginning of the renamer, we extract all of the default values for fields into separate bindings. In the type declarations, the user-provided expressions are then replaced (inside the `ConDeclField` structures) with simple variable expressions. The user-provided expressions are given freshly generated names and grouped together with other value declarations for the module. All the declarations are fed to the renamer in this way, and there the next phase of renaming proceeds.

5.3.2. Conversion

Early in the renaming process, the abstract syntax for type declarations is parsed into more convienient data structures. It is here that record selector functions are created and their names are placed inside the `FieldLabel` objects created alongside them. This means that it's the perfect place for inserting the names of the default expressions into the `FieldLabel`s. To do that, before checking individual constructors or fields, a mapping is constructed of all field names from a given type declaration to the names of their default expressions.

5.3.3. Missing fields detection

The fields that have been given default values should be excluded from both the list of missing strict fields (which produces an error when nonempty) and the list of missin nonstrict fields (which only produces a warning). When detecting if fields are missing, the algorithm compares the available `FieldLabel`s for the given constructor with the occurrences provided in a given expression. Thus, to exclude defaulted fields, it is sufficient to filter out the `FieldLabel`s that have their `flIniExpr` field set to a `Just` value.

5.4. Desugarer

During desugaring, we need to fill any unspecified record fields with the variable name corresponding to the appropriate default expression. Normally, the desugarer fills all missing

fields with error ($\perp$) values. Given that the desugarer has access to the `FieldLabel`s of the arguments being desugared, we can check if they contain the default expression and desugar the field to contain the variable name instead of an error.

Chapter 6

Benchmarks

[jo: TODO]

Bibliography